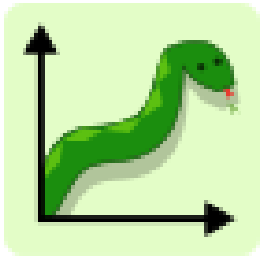
PyMeasure Documentation

Release 0.15.1.dev366+gcb643a9c2.d20250827

PyMeasure Developers

Aug 27, 2025

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:

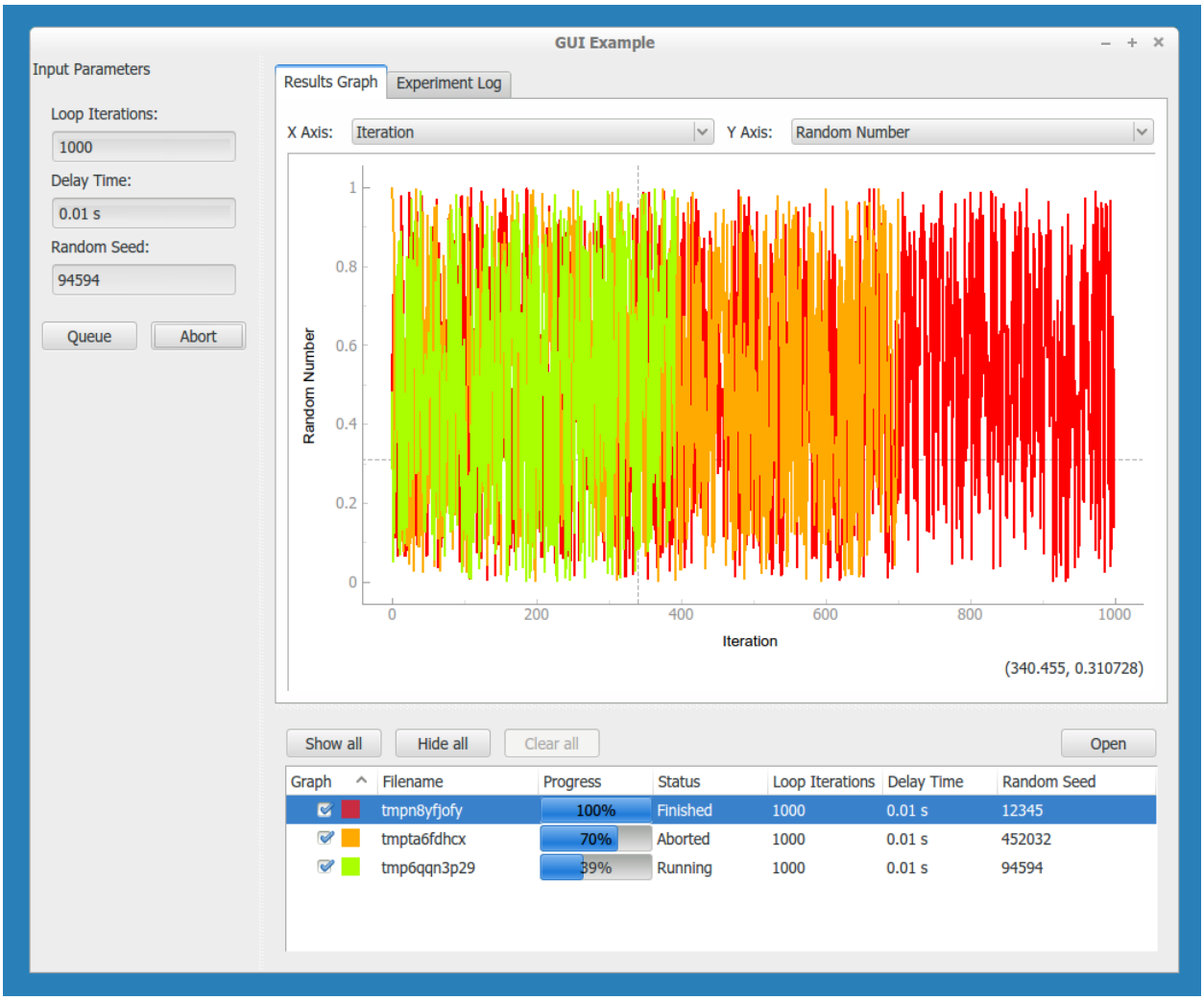
```
from pymeasure.instruments.keithley import Keithley2400
```

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 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 manager 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.resources.list_resources()` function to list all available resources:

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

If you know the USB properties (vendor id, product id, serial numer) of the serial device, you can query for the VISA resource string:

```
from pymeasure.instruments import find_serial_port
resource_name = find_serial_port(vendor_id=15, product_id=0x12e5, serial_number="sn56X
↪")
```

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
```

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.

Note

Instead of emitting results one by one, it is also possible to emit results in batch. As an example consider a device that returns multiple points for each measurement (such as an oscilloscope or a CCD):

```
intensities = self.ccd.capture()  # Assume this function returns a list of␣
↪intensities for each pixel
pixels = np.arange(len(intensities))

for pixel, intensity in zip(pixels, intensities):
    self.emit('results', {'Pixel': pixel, 'Intensity': intensity})
```

The downside to this method is that it is cumbersome to write and can be slow when the array of data is large. Instead it is possible to emit the data as as a whole:

```
intensities = self.ccd.capture()  # Assume this function returns a list of␣
↪intensities for each pixel
pixels = np.arange(len(intensities))

self.emit('batch results', {'Pixel': pixels, 'Intensity': intensities})
```

Please note that you have to use `'batch results'` as the topic when emitting results this way.

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):
```

(continues on next page)

(continued from previous page)

```
        """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)
```

(continues on next page)

(continued from previous page)

```
        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 run; 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')
```

(continues on next page)

(continued from previous page)

```
    # 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 run 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)
```

(continues on next page)

(continued from previous page)

```
    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,
                '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")
```

(continues on next page)

(continued from previous page)

```
    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-fledged 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()
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            self.emit('progress', 100 * i / self.iterations)
            sleep(self.delay)
            if self.should_stop():
```

(continues on next page)

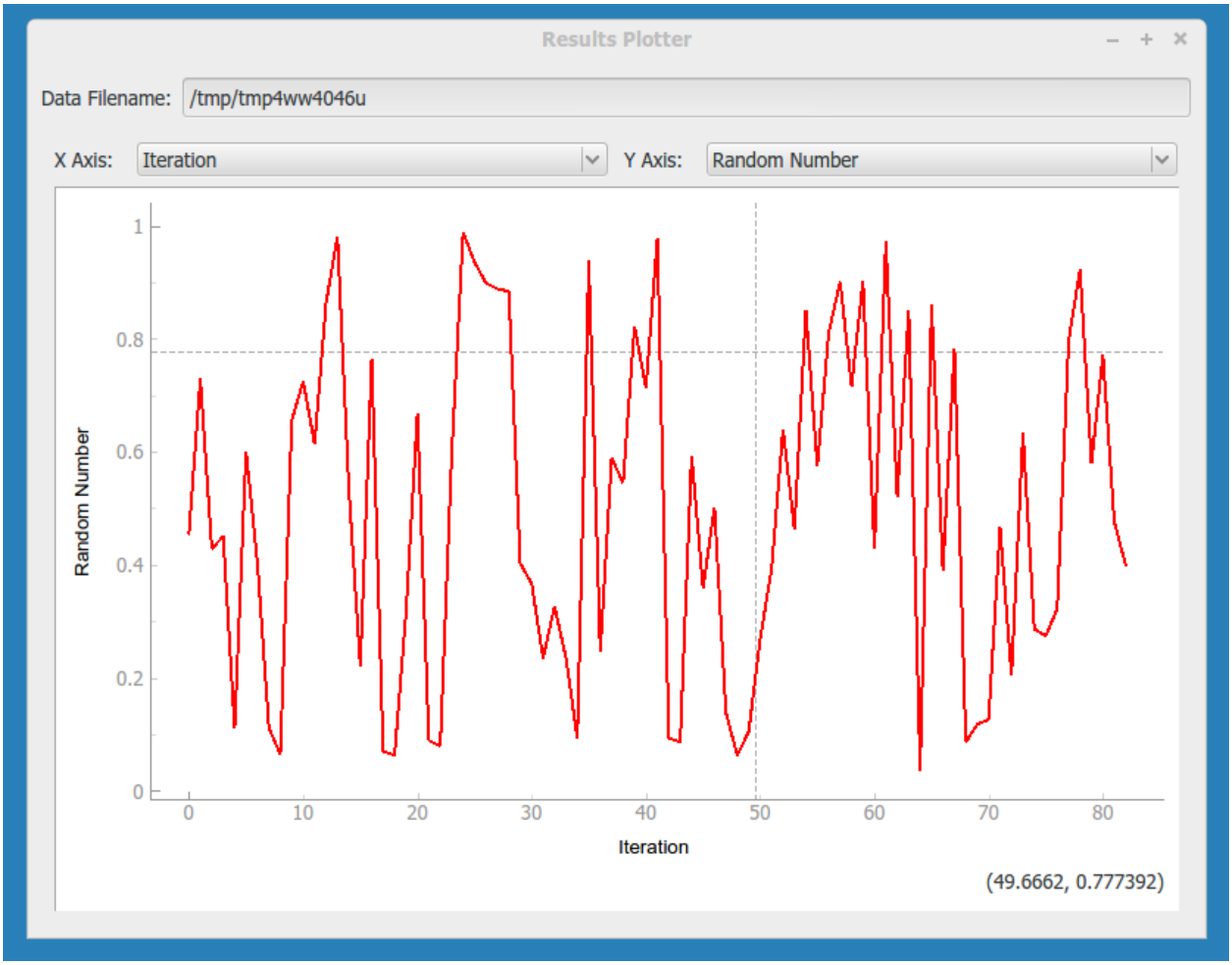
(continued from previous page)

```
                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.

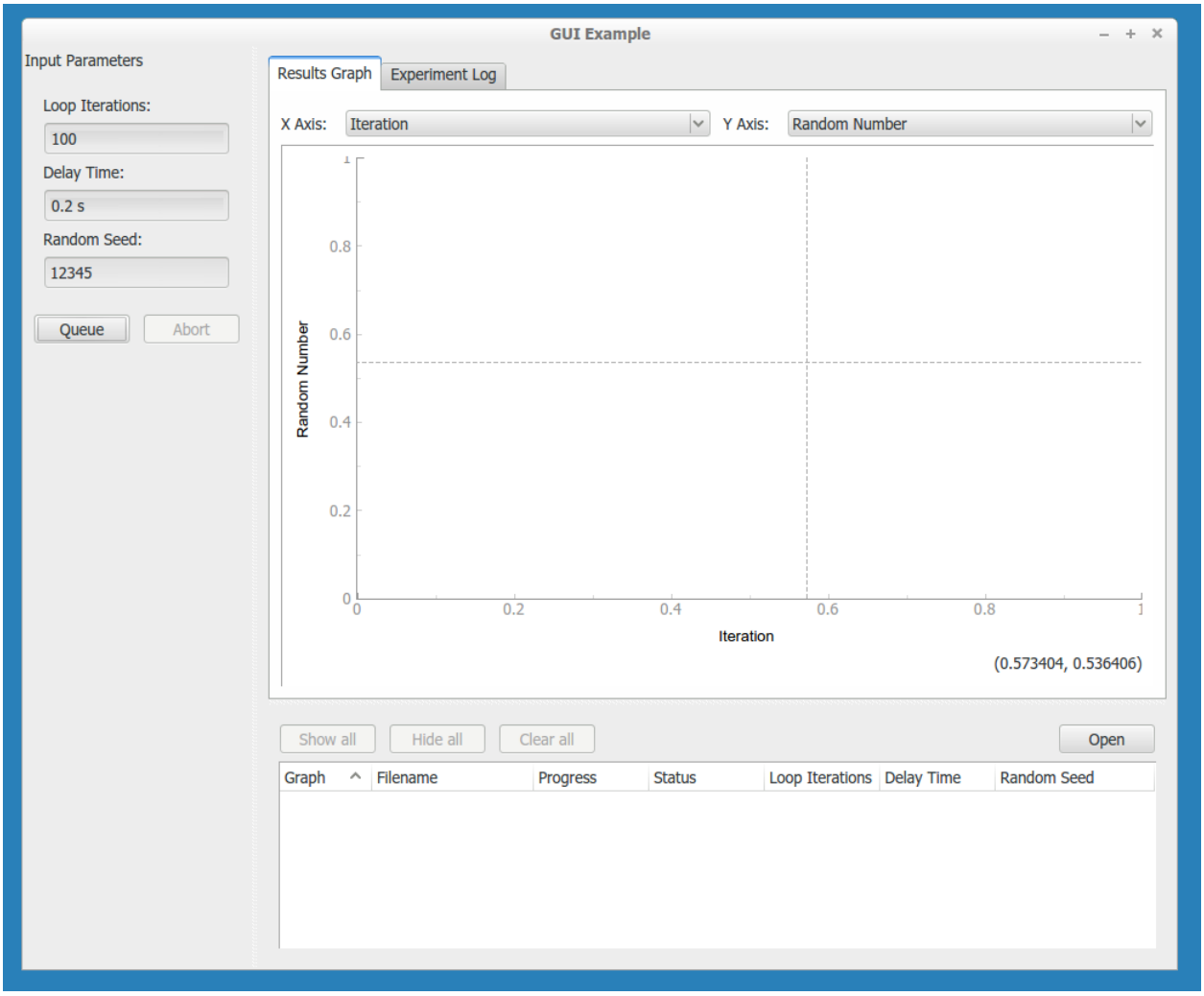
```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

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

(continues on next page)

(continued from previous page)

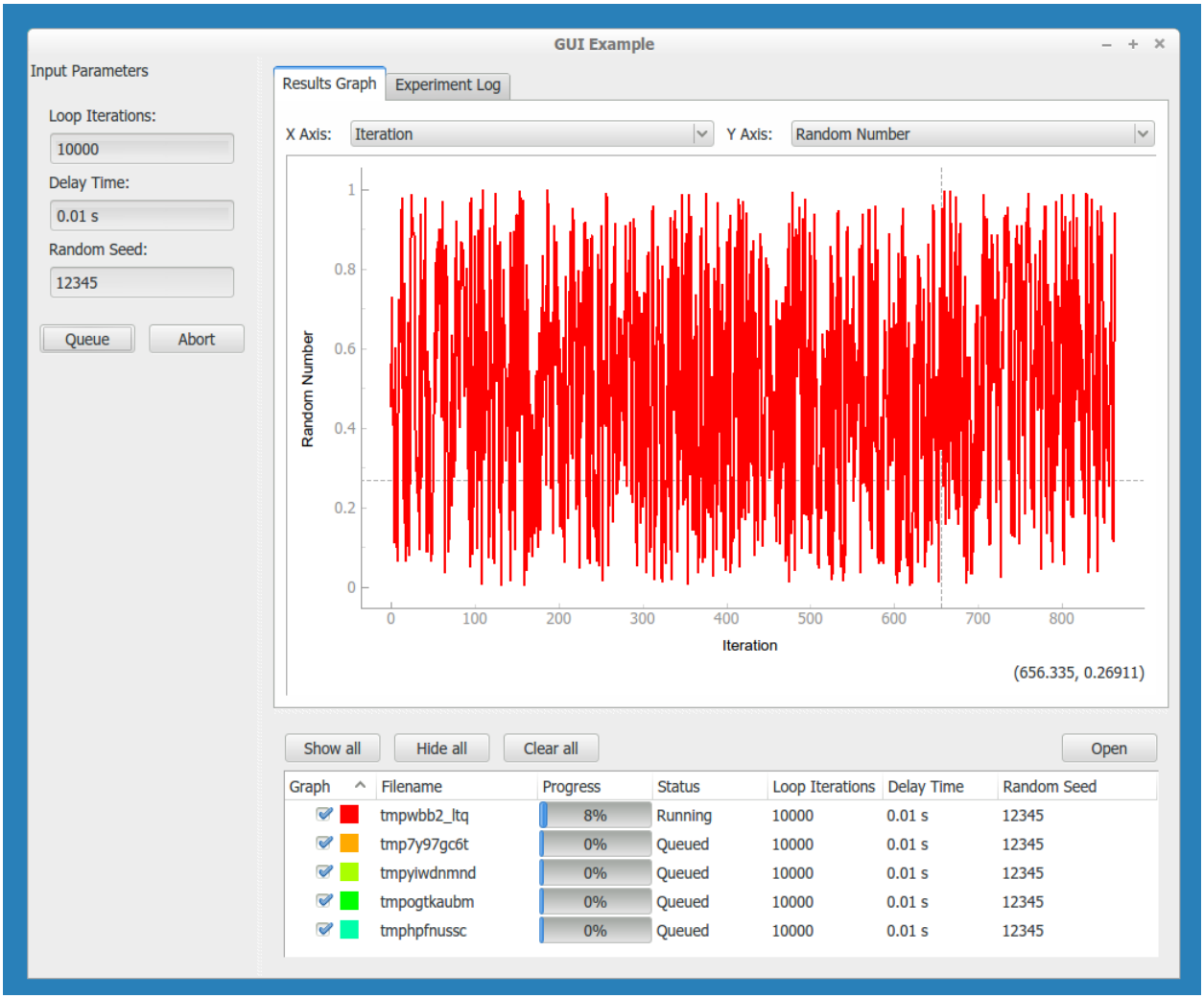
```
from pymeasure.log import console_log
from pymeasure.display.Qt import QtWidgets
from pymeasure.display.windows import ManagedWindow
from pymeasure.experiment import Procedure, Results
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations', default=100)
    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')


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

This results in the following graphical display.

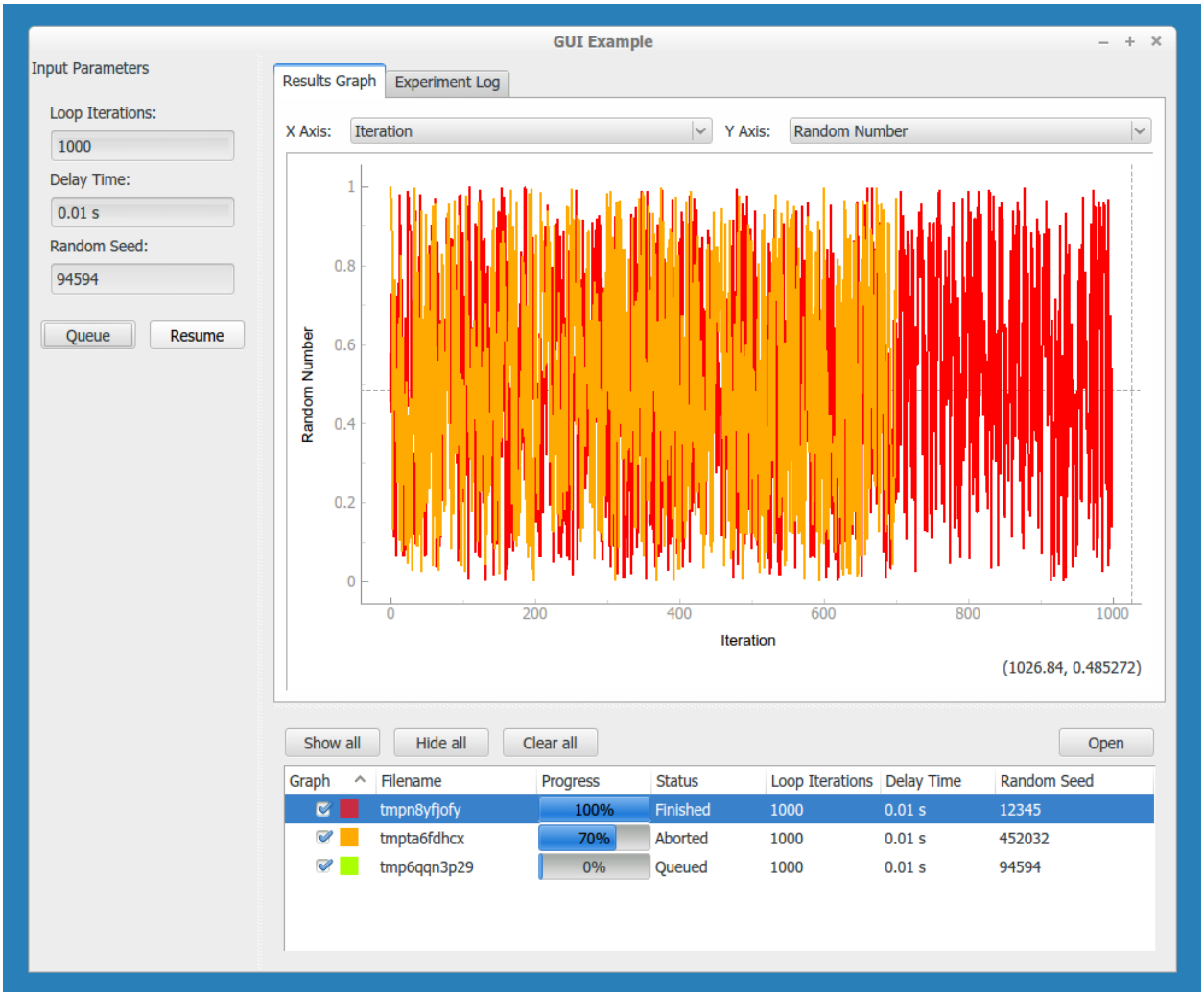


In the code, the `MainWindow` class is a sub-class of the *`ManagedWindow`* class. We override the constructor to provide information about the procedure class and its options. The `inputs` are a list of `Parameters` class-variable names, which the display will generate graphical fields for. When the 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.

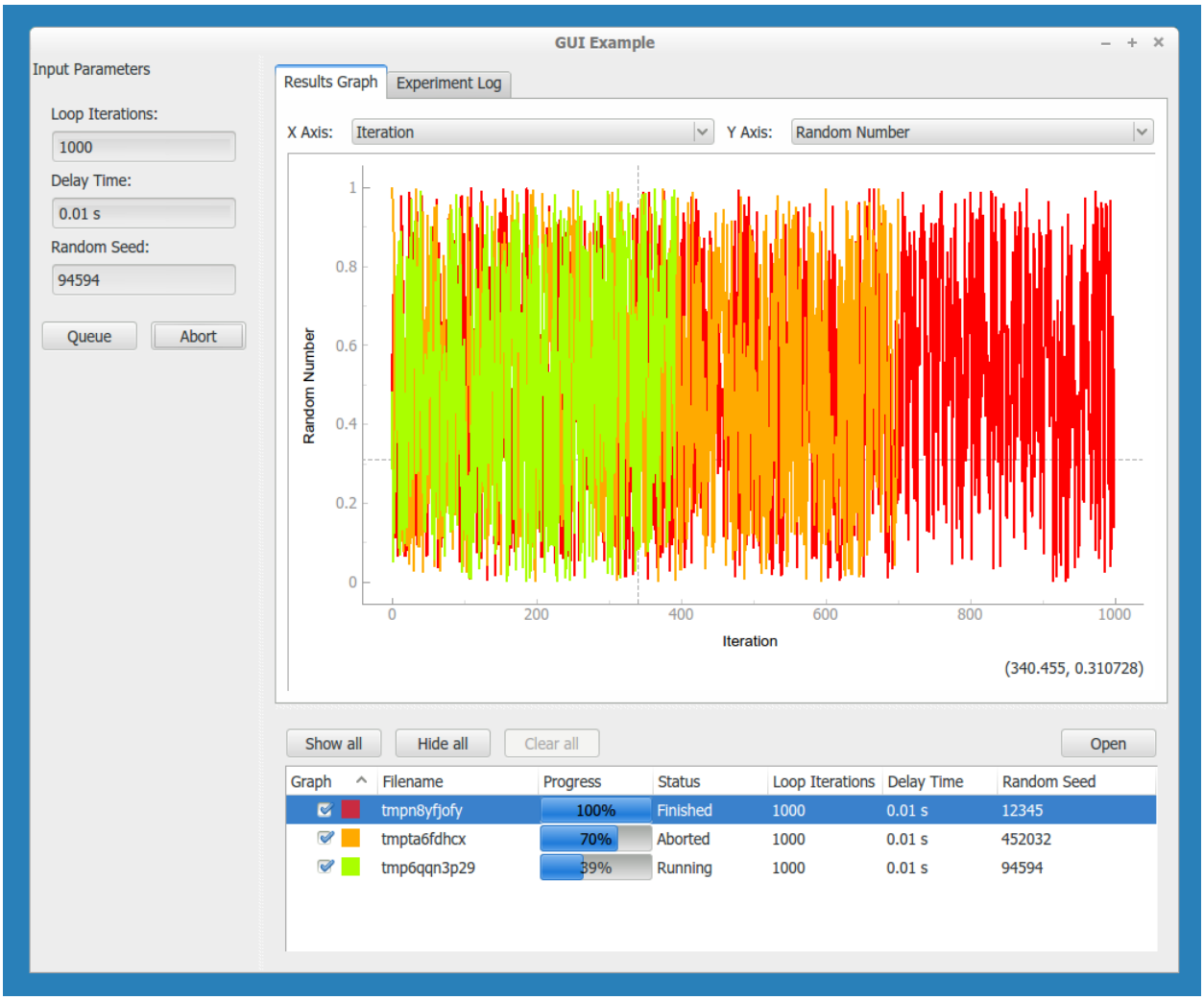
As a bit of background information (for basic usage this needs not be known): the *`queue()`* method establishes how the *`Procedure`* object is constructed. The `make_procedure()` method is used to create a *`Procedure`* based on the graphical input fields. Here we are free to modify the procedure before 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.

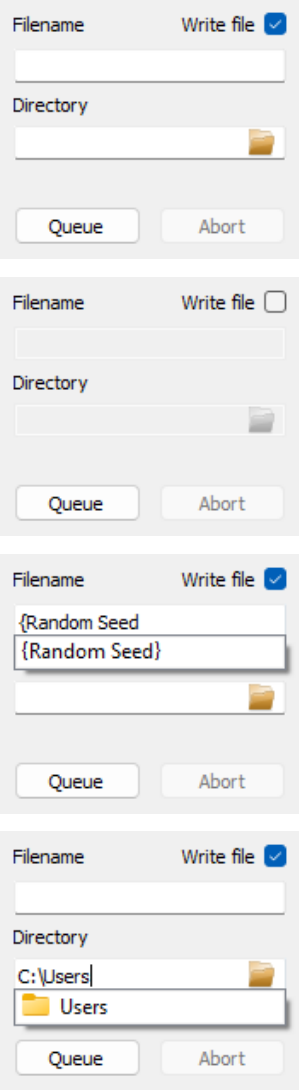
Note

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 preferred 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)
```

The filename and directory input

By default, a ManagedWindow instance contains fields for the filename and the directory (as part of the *FileInputWidget*) to control where the results of an experiment are saved.



If the checkbox named *Save data* is enabled, the measurement is written to a file. Otherwise, it is stored in a temporary file.

The filename in the designated field can be entered with or without extension. If the entered extension is recognized (by default `.csv` and `.txt` are recognized), that extension is used. If the extension is not recognized, the first of the available extensions will be used (default is `.csv`). Additionally, a sequence number is added just before the extension to ensure the uniqueness of the filename.

The filename can also contain placeholders, which are filled in using the standard python `format` function (i.e., placeholders can be entered as `'{placeholder name:formatspec}'`). Valid placeholders are the names of the all the input parameters or metadata of the measurement procedure; the valid placeholders are listed in the tooltip of the input field. As the standard `format` functionality is used, the placeholders can be formatted as such; for example, the filename `'DATA_delay{Delay Time:08.3f}s'` gets formatted into `'DATA_delay0000.010s'`.

Both the filename and the directory field are provided with auto-completion to help with filling in these fields. The directory field contains a button on the right side to open a folder-selection window.

The default values can be easily set after the *ManagedWindow* has been initialized; this allows setting a default location and a default filename, changing the default recognized extensions, control the default toggle-value for the *Save data* option, and control whether the filename input field is frozen.

```
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',
        )
        self.setWindowTitle('GUI Example')

        self.filename = r'default_filename_delay{Delay Time:4f}s'   # Sets default␣
↪filename
        self.directory = r'C:/Path/to/default/directory'            # Sets default␣
↪directory
        self.store_measurement = False                              # Controls the 'Save␣
↪data' toggle
        self.file_input.extensions = ["csv", "txt", "data"]         # Sets recognized␣
↪extensions, first entry is the default extension
        self.file_input.filename_fixed = False                      # Controls whether␣
↪the filename-field is frozen (but still displayed)
```

The presence of the widget is controlled by the boolean argument `enabled_file_input` of the *ManagedWindow* init. Note that when this is set to `False`, the default *queue()* method of the *ManagedWindow* class will no longer work, and a new, custom, method needs to be implemented; a basic implementation is shown in the documentation of the *queue()* method.

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 Using tabular format

In some experiments, data in tabular format may be useful in addition or in alternative to graphical plot. *ManagedWindowBase* allows adding a *TableWidget* to show experiments data, the widget supports also exporting data in some popular format like CSV, HTML, etc. Below an example on how to customize *ManagedWindowBase* to use tabular format, it derived from example above and changed lines are marked.

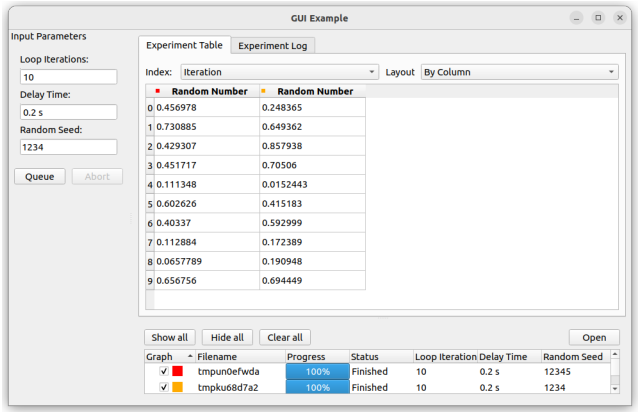
```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import sys
import tempfile
import random
from time import sleep
from pymeasure.log import console_log
from pymeasure.display.Qt import QtWidgets
from pymeasure.display.windows import ManagedWindowBase
from pymeasure.display.widgets import TableWidget, LogWidget
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']

    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(ManagedWindowBase):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        widget_list = (TableWidget("Experiment Table",
                                   RandomProcedure.DATA_COLUMNS,
                                   by_column=True,
                                   ),
                       LogWidget("Experiment Log"),
                       )
        super().__init__(
            procedure_class=RandomProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            widget_list=widget_list,
        )
        logging.getLogger().addHandler(widget_list[1].handler)
        log.setLevel(self.log_level)
        log.info("ManagedWindow connected to logging")
        self.setWindowTitle('GUI Example')

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

This results in the following graphical display.



3.3.5 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`
- *`TableWidget`*

3.3.6 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.

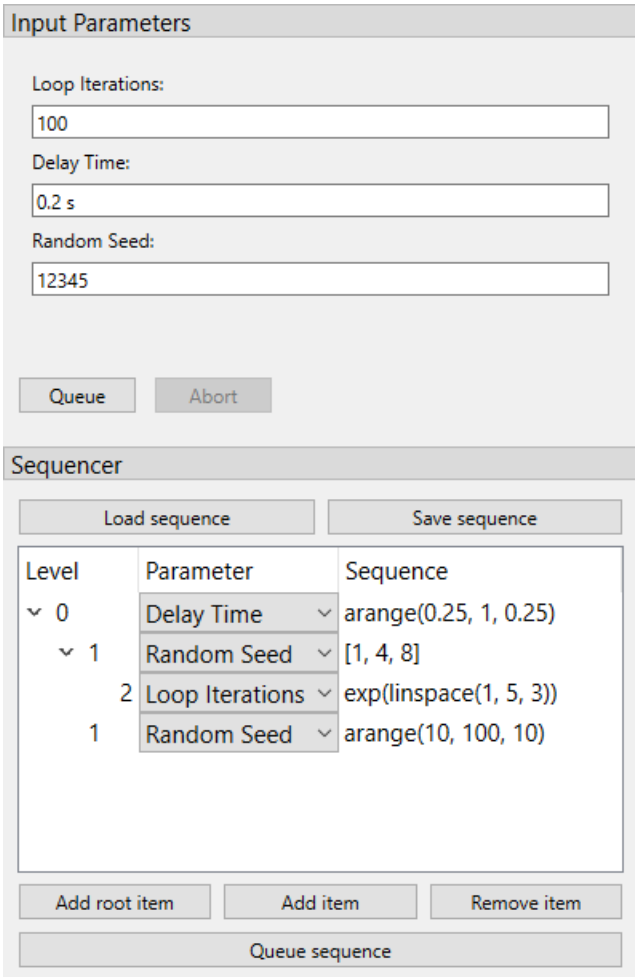
To activate the sequencer, two additional keyword arguments are added to *`ManagedWindow`*, namely `sequencer` and `sequencer_inputs`. `sequencer` accepts a boolean stating whether or not the sequencer has to be included into the window and `sequencer_inputs` accepts either `None` or a list of the parameter 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')
```

This adds the sequencer underneath 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 latter two either add an 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.

3.3.7 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:

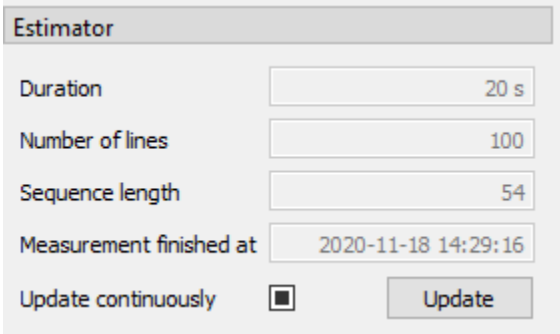
```
class RandomProcedure(Procedure):

    # ...

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

        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))),
```

(continues on next page)

(continued from previous page)

```
        ]

        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 20, and lines 32 and 33, we add two new columns of data to be recorded `'Random Number 2'` and `'Random Number 3'`
3. On line 44 we make `MainWindow` a subclass of `ManagedDockWindow`
4. On line 51 we will pass in a list of strings from `DATA_COLUMNS` to the `x_axis` argument
5. On line 52 we will pass in a list of strings from `DATA_COLUMNS` to the `y_axis` argument

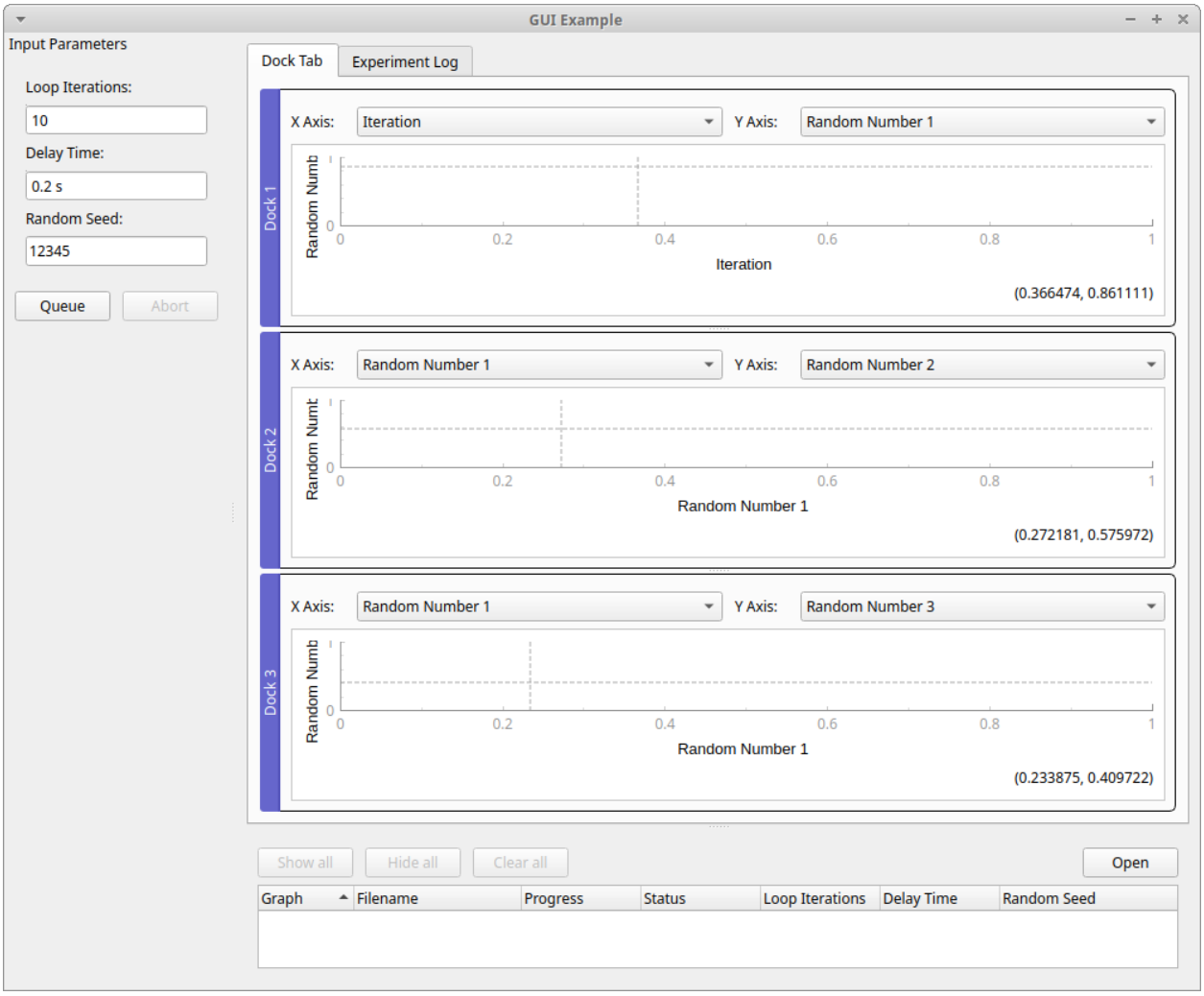
```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import sys
import tempfile
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
```

(continues on next page)

(continued from previous page)

```
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')


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']
```

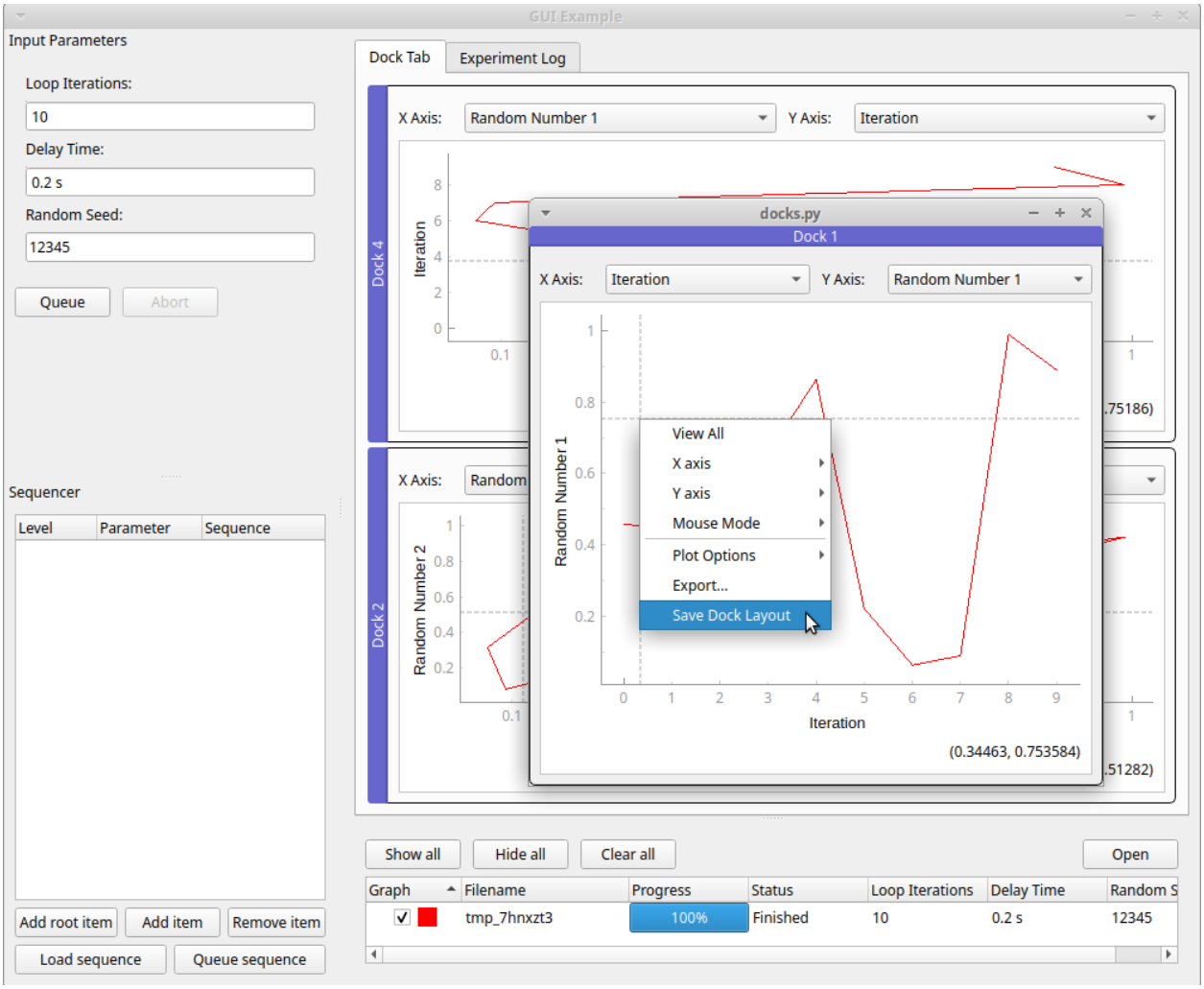
If one list is longer than the last element if the other list is used as the default label for the rest of the dockable plots. In our example that is why we have two **X Axis** labels with "Random Number 1". The longest list between `x_axis` and `y_axis` determines the number of plots. In our example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

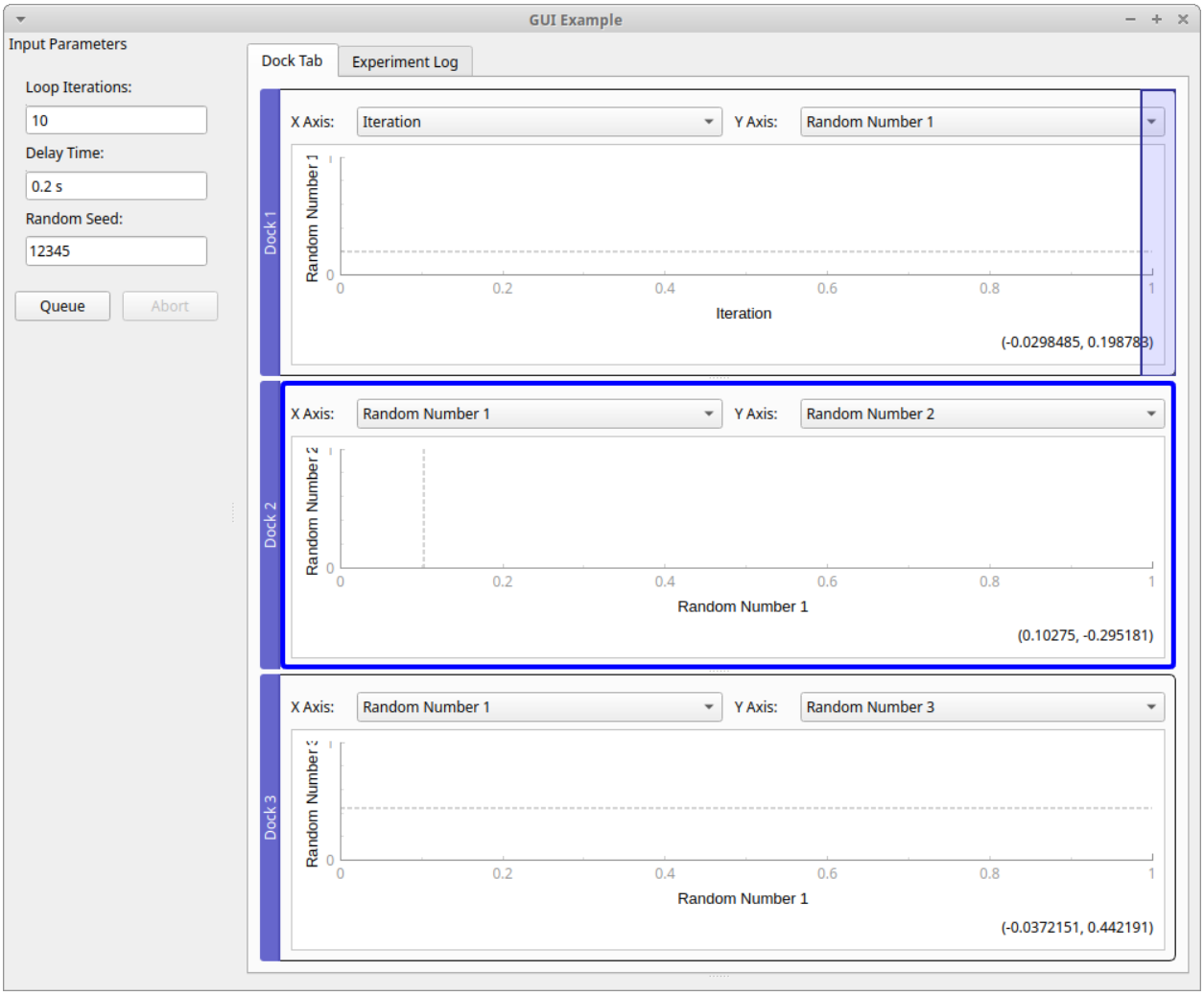
You can return the popped out window to the main window by clicking the close icon X in the top right.

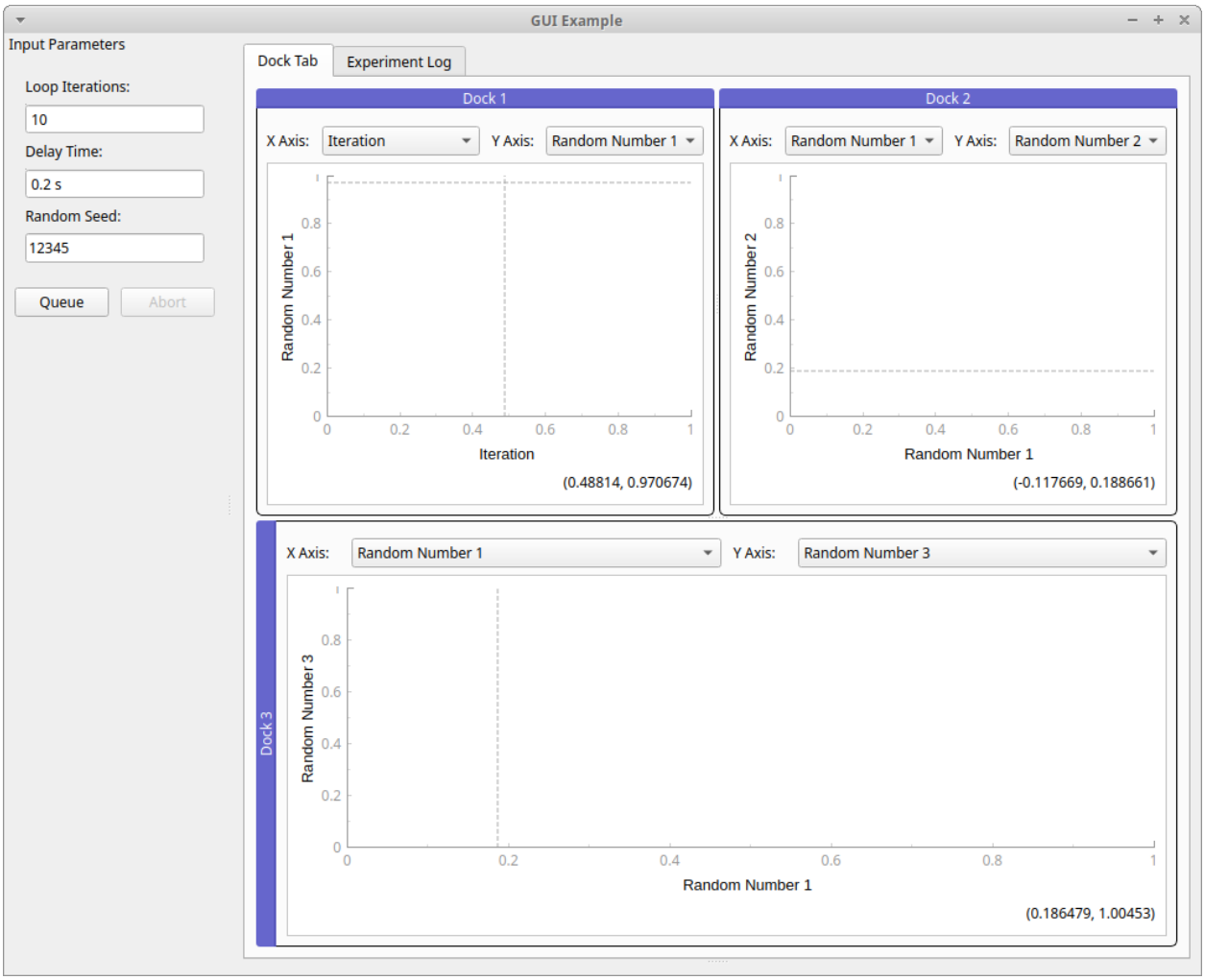
After positioning your dock windows, you can save the layout by right-clicking a dock widget and select "Save Dock Layout" from the context menu. This will save the layout of all docks and the settings for each plot to a file. By default the file path is the current working directory of the python file that started `ManagedDockWindow`, and the default file name is '*procedure class* + "_dock_layout.json"'. For our example, that would be "./RandomProcedure_dock_layout.json"

When you run the python file that invokes `ManagedDockWindow` again, it will look for and load the dock layout file if it exists.

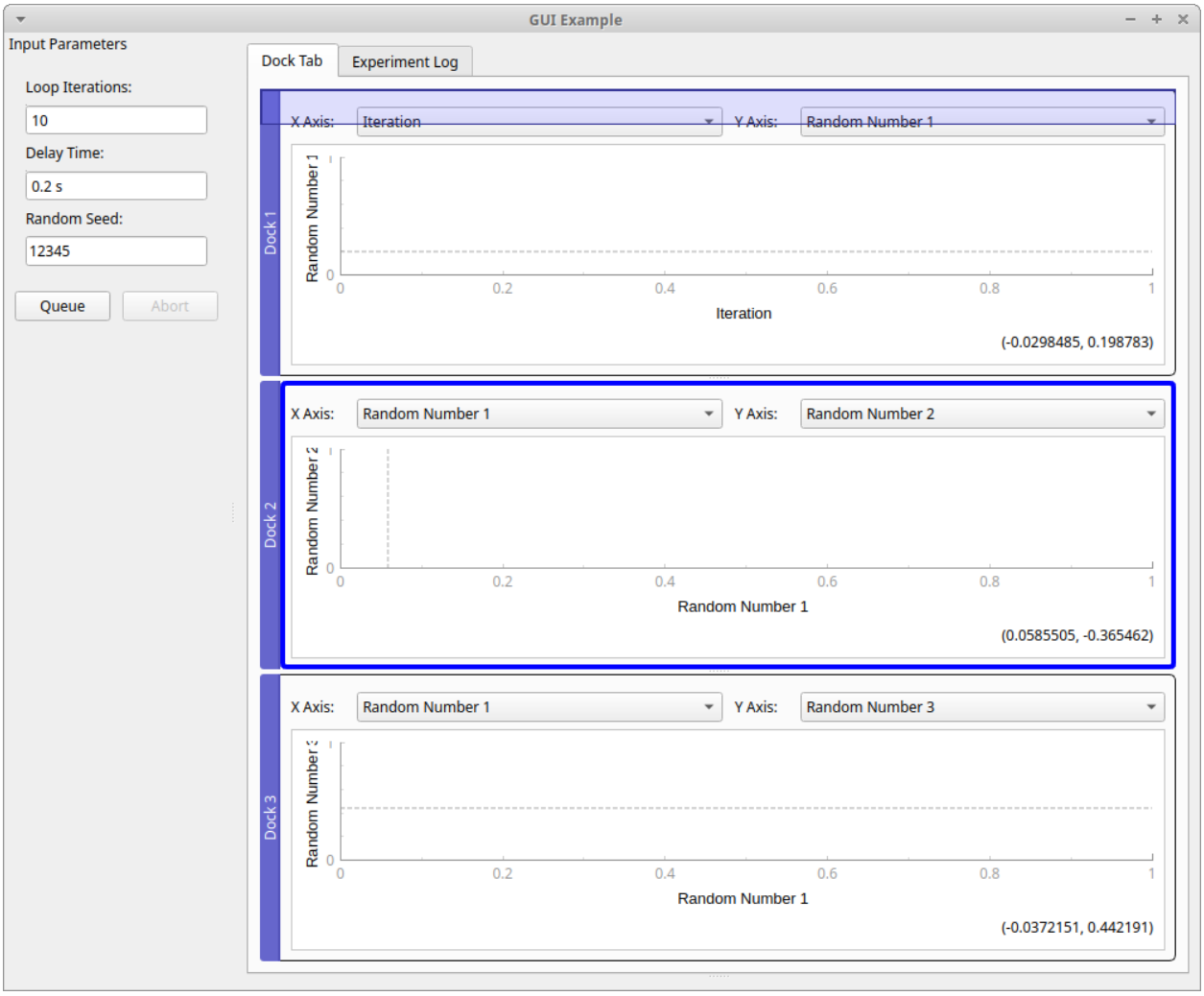


You can drag a dockable plot to reposition it in reference to other plots in the main dock window in several ways. You can drag the blue "Dock #" title bar to the left or right side of another plot to reposition a plot to be side by side with another plot:

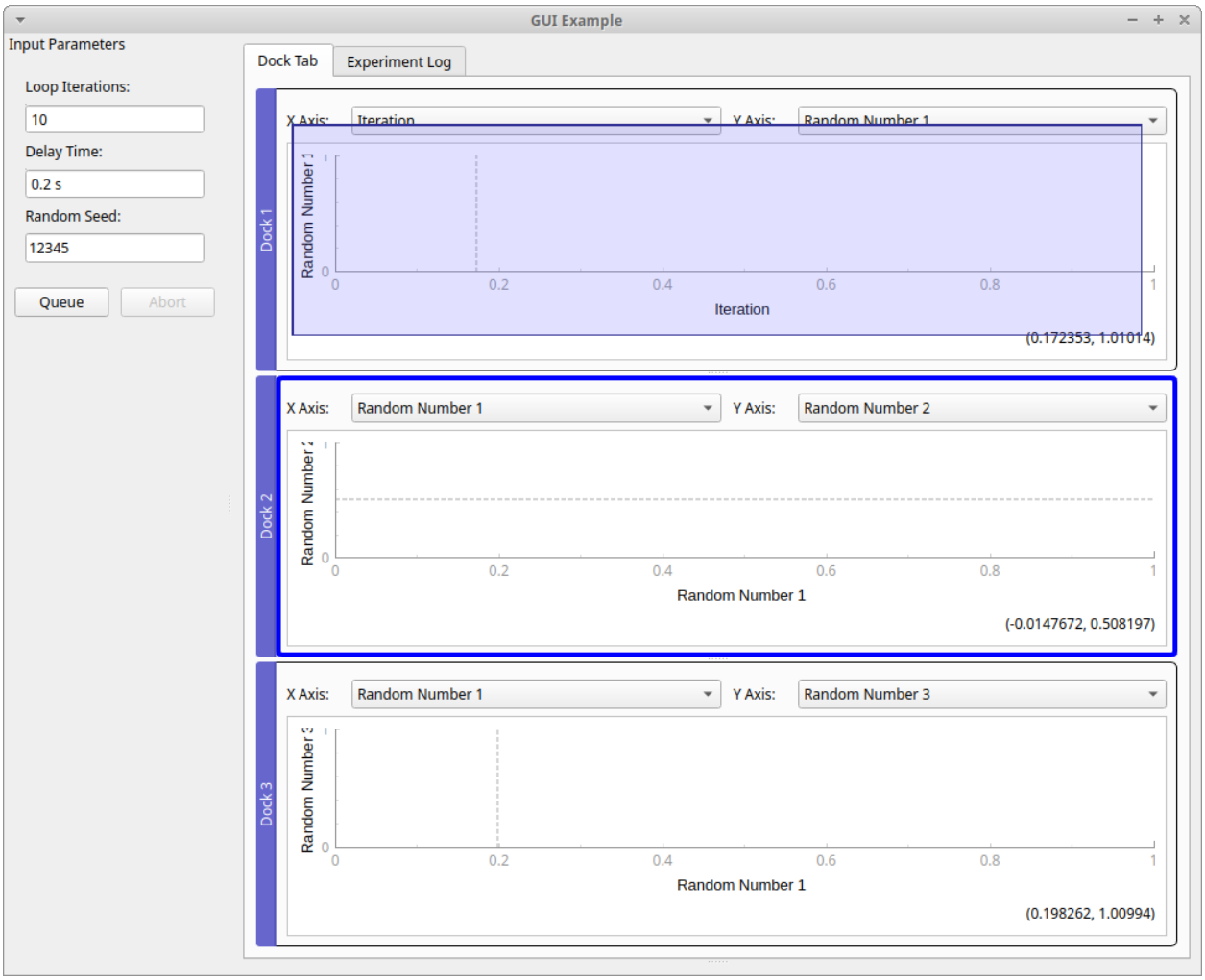

GUI Example
Input Parameters
Loop Iterations:
10
Delay Time:
0.2 s
Random Seed:
12345
Queue
Abort
Dock Tab
Experiment Log
Dock 1
X Axis:
Iteration
Y Axis:
Random Number 1
Random Number 1
Iteration
(-0.0298485, 0.198783)
Dock 2
X Axis:
Random Number 1
Y Axis:
Random Number 2
Random Number 1
(0.10275, -0.295181)
Dock 3
X Axis:
Random Number 1
Y Axis:
Random Number 3
Random Number 1
(-0.0372151, 0.442191)

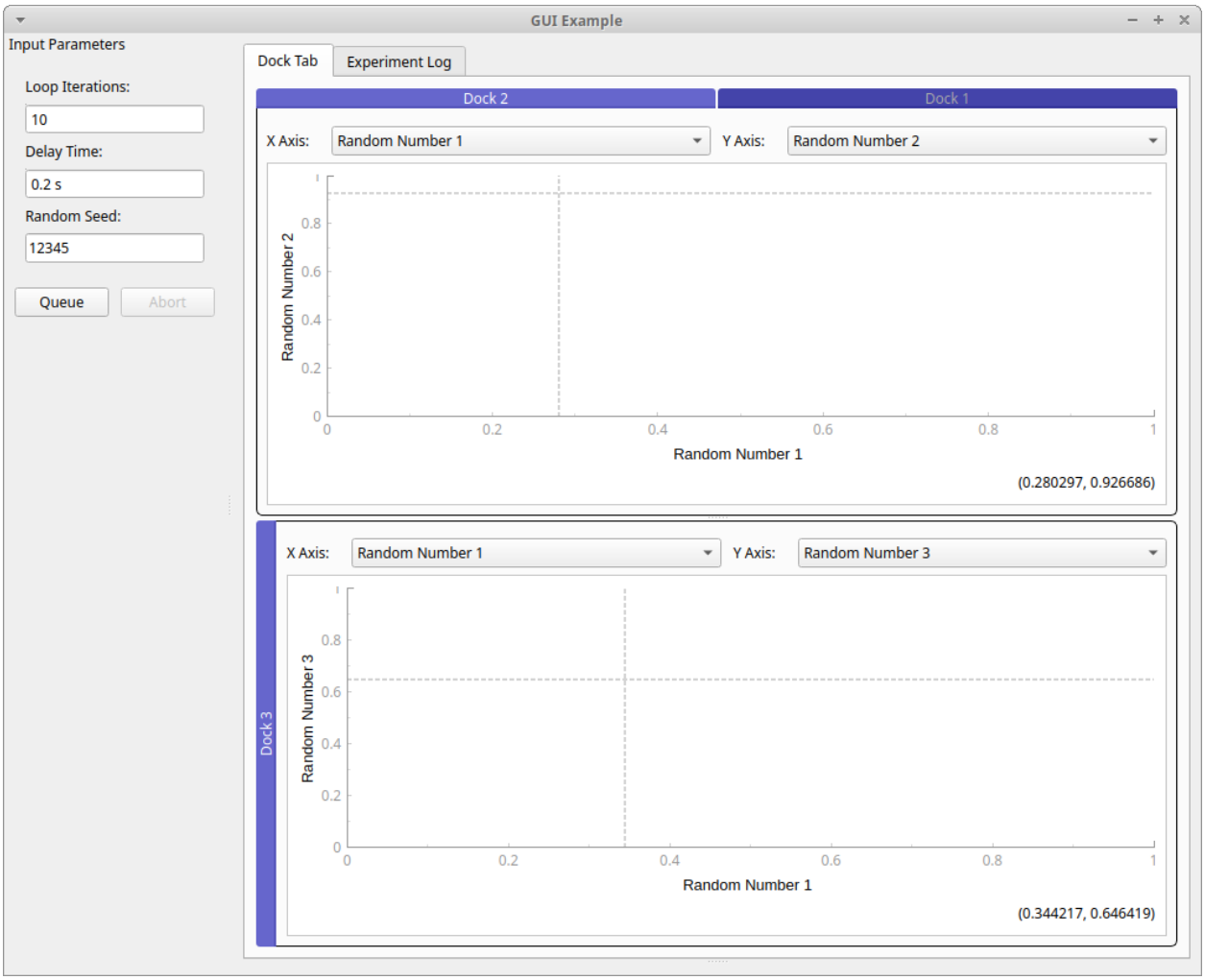


You can also drag the blue "Dock #" title bar to the top or bottom side of another plot to reposition a plot to rearrange the vertical order of the plots:



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)



3.3.10 Using the ManagedConsole

The *ManagedConsole* is the most convenient tool for running measurements with your Procedure using a command line interface. The *ManagedConsole* allows to run an experiment with the same set of parameters available in the *ManagedWindow*, but they are defined using a set of command line switches.

It is also possible to define a test that uses both *ManagedConsole* or *ManagedWindow* according to user selection in the command line.

Enabling console mode is easy and straightforward and the following example demonstrates how to do it.

The following example is a variant of the code example from *Using the ManagedWindow* where some parts have been highlighted:

1. On line 8 we now import *ManagedConsole*
2. On line 73, we add the support for console mode

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

(continues on next page)

(continued from previous page)

```
from pymeasure.experiment import Procedure, IntegerParameter, Parameter, FloatParameter
from pymeasure.experiment import Results
from pymeasure.display.console import ManagedConsole
from pymeasure.display.Qt import QtWidgets
from pymeasure.display.windows import ManagedWindow
import logging

log = logging.getLogger('')
log.addHandler(logging.NullHandler())


class TestProcedure(Procedure):
    iterations = IntegerParameter('Loop Iterations', default=100)
    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 up random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting to generate numbers")
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
            }
            log.debug("Produced numbers: %s" % data)
            self.emit('results', data)
            self.emit('progress', 100 * (i + 1) / self.iterations)
            sleep(self.delay)
            if self.should_stop():
                log.warning("Catch stop command in procedure")
                break

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


class MainWindow(ManagedWindow):

    def __init__(self):
        super(MainWindow, self).__init__(
            procedure_class=TestProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number'
        )
```

(continues on next page)

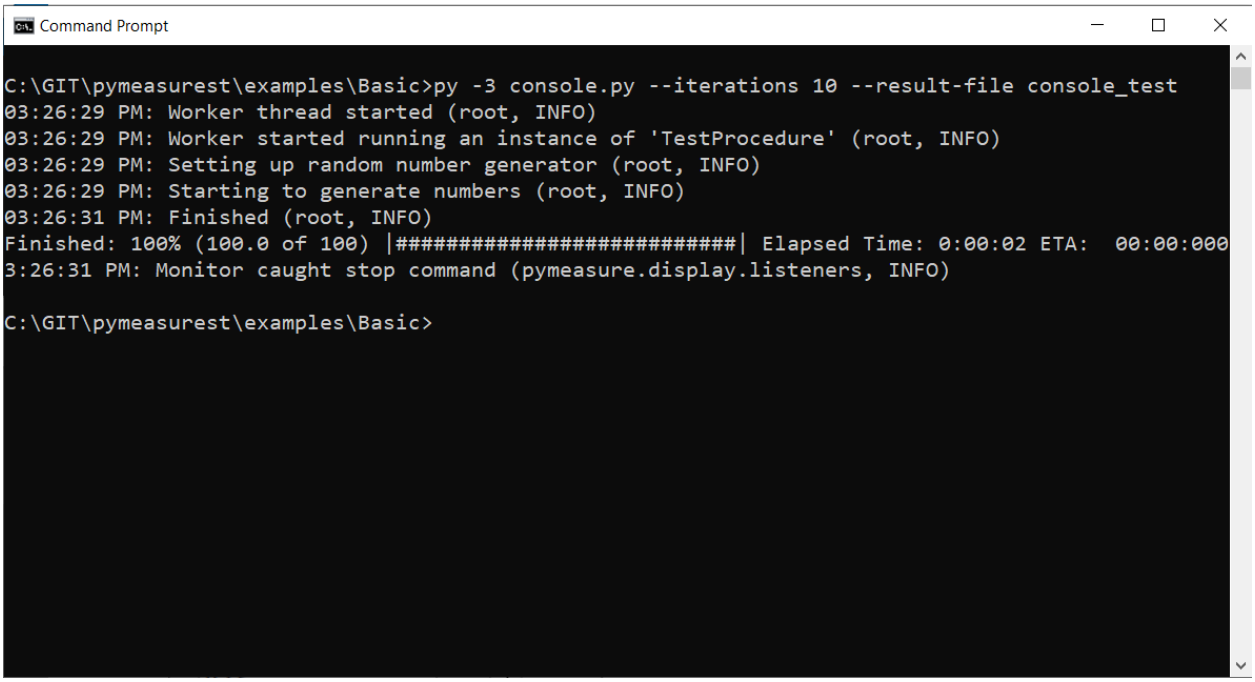
(continued from previous page)

```
        self.setWindowTitle('GUI Example')


if __name__ == "__main__":
    if len(sys.argv) > 1:
        # If any parameter is passed, the console mode is run
        # This criteria can be changed at user discretion
        app = ManagedConsole(procedure_class=TestProcedure)
    else:
        app = QtWidgets.QApplication(sys.argv)
        window = MainWindow()
        window.show()

    sys.exit(app.exec())
```

If we run the script above without any parameter, you will have the graphical user interface example. If you run as follow, you will use the command line mode:

```
python console.py --iterations 10 --result-file console_test
```

Console output is as follow (to show the progress bar, you need to install the optional module progressbar2):



Other useful commands

To show all the command line switches:

```
python console.py --help
```

To run an experiment with parameters retrieved from an existing result file.

```
python console.py --use-result-file console_test2023-08-09_1.csv
```

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 communication 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**(*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

- **log** – Parent logger of the 'Adapter' logger.
- ****kwargs** – Keyword arguments just to be cooperative.

close()

Close the connection.

flush_read_buffer()

Flush and discard the input buffer. Implement in subclass.

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'>*, *sep=''*, ***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.
- **sep** (*string*) – Separator between chars. If given, use fromstring, otherwise frombytes.
- ****kwargs** – Further arguments for the NumPy fromstring / frombytes method.

Returns
NumPy array of values

Raises
ValueError – if the data buffer is empty or malformed

read_bytes(*count=-1*, *break_on_termchar=False*, ***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 from the whole read buffer.
- **break_on_termchar** (*bool*) – Stop reading at a termination character.
- ****kwargs** – Keyword arguments for the connection itself.

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

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=''*, *log=None*, ***kwargs*)

Bases: *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.
- **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*.

close()

Close the connection.

> **Note**
>
> This closes the connection to the resource for all adapters using it currently (e.g. different adapters using the same GPIB line).

flush_read_buffer()

Flush and discard the input buffer

As detailed by pyvisa, discard the read and receivee 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'>*, *sep=''*, ***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.
- **sep** (*string*) – Separator between chars. If given, use fromstring, otherwise frombytes.
- ****kwargs** – Further arguments for the NumPy fromstring / frombytes method.

Returns
NumPy array of values

Raises
ValueError – if the data buffer is empty or malformed

read_bytes(*count=-1*, *break_on_termchar=False*, ***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 from the whole read buffer.
- **break_on_termchar** (*bool*) – Stop reading at a termination character.
- ****kwargs** – Keyword arguments for the connection itself.

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

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*, *write_termination=''*, *read_termination=''*, ***kwargs*)

Bases: *Adapter*

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

Parameters

- **port** – Serial port
- **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 endianness.
- **header_fmt** – Format of the header prefixing the data ("ieee", "hp", "empty").

Returns
binary string.

Return type
bytes

`close()`

Close the connection.

`flush_read_buffer()`

Flush and discard the input buffer.

`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'>, sep='', **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.
- **`sep`** (*string*) – Separator between chars. If given, use fromstring, otherwise frombytes.
- **`**kwargs`** – Further arguments for the NumPy fromstring / frombytes method.

Returns
NumPy array of values

Raises
`ValueError` – if the data buffer is empty or malformed

`read_bytes(count=-1, break_on_termchar=False, **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 from the whole read buffer.
- **`break_on_termchar`** (*bool*) – Stop reading at a termination character.
- **`**kwargs`** – Keyword arguments for the connection itself.

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

`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*, *auto=False*, *eoi=True*, *eos='\n'*, *gpib_read_timeout=None*, ***kwargs*)

Bases: *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.
- **auto** – Enable or disable read-after-write and address instrument to listen.
- **eoi** – Enable or disable EOI assertion.
- **eos** – Set command termination string (CR+LF, CR, LF, or "")
- **gpib_read_timeout** – Set read timeout for GPIB communication in milliseconds from 1..3000
- **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
```

Since the Prologix adapter uses the same communication channel (an USB CDC) for both, communication with the adapter itself, as well as communication over GPIB, certain things need to be kept in mind:

- Operations that need to read from GPIB use the standard *read()* method.
- Operations that just read responses from the Prologix itself need to add the parameter `prologix=True` to *read()*; this avoids requesting data from GPIB. This is also necessary when the adapter is put into "listen-only" mode, where all GPIB traffic is automatically being passed up.
- Binary data must be passed to the bus using *write_binary_values()*. This takes care of properly escaping those binary values that would otherwise be interpreted by the Prologix adapter. Note that the default for *write_binary_values()* are to assume floating-point binary data, and prepend IEEE headers. In order to pass just plain bytes to the adapter, tune the `datatype` and `header_fmt` parameters:

  ```
  multimeter.write_binary_values('W', [addr], datatype='B', header_fmt='empty')
  ```

_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 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

property auto

Control whether to address instruments to talk after sending them a command (bool).

Configure Prologix GPIB controller to automatically address instruments to talk after sending them a command in order to read their response. The feature called, Read-After-Write, saves the user from having to issue read commands repeatedly. This property enables (True) or disables (False) this feature.

`close()`

Close the connection.

> **Note**
>
> This closes the connection to the resource for all adapters using it currently (e.g. different adapters using the same GPIB line).

`property eoi`

Control whether to assert the EOI signal with the last character of any command sent over GPIB port (bool).

Some instruments require EOI signal to be asserted in order to properly detect the end of a command.

`property eos`

Control GPIB termination characters (str).

possible values:

- CR+LF
- CR
- LF
- empty string

When data from host is received, all non-escaped LF, CR and ESC characters are removed and GPIB terminators, as specified by this command, are appended before sending the data to instruments. This command does not affect data from instruments received over GPIB port.

`flush_read_buffer()`

Flush and discard the input buffer

As detailed by pyvisa, discard the read and receivee 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

`property gpib_read_timeout`

Control the timeout value for the GPIB communication in milliseconds

possible values: 1 - 3000

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'>*, *sep=''*, ***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.
- **sep** (*string*) – Separator between chars. If given, use fromstring, otherwise frombytes.
- ****kwargs** – Further arguments for the NumPy fromstring / frombytes method.

Returns
NumPy array of values

Raises
ValueError – if the data buffer is empty or malformed

read_bytes(*count=-1*, *break_on_termchar=False*, ***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 from the whole read buffer.
- **break_on_termchar** (*bool*) – Stop reading at a termination character.
- ****kwargs** – Keyword arguments for the connection itself.

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

reset()

Perform a power-on reset of the controller.

The process takes about 5 seconds. All input received during this time is ignored and the connection is closed.

property version

Get the version string of the Prologix controller.

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 Test adapters

These pieces are useful when writing tests.

pymeasure.test.**expected_protocol**(*instrument_cls: type[Inst]*, *comm_pairs: Sequence[tuple[bytes | bytearray | str | list[int] | tuple[int, ...] | int | float | None, bytes | bytearray | str | list[int] | tuple[int, ...] | int | float | None]]*, *connection_attributes: dict[str, Any] | None = None*, *connection_methods: dict[str, Any] | None = None*, ***kwargs*) → Generator[Inst, Any, None]

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: Sequence[tuple[bytes | bytearray | str | list[int] | tuple[int, ...] | int | float | None, bytes | bytearray | str | list[int] | tuple[int, ...] | int | float | None]] | None = None*, *connection_attributes=None*, *connection_methods=None*, ***kwargs*)

Bases: *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.

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).

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

Bases: *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() == ""
```

(continues on next page)

(continued from previous page)

```
assert a.ask("10") == "10"
assert a.values("10") == [10]
```

close()

Close the connection.

flush_read_buffer()

Flush and discard the input buffer. Implement in subclass.

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'>*, *sep=''*, ***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.
- **sep** (*string*) – Separator between chars. If given, use fromstring, otherwise frombytes.
- ****kwargs** – Further arguments for the NumPy fromstring / frombytes method.

Returns
NumPy array of values

Raises
ValueError – if the data buffer is empty or malformed

read_bytes(*count=-1*, *break_on_termchar=False*, ***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 from the whole read buffer.
- **break_on_termchar** (*bool*) – Stop reading at a termination character.
- ****kwargs** – Keyword arguments for the connection itself.

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

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.

class pymeasure.generator.**Generator**

Generates tests from the communication with an instrument.

Example usage:

```
g = Generator()
inst = g.instantiate(TC038, "COM5", 'hcp', adapter_kwargs={'baud_rate': 9600})
inst.information  # returns the 'information' property and adds it to the tests
inst.setpoint = 20
inst.setpoint == 20  # should be True
g.write_file("test_tc038.py")  # write the tests to a file
```

instantiate(*instrument_class*, *adapter*, *manufacturer*, *adapter_kwargs=None*, ***kwargs*)

Instantiate the instrument and store the instantiation communication.

..note:

```
You have to give all keyword arguments necessary for adapter instantiation in
`adapter_kwargs`, even those, which are defined somewhere in the instrument's
``__init__`` method, be it as a default value, be it directly in the
``Instrument.__init__()`` call.
```

Parameters

- **instrument_class** – Class of the instrument to test.
- **adapter** – Adapter (instance or str) for the instrument instantiation.
- **manufacturer** – Module from which to import the instrument, e.g. 'hcp' if instrument_class is 'pymeasure.hcp.tc038'.

- **adapter_kwargs** – Keyword arguments for the adapter instantiation (see note above).
- ****kwargs** – Keyword arguments for the instrument instantiation.

Returns
A man-in-the-middle instrument, which can be used like a normal instrument.

parse_stream()

Parse the stream not yet read.

test_method(*method_name*, **args*, ***kwargs*)

Test calling the *method_name* of the instruments with *args* and *kwargs*.

test_property_getter(*property*)

Test getting the *property* of the instrument, adding it to the list.

test_property_setter(*property*, *value*)

Test setting the *property* of the instrument to *value*, adding it to the list.

test_property_setter_batch(*property*, *values*)

Test setting *property* to each element in *values*.

write_file(*filename='tests.py'*)

Write the tests into the file.

Parameters
filename – Name to save the tests to, may contain the path, e.g. "/tests/test_abc.py".

write_getter_test(*file*, *property*, *parameters*)

Write a getter test.

write_init_test(*file*)

Write the header and init test.

write_method_test(*file*, *method*, *parameters*)

Write a test for a method.

write_method_tests(*file*)

Write all parametrized method tests in alphabetic order.

write_property_tests(*file*)

Write tests for properties in alphabetic order.

If getter and setter exist, the setter is the first test.

write_setter_test(*file*, *property*, *parameters*)

Write a setter test.

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: 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: QueueListener

class pymeasure.experiment.listeners.**Recorder**(*results*, *queue*, ***kwargs*)

Bases: 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.

is_last() → bool

Check if the procedure is the last one in the queue.

This method must be monkey patched by a worker to provide functionality.

Returns:

bool: True if the procedure is the last one in the queue, False otherwise.

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

static parse_columns(*columns*)

Get columns with any units in parentheses. For each column, if there are matching parentheses containing text with no spaces, parse the value between the parentheses as a Pint unit. For example, "Source Voltage (V)" will be parsed and matched to `Unit('volt')`. Raises an error if a parsed value is undefined in Pint unit registry. Return a dictionary of matched columns with their units.

Parameters
columns – List of columns to be parsed.

Returns
Dictionary of columns with Pint units.

classmethod placeholder_names()

Collect the names of all eligible placeholders (parameters & metadata)

placeholder_objects()

Collect all eligible placeholders (parameters & metadata) with their value in a dict.

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*, *description=None*)

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

convert(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

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

convert(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

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

convert(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

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.

convert(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

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*, *description=None*)

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 containing 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.

Description
A string providing a human-friendly description for the parameter.

`property cli_args`

helper for command line interface parsing of parameters

This property returns a list of data to help formatting a command line interface interpreter, the list is composed of the following elements: - index 0: default value - index 1: List of value to format an help string, that is either, the name of the fields to be documented or a tuple with (helps_string, field) - index 2: type

`convert`(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

`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

`convert`(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

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

convert(*value*)

Convert user input to python data format

Subclasses are expected to customize this method. Default implementation is the identity function

Parameters
value – value to be converted

Returns
converted value

5.5 Worker class

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

Bases: 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: str*, *record: Any*)

Emits data of some topic over TCP

handle_abort()

handle_batch_record(*record: Any*)

handle_error()

handle_record(*record: dict[str, Any]*)

is_last()

join(*timeout: int = 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: int*)

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.**BaseBrowserItem**

Bases: object

Base class for an experiment's browser item. BaseBrowerItem outlines core functionality for displaying progress of an experiment to the user.

class pymeasure.display.browser.**Browser**(*procedure_class*, *display_parameters*, *measured_quantities*, *sort_by_filename=False*, *parent=None*)

Bases: 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**(*results*, *color*, *parent=None*)

Bases: QTreeWidgetItem, *BaseBrowserItem*

Represent a row in the *Browser* tree widget

6.2 Console class

class pymeasure.display.console.**ConsoleArgumentParser**(*procedure_class*, ***kwargs*)

Bases: ArgumentParser

setup_parser()

Setup command line arguments parsing from parameters information

class pymeasure.display.console.**ConsoleBrowserItem**(*progress_bar*)

Bases: *BaseBrowserItem*

class pymeasure.display.console.**ManagedConsole**(*procedure_class*, *log_channel=''*, *log_level=20*, ***kwargs*)

Bases: QCoreApplication

Base class for console experiment management.

Parameters for __init__ constructor.

Parameters

- **procedure_class** – procedure class describing the experiment (see *Procedure*)
- **log_channel** – logging.Logger instance to use for logging output
- **log_level** – logging level
- **kwargs** – additional keyword arguments to be passed to the *ConsoleArgumentParser* constructor

abort()

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

exec() → int

get_filename(*directory*, *procedure=None*)

Return filename for saving results file

Parameters

directory – directory of the returned filename.

6.3 Curves classes

class pymeasure.display.curves.**BufferCurve**(***kwargs*)

Bases: 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**(*plot*, *pen=None*)

Bases: 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**(*results*, *x*, *y*, *force_reload=False*, *wdg=None*, ***kwargs*)

Bases: 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**(*results*, *x*, *y*, *z*, *force_reload=False*, *wdg=None*, ***kwargs*)

Bases: 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

6.4 Inputs classes

class pymeasure.display.inputs.**BooleanInput**(*parameter*, *parent=None*, ***kwargs*)

Bases: *Input*, 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**(*parameter*, *parent=None*, ***kwargs*)

Bases: *Input*, 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.

stepEnabled(*self*) → QAbstractSpinBox.StepEnabled

class pymeasure.display.inputs.**ListInput**(*parameter*, *parent=None*, ***kwargs*)

Bases: *Input*, 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**(*parameter*, *parent=None*, ***kwargs*)

Bases: *Input*, 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.

stepEnabled(*self*) → QAbstractSpinBox.StepEnabled

textFromValue(*self*, *v: float*) → str

validate(*self*, *input: str | None*, *pos: int*)

valueFromText(*self*, *text: str | None*) → float

class pymeasure.display.inputs.**StringInput**(*parameter*, *parent=None*, ***kwargs*)

Bases: *Input*, 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.

6.5 Listeners classes

class pymeasure.display.listeners.**Monitor**(*queue*)

Bases: QThread

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

run(*self*)

class pymeasure.display.listeners.**QListener**(*port*, *topic=''*, *timeout=0.01*)

Bases: *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

6.6 Log classes

class pymeasure.display.log.**LogHandler**

Bases: Handler

class Emitter

Bases: 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.

6.7 Manager classes

class pymeasure.display.manager.**BaseManager**(*port=5888*, *log_level=20*, *parent=None*)

Bases: QObject

Controls the execution of *Experiment* classes by implementing a queue system in which Experiments are added, removed, executed, or aborted.

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.

class pymeasure.display.manager.**Experiment**(*results*, *curve_list=None*, *browser_item=None*, *parent=None*)

Bases: 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. Not required for *.ManagedConsole* displayed experiments.
- **browser_item** – *BaseBrowserItem* based object

class pymeasure.display.manager.**ExperimentQueue**

Bases: 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**(*widget_list*, *browser*, *port=5888*, *log_level=20*, *parent=None*)

Bases: *BaseManager*

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.

load(*experiment*)

Load a previously executed Experiment

remove(*experiment*)

Removes an Experiment

6.8 Plotter class

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

Bases: 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.

6.9 Qt classes

All Qt imports should reference pymeasure.display.Qt, for consistent 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 accessibility.

6.10 Thread classes

class pymeasure.display.thread.**StoppableQThread**(*parent=None*)

Bases: 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

6.11 Widget classes

class pymeasure.display.widgets.browser_widget.**BrowserWidget**(**args*, *parent=None*)

Bases: QWidget

Widget wrapper for *Browser* class

class pymeasure.display.widgets.directory_widget.**DirectoryLineEdit**(*parent=None*)

Bases: 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**(*get_estimates_callable*)

Bases: *StoppableQThread*

run(*self*)

class pymeasure.display.widgets.estimator_widget.**EstimatorWidget**(*parent=None*)

Bases: 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.fileinput_widget.**FileInputWidget**(*parent=None*)

Bases: QWidget

Widget for controlling where the data of an experiment will be stored.

The widget consists of a field for the filename (*FilenameLineEdit*), a field for the directory (*DirectoryLineEdit*), and a checkbox to control whether the measurement is stored.

property directory

String controlling the directory where the file will be stored.

property extensions

List of extensions that are recognized by the widget.

The first value of this list will be used as default value in case no extension is provided in the filename input field.

property filename

String controlling the filename that is shown in the filename input field.

property filename_base

String containing the base of the filename with which the file will be stored. Can only be read.

property filename_extension

String containing the file extension with which the file will be stored.

Can only be read.

property filename_fixed

Boolean controlling whether the filename input field is frozen. If *True*, the filename field will be visible but disabled (i.e., grayed out).

property store_measurement

Boolean controlling whether the measurement will be stored.

class pymeasure.display.widgets.filename_widget.**FilenameLineEdit**(*procedure_class*, *parent=None*)

Bases: QLineEdit

Widget that allows to choose a filename. A completer is implemented for quick completion of placeholders

class pymeasure.display.widgets.filename_widget.**FilenameValidator**(*placeholders*, *parent*)

Bases: QValidator

fixup(*self*, *a0: str | None*) → str

validate(*self*, *a0: str | None*, *a1: int*)

class pymeasure.display.widgets.filename_widget.**PlaceholderCompleter**(*placeholders*)

Bases: QCompleter

splitPath(*self*, *path: str | None*) → list[str]

class pymeasure.display.widgets.image_frame.**ImageFrame**(*x_axis*, *y_axis*, *z_axis=None*, *refresh_time=0.2*, *check_status=True*, *parent=None*)

Bases: *PlotFrame*

Extends *PlotFrame* to plot also axis Z using colors

ResultsClass

alias of *ResultsImage*

class pymeasure.display.widgets.image_widget.**ImageWidget**(*name*, *columns*, *x_axis*, *y_axis*, *z_axis=None*, *refresh_time=0.2*, *check_status=True*, *parent=None*)

Bases: *TabWidget*, 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=<PyQt5.QtGui.QColor object>*, ***kwargs*)

Creates a new image

remove(*curve*)

Remove curve from widget

sizeHint(*self*) → QSize

class pymeasure.display.widgets.inputs_widget.**InputsWidget**(*procedure_class*, *inputs=()*, *parent=None*, *hide_groups=True*, *inputs_in_scrollarea=False*)

Bases: QWidget

Widget wrapper for various *Inputs classes*

get_procedure()

Returns the current procedure

class pymeasure.display.widgets.log_widget.**HTMLFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*, ***, *defaults=None*)

Bases: Formatter

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

class pymeasure.display.widgets.log_widget.**LogWidget**(*name*, *parent=None*, *fmt=None*, *datefmt=None*)

Bases: *TabWidget*, 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**(*x_axis=None*, *y_axis=None*, *refresh_time=0.2*, *check_status=True*, *parent=None*)

Bases: 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 *ResultsCurve*

parse_axis(*axis*)

Returns the units of an axis by searching the string

class pymeasure.display.widgets.plot_widget.**PlotWidget**(*name*, *columns*, *x_axis=None*, *y_axis=None*, *refresh_time=0.2*, *check_status=True*, *linewidth=1*, *parent=None*)

Bases: *TabWidget*, QWidget

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

clear_widget()

Clear widget content

Behaviour is widget specific and it is currently used in preview mode

load(*curve*)

Add curve to widget

new_curve(*results*, *color=<PyQt5.QtGui.QColor object>*, ***kwargs*)

Create a new curve

preview_widget(*parent=None*)

Return a widget suitable for preview during loading

remove(*curve*)

Remove curve from widget

set_color(*curve*, *color*)

Change the color of the pen of the curve

sizeHint(*self*) → QSize

class pymeasure.display.widgets.results_dialog.**ResultsDialog**(*procedure_class*, *widget_list=()*, *parent=None*)

Bases: 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**(*owner*, *choices*)

Bases: QStyledItemDelegate

createEditor(*self*, *parent: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*) → QWidget | None

setEditorData(*self*, *editor: QWidget | None*, *index: QModelIndex*)

setModelData(*self*, *editor: QWidget | None*, *model: QAbstractItemModel | None*, *index: QModelIndex*)

updateEditorGeometry(*self*, *editor: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*)

class pymeasure.display.widgets.sequencer_widget.**ExpressionValidator**

Bases: QValidator

validate(*self*, *a0: str | None*, *a1: int*)

class pymeasure.display.widgets.sequencer_widget.**LineEditDelegate**

Bases: QStyledItemDelegate

createEditor(*self*, *parent: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*) → QWidget | None

setEditorData(*self*, *editor: QWidget | None*, *index: QModelIndex*)

setModelData(*self*, *editor: QWidget | None*, *model: QAbstractItemModel | None*, *index: QModelIndex*)

updateEditorGeometry(*self*, *editor: QWidget | None*, *option: QStyleOptionViewItem*, *index: QModelIndex*)

class pymeasure.display.widgets.sequencer_widget.**SequenceDialog**(*save=False*, *parent=None*)

Bases: 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**(*data*, *header=('Level', 'Parameter', 'Sequence')*, *parent=None*)

Bases: 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

setData(*self*, *index: QModelIndex*, *value: Any*, *role: int = Qt.ItemDataRole.EditRole*) → bool

visit_tree(*parent*)

Return a generator to enumerate all the nodes in the tree

class pymeasure.display.widgets.sequencer_widget.**SequencerTreeView**(*parent=None*)

Bases: QTreeView

setModel(*self*, *model: QAbstractItemModel | None*)

class pymeasure.display.widgets.sequencer_widget.**SequencerWidget**(*inputs=None*, *sequence_file=None*, *parent=None*)

Bases: 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

clear_widget()

Clear widget content

Behaviour is widget specific and it is currently used in preview mode

load(*curve*)

Add curve to widget

new_curve(**args*, ***kwargs*)

Create a new curve

preview_widget(*parent=None*)

Return a Qt widget suitable for preview during loading

See also *ResultsDialog* If the object returned is not None, then it should have also an attribute *name*.

remove(*curve*)

Remove curve from widget

set_color(*curve*, *color*)

Set color for widget

class pymeasure.display.widgets.dock_widget.**DockWidget**(*name*, *procedure_class*, *x_axis_labels=None*, *y_axis_labels=None*, *linewidth=1*, *layout_path='./'*, *layout_filename=''*, *parent=None*)

Bases: *TabWidget*, 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*
- **layout_path** – Directory path to save dock layout state. Default is './'
- **layout_filename** – Optional filename for dock layout file. Default: *current procedure class* + "_dock_layout.json"
- **parent** – Passed on to QtWidgets.QWidget. Default is None

contextMenuEvent(*self*, *a0: QContextMenuEvent | None*)

new_curve(*results*, *color=<PyQt5.QtGui.QColor object>*, ***kwargs*)

Create a new curve

save_dock_layout()

Save the current layout of the docks and the plot settings. When running the GUI you can access this function by right-clicking in the widget area to bring up the context menu and selecting "Save Dock Layout"

class pymeasure.display.widgets.table_widget.**PandasModelBase**(*column_index=None*, *results_list=[]*, *parent=None*)

Bases: QAbstractTableModel

This class provided a model to manage multiple panda dataframes and display them as a single table.

The multiple pandas dataframes are provided as ResultTable class instances and all of them share the same number of columns.

There are some assumptions: - Series in the dataframe are identical, we call this number k - Series length can be different, we call this number l(x), where x=1..n

The data can be presented as follow: - By column: each series in a separate column, in this case table shape will be: (k*n) x (max(l(x) x=1..n) - By row: column fixed to the number of series, in this case table shape will be: k x (sum of l(x) x=1..n)

columnCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

data(*self*, *index: QModelIndex*, *role: int = Qt.ItemDataRole.DisplayRole*) → Any

headerData(*section*, *orientation*, *role*)

Return header information

Override method from QAbstractTableModel

pandas_column_count()

Return total column count of the panda dataframes

The value depends on the geometry selected to display dataframes

pandas_row_count()

Return total row count of the panda dataframes

The value depends on the geometry selected to display dataframes

rowCount(*self*, *parent: QModelIndex = QModelIndex()*) → int

translate_to_global(*results*, *row*, *col*)

Translate from single results coordinates to full table coordinates

translate_to_local(*row*, *col*)

Translate from full table coordinate to single results coordinates

class pymeasure.display.widgets.table_widget.**PandasModelByColumn**(*column_index=None*, *results_list=[]*, *parent=None*)

Bases: *PandasModelBase*

pandas_column_count()

Return total column count of the panda dataframes

The value depends on the geometry selected to display dataframes

pandas_row_count()

Return total row count of the panda dataframes

The value depends on the geometry selected to display dataframes

translate_to_global(*results*, *row*, *col*)

Translate from single results coordinates to full table coordinates

translate_to_local(*row*, *col*)

Translate from full table coordinate to single results coordinates

class pymeasure.display.widgets.table_widget.**PandasModelByRow**(*column_index=None*, *results_list=[]*, *parent=None*)

Bases: *PandasModelBase*

pandas_column_count()

Return total column count of the panda dataframes

The value depends on the geometry selected to display dataframes

pandas_row_count()

Return total row count of the panda dataframes

The value depends on the geometry selected to display dataframes

translate_to_global(*results*, *row*, *col*)

Translate from single results coordinates to full table coordinates

translate_to_local(*row*, *col*)

Translate from full table coordinate to single results coordinates

class pymeasure.display.widgets.table_widget.**ResultsTable**(*results*, *color*, *column_index=None*, *force_reload=False*, *wdg=None*, ***kwargs*)

Bases: QObject

Class representing a panda dataframe

class pymeasure.display.widgets.table_widget.**Table**(*refresh_time=0.2*, *check_status=True*, *force_reload=False*, *layout_class=<class 'pymeasure.display.widgets.table_widget.PandasModelByColumn'>*, *column_index=None*, *float_digits=6*, *parent=None*)

Bases: QTableView

Table format view of *Experiment* objects

setModel(*self*, *model: QAbstractItemModel | None*)

set_model(*model_class*)

Replace model with new instance of model_class

class pymeasure.display.widgets.table_widget.**TableWidget**(*name*, *columns*, *by_column=True*, *column_index=None*, *refresh_time=0.2*, *float_digits=6*, *check_status=True*, *parent=None*)

Bases: *TabWidget*, QWidget

Widget to display experiment data in a tabular format

clear_widget()

Clear widget content

Behaviour is widget specific and it is currently used in preview mode

load(*table*)

Add curve to widget

new_curve(*results*, *color=<PyQt5.QtGui.QColor object>*, ***kwargs*)

Create a new curve

preview_widget(*parent=None*)

Return a widget suitable for preview during loading

remove(*table*)

Remove curve from widget

set_color(*table*, *color*)

Change the color of the pen of the curve

6.12 Windows classes

class pymeasure.display.windows.managed_image_window.**ManagedImageWindow**(*procedure_class*, *x_axis*, *y_axis*, *z_axis=None*, ***kwargs*)

Bases: *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**(*procedure_class*, *x_axis=None*, *y_axis=None*, *linewidth=1*, *log_fmt=None*, *log_datefmt=None*, ***kwargs*)

Bases: *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
- **log_fmt** – formatting string for the log-widget
- **log_datefmt** – formatting string for the date in the log-widget
- ****kwargs** – optional keyword arguments that will be passed to *ManagedWindowBase*

class pymeasure.display.windows.managed_window.**ManagedWindowBase**(*procedure_class*, *widget_list=()*, *inputs=()*, *displays=()*, *log_channel=''*, *log_level=20*, *parent=None*, *sequencer=False*, *sequencer_inputs=None*, *sequence_file=None*, *inputs_in_scrollarea=False*, *enable_file_input=True*, *hide_groups=True*)

Bases: 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 directly, 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 `Load sequence` button
- **inputs_in_scrollarea** – boolean that display or hide a scrollbar to the input area
- **enable_file_input** – a boolean controlling whether a *FileInputWidget* to specify where the experiment's result will be saved is displayed (True, default) or not (False). This widget contains a field to enter the (base of the) filename (with or without extension; if absent, the extension will be appended). This field also allows for placeholders to use parameter-values and metadata-value in the filename. The widget also has a field to select the directory where the file is to be stored, and a toggle to control whether the data should be saved to the selected file, or not (i.e., to a temporary file instead).
- **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*)

Queue a measurement based on the parameters in the input-widget.

Semi-abstract method, which must be overridden by the child class if the filename- and directory-inputs are disabled. When filename- and directory inputs are enabled, overwriting is not required, but can be done for custom naming, input processing, or other features.

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)
```

(continues on next page)

(continued from previous page)

```
        self.manager.queue(experiment)
```

reveal_in_file_explorer(*filename: str*) → None

Method to open the file explorer at the location of the given filename.

Args:
filename (str): Path to the file to be revealed in the file explorer.

save_experiment_copy(*source_filename*)

Save a copy of the datafile to a selected folder and file. Primarily useful for experiments that are stored in a temporary file.

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**(*plotter*, *refresh_time=0.1*, *linewidth=1*, *parent=None*)

Bases: 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**(*procedure_class*, *x_axis=None*, *y_axis=None*, *linewidth=1*, *log_fmt=None*, *log_datefmt=None*, ***kwargs*)

Bases: *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
- **linewidth** – linewidth for the displayed curves, default is 1
- **log_fmt** – formatting string for the log-widget
- **log_datefmt** – formatting string for the date in the log-widget

- ****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 BaseChannelCreator(*cls*, ***kwargs*)

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with *CommonBase.add_child()*. The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.

- ****kwargs** – Keyword arguments for all children.

class MultiChannelCreator(*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 attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.

- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns
Instance of the created child.

ask(*command*, *query_delay=None*)

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=None*, ***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 and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

static control(*get_command*, *set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=None*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=','*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **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

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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 get_channel_pairs(*cls*)

Return a list of all the Instrument's channel pairs

static get_channels(*cls*)

Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

static measurement(*get_command*, *docs*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *command_process=None*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=','*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value

- **get_process_list** – A function that takes the value list and processes it.
- **command_process** – A function that take a command and allows processing before executing the command, for getting

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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*)

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*, ***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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the *ask()* method.

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

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Implement in subclass!

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

class pymeasure.instruments.**Instrument**(*adapter*, *name*, *includeSCPI=None*, ***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** – An obligatory boolean, which toggles the inclusion of standard SCPI commands

  Deprecated since version 0.14: If True, inherit the *SCPIMixin* class instead.
- ****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 and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clears the instrument status byte

property complete

Get the synchronization bit.

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

Get the identification of the instrument.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get 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

Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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 and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

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=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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='Fake Instrument'*, *includeSCPI=False*, ***kwargs*)

Bases: *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**(*name='Mock instrument'*, *wait=0.1*, ***kwargs*)

Bases: *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

Control the 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

Control frame height in pixels.

property frame_width

Control frame width in pixels.

property output_voltage

Control the voltage.

property time

Control the elapsed time.

property voltage

Measure the voltage.

property wave

Measure a waveform.

7.2 Generic Instrument Types Mixins

You can use these Mixins as additional parent classes for your instrument. For more informations, see *Common instrument types*.

class pymeasure.instruments.generic_types.**SCPIMixin**(**args*, ***kwargs*)

Mixin class for SCPI instruments with the default implementation of base SCPI commands.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

reset()

Reset the instrument.

property status

Get the status byte and Master Summary Status bit.

7.3 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

7.4 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

7.5 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:

7.6 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*.

7.6.1 Active Technologies AWG-401x 1.2GS/s Arbitrary Waveform Generator

class `pymeasure.instruments.activetechnologies.`**AWG401x_AFG**(*adapter*, ***kwargs*)

Bases: `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.channels[1].shape = "SINUSOID"    # Sets a sine waveform on CH1
wfg.channels[1].frequency = 4.7e3     # Sets the frequency to 4.7 kHz on CH1
wfg.channels[1].amplitude = 1         # Set amplitude of 1 V on CH1
wfg.channels[1].offset = 0            # Set the amplitude to 0 V on CH1
wfg.channels[1].enabled = True        # Enables the CH1

wfg.channels[2].shape = "SQUARE"      # Sets a square waveform on CH2
wfg.channels[2].frequency = 100e6     # Sets the frequency to 100 MHz on CH2
wfg.channels[2].amplitude = 0.5         # Set amplitude of 0.5 V on CH2
wfg.channels[2].offset = 0            # Set the amplitude to 0 V on CH2
wfg.channels[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
```

ch_1

Channel

ChannelAFG

ch_2

Channel
ChannelAFG

property enabled

Control whether the generation of signals is enabled (bool).

class pymeasure.instruments.activetechnologies.**AWG401x_AWG**(*adapter*, ***kwargs*)

Bases: 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].channels[i].voltage_high = 1       # Sets high voltage = 1
    wfg.entries[1].channels[i].voltage_low = 0        # Sets low voltage = 1
    wfg.entries[1].channels[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].channels[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: Sequence

Dummy List Class to list every sequencer entry. The content is loaded in real-time.

class WaveformsLazyDict(*parent*)

Bases: 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

Control the burst count parameter.(dynamic)

property burst_count_max

Get the maximum burst count parameter.

property burst_count_min

Get the minimum burst count parameter.

property enabled

Control whether generation of signals in enabled.

property entry_level_strategy

Control 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

Get the number of analog channels.

property num_dch

Get the number of digital channels.

remove_file(*file_name*, *path=None*)

Remove a specified file

property run_mode

Control 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

Get the run state of the AWG. The possible values are: STOPPED, WAITING_TRIGGER, RUNNING

property sample_decreasing_strategy

Control 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

Control 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

Control the sample rate for the Sampling Clock.(dynamic)

property sampling_rate_max

Get the maximum sample rate for the Sampling Clock.

property sampling_rate_min

Get 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

Control 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

Get 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

class pymeasure.instruments.activetechnologies.AWG401x.**ChannelAFG**(*instrument*, *id*)

Bases: ChannelBase

Implementation of a Active Technologies AWG-4000 channel in AFG mode.

property baseline_offset

Control the offset level for the specified channel. The offset range setting depends on the amplitude parameter. (dynamic)

property baseline_offset_max

Get the maximum offset voltage level that can be set to the output waveform.

property baseline_offset_min

Get the minimum offset voltage level that can be set to the output waveform.

property frequency

Control the frequency of the output waveform. This command is available when the Run Mode is set to any setting other than Sweep. The output frequency range setting depends on the type of output waveform. If you change the type of output waveform, it may change the output frequency because changing waveform

types affects the setting range of the output frequency. The output frequency range setting depends also on the amplitude parameter.(dynamic)

property frequency_max

Get the maximum frequency that can be set to the output waveform.

property frequency_min

Get the minimum frequency that can be set to the output waveform.

property load_impedance

Control the output load impedance for the specified channel. The specified value is used for amplitude, offset, and high/low level settings. You can set the impedance to any value from 1 to 1 M. The default value is 50 .

property output_impedance

Control the instrument output impedance, the possible values are: 5 Ohm or 50 Ohm (default).

property phase

Control the phase of the output waveform for the specified channel. The value is in degrees.(dynamic)

property phase_max

Get the maximum phase that can be set to the output waveform.

property phase_min

Get the minimum phase that can be set to the output waveform.

property shape

Control the shape of the carrier waveform. Allowed choices depends on the choosen modality, please refer on instrument manual. When you set this property with a different value, if the instrument is running it will be stopped. Can be set to: SIN<USOID>, SQU<ARE>, PULS<E>, RAMP, PRN<OISE>, DC, SINC, GAUS<SIAN>, LOR<ENTZ>, ERIS<E>, EDEC<AY>, HAV<ERSINE>, ARBB, EFIL<E>, DOUBLEPUL<SE>

property voltage_amplitude

Control the output amplitude for the specified channel. The measurement unit of amplitude depends on the selection operated using the voltage_unit property. If the carrier is Noise the amplitude is Vpk instead of Vpp. If the carrier is DC level this command causes an error. The range of the amplitude setting could be limited by the frequency and offset parameter of the carrier waveform. (dynamic)

property voltage_amplitude_max

Get the maximum amplitude voltage level that can be set to the output waveform.

property voltage_amplitude_min

Get the minimum amplitude voltage level that can be set to the output waveform.

property voltage_high

Control the high level of the waveform. The high level could be limited by noise level to not exceed the maximum amplitude. If the carrier is Noise or DC level, this command and this query cause an error.(dynamic)

property voltage_high_max

Get the maximum high voltage level that can be set to the output waveform.

property voltage_high_min

Get the minimum high voltage level that can be set to the output waveform.

property voltage_low

Control the low level of the waveform. The low level could be limited by noise level to not exceed the maximum amplitude. If the carrier is Noise or DC level, this command and this query cause an error.(dynamic)

property voltage_low_max

Get the maximum low voltage level that can be set to the output waveform.

property voltage_low_min

Get the minimum low voltage level that can be set to the output waveform.

property voltage_offset

Control the offset level for the specified channel. The offset range setting depends on the amplitude parameter. (dynamic)

property voltage_offset_max

Get the maximum offset voltage level that can be set to the output waveform.

property voltage_offset_min

Get the minimum offset voltage level that can be set to the output waveform.

property voltage_unit

Control the units of output amplitude, the possible choices are: VPP, VRMS, DBM. This command does not affect the offset, high level, or low level of output.

7.7 Aculight

This section contains specific documentation on the (Lockheed Martin) Aculight instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.7.1 Aculight Argos 2400 OPO system

class pymeasure.instruments.aculight.argos.**Argos**(*adapter*, *name='Argos 2400 OPO system'*, *query_delay=0.01*, ***kwargs*)

Bases: *Instrument*

Communication model for the Aculight Argos 2400 OPO system.

Note that the device can only return all values at once, which takes around 100 ms. If you need more than one value, read the *state* property instead of individual properties. Reading any property reads all properties and returns the corresponding value.

Note also, that the block temperature cannot be set remotely, only via the front panel.

Cable type: RS232 null modem female DB9 connectors: The pins 2 and 3 have to be crossed between both sides. Pin2 is RX and Pin3 is Tx, Pin 5 is ground.

Parameters

- **adapter** – resource name
- **query_delay** – Delay between write and read in seconds.

check_set_errors()

Read after setting.

property etalon

Control the etalon angle in degrees.

property seed_voltage

Control the seed source tuning voltage.

property state: State

Get the current state of the system as a namped tuple

Return named tuple
with 'crystal_temperature', 'crystal_temperature_setpoint', 'etalon_angle', and 'seed_voltage'.

property temperature

Get the current crystal temperature in °C.

property temperature_setpoint

Control the crystal temperature setpoint in °C.

property version

Get the firmware version.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

7.8 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*.

7.8.1 Advantest R3767CG Vector Network Analyzer

class pymeasure.instruments.advantest.advantestR3767CG.**AdvantestR3767CG**(*adapter*, *name='Advantest R3767CG'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Advantest R3767CG VNA. Implements controls to change the analysis range and to retrieve the data for the trace.

property center_frequency

Control the center Frequency in Hz.

property id

Get the instrument identification.

property span_frequency

Control the frequency span in Hz.

property start_frequency

Control the starting frequency in Hz.

property stop_frequency

Control the stopping frequency in Hz.

property trace_1

Get the data array from trace 1 after formatting.

7.8.2 Advantest R6245/R6246 DC Voltage/Current Sources/Monitors

Main Classes

class pymeasure.instruments.advantest.advantestR624X.**AdvantestR6245**(*adapter*, *name='Advantest R6245 SourceMeter'*, ***kwargs*)

Bases: *AdvantestR624X*

Main instrument class for Advantest R6245 DC Voltage/Current Source/Monitor

ch_A

Channel
SMUChannel

ch_B

Channel
SMUChannel

class pymeasure.instruments.advantest.advantestR624X.**AdvantestR6246**(*adapter*, *name='Advantest R6246 SourceMeter'*, ***kwargs*)

Bases: *AdvantestR624X*

Main instrument class for Advantest R6246 DC Voltage/Current Source/Monitor

ch_A

Channel
SMUChannel

ch_B

Channel
SMUChannel

class pymeasure.instruments.advantest.advantestR624X.**AdvantestR624X**(*adapter*, *name='R624X Source meter Base Class'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Advantest R624X series (channel A and B) SourceMeter and provides a high-level interface for interacting with the instrument.

This is the base class for both AdvantestR6245 and AdvantestR6246 devices. It's not necessary to instantiate this class directly instead create an instance of the AdvantestR6245 or AdvantestR6246 class as shown in the following example:

```
smu = AdvantestR6246("GPIB::1")
smu.reset()                                                    # Set default␣
↪parameters
smu.ch_A.current_source(source_range = CurrentRange.FIXED_60mA,
                        source_value = 0,                      # Source current at␣
↪0 A
                        voltage_compliance = 10)               # Voltage␣
↪compliance at 10 V
```

(continues on next page)

(continued from previous page)

```
smu.ch_A.enable_source()                                    # Enables the␣
↪source output
smu.ch_A.measure_voltage()
smu.ch_A.current_change_source = 5e-3                       # Change to 5mA
print(smu.read_measurement())                               # Read and print␣
↪the voltage
smu.ch_A.standby()                                          # Put channel A in␣
↪standby
```

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.

check_errors()

Read all errors from the instrument.

Returns

List of error entries.

enable_source()

Put channel A & B into the operating state (CN).

> **Note**
>
> When the 'interlock control' of the 'SCT' command is '2' and the clock signal is 'HI', it will not enter the operating state.

standby()

Put channel A & B in standby mode (CL).

clear_status_register()

Clears the Standard Event Status Register (SESR) and related queues (excluding output queues) (*CLS).

property srq_enabled

Set a boolean that controls whether the GPIB SRQ feature is enabled, takes values of True or False (S0/S1).

Type

bool

The SRQ feature of the GPIB bus provides hardware handshaking between the GPIB controller card in the PC and the instrument. This allows synchronization between moving data to the PC with the state of the instrument without the need to use time delay functions.

trigger()

Outputs the trigger signal or the start of sweep and search measurement to both A and B channels and the trigger link (XE).

> **Note**

- When both A channel and B channel are waiting for a trigger, both channels are triggered.
- When either channel A or B is waiting for a trigger, only the channel that is waiting for a trigger is triggered.
- When both A channel and B channel are waiting for sweep start, this will apply sweep start to both channels.
- When either channel A or B is in the sweep start waiting state, only the channel in the sweep start waiting state is started.
- When either channel A or B is waiting for a trigger and the other is waiting for a sweep start, trigger and sweep start are applied, respectively.
- When the trigger link is ON and this is the master unit, set the *TRG signal on the trigger link bus to TRUE.
- When the trigger link is ON and the master unit, the trigger link is activated.

stop()

Stops the sweep when the sweep is started by the XE command or the trigger input signal (SP).

set_digital_output(*values*)

Outputs a 16-bit signal from the DIGITAL OUT output terminal on the rear panel. You can set up to 9 output data (DIOS). If there are multiple values specified, the data is output at intervals of about 2ms and fixed as the final data.

Parameters
values (*int or list*) – Digital out bit values

Note

The output of digital data to the DIGITAL OUT pin is only the bits specified by the DIOE command. Bits that are not specified will result in alarm output or unused, and no digital data will be output.

property sweep_delay_time

Set the sweep delay time (Ta) or generation / delay time (Ta) of the master channel and slave channel during delayed sweep operation or synchronous operation between pulse measurements (GDLY).

Type
float

Note

If the sweep delay time does not meet (Ta<Tw and Ta<Td+Tit), an execution error will occur and it will not be set:

Tw: Pulse width Td: Major delay time Tit: Integration time

init_sequence()

This function starts the redirection of *write()* to *store_sequence_command()* to automatically create a sequence program.

start_sequence(*repeat=1*)

This function starts the sequence program which is initiated by *init_sequence()* and ended by

end_sequence().

end_sequence()

This function ends the sequence program which is initiated by *init_sequence()*.

sequence_wait(*wait_mode*, *wait_value*)

Waits for program execution and is used only for sequence programs (WAIT).

Parameters

- **wait_mode** (*int*) – Whether wait time (1) or trigger input count (2) is specified
- **wait_value** (*float*) – Wait time or trigger input count as specified by wait_mode

This command has the following functions:

- Make the execution of the next program wait for the specified time.
- Makes the next program execution wait until the specified number of triggers is input.

Regardless of the wait mode, if the wait data is 0, the wait operation is not performed. When the wait mode is "2", the following commands and signals can be used as trigger inputs:

- XE (XE 0, XE 1, XE 2)
- *TRG
- GET command (group execute trigger)
- Trigger input signal on rear panel

start_sequence_program(*start*, *stop*, *repeat*)

Starts from the program number until the stop of the sequence program (RU). Executes sequentially up to the program number, and repeats for the number of times of specified.

Parameters

- **start** (*int*) – Number of the program to start from ranging 1 to 100
- **stop** (*int*) – Number of the program to stop at ranging from 1 to 100
- **repeat** (*int*) – Number of times repeated from 1 to 100

store_sequence_command(*line*, *command*)

Stores the program to be executed in the sequence program (ST). If the program already exists, it is replaced with the new sequence.

Parameters

- **line** (*int*) – Line number specified of memory location
- **command** (*str*) – Command(s) specified to be stored delimited by a semicolon (;)

interrupt_sequence_command(*action*)

Interrupts the sequence program executed by the *start_sequence_program()* command (SQSP).

Parameters

action (*SequenceInterruptionType*) – Specifies sequence interruption setup

property sequence_program_number

Measure the amount of program sequences stored in the sequence memory (LNUB?).

sequence_program_listing(*line*)

This is a query command to know the command list stored in the program number of the sequence program memory (LST?).

Parameters
action (*int*) – Specifying the memory location for reading the commands

Returns
Commands stored in sequence memory

Return type
str

trigger_output_signal(*trigger_output*, *alarm_output*, *scanner_output*)

Directly output the trigger output signal, alarm output signal, scanner (start/stop) output signal from GPIB (OSIG).

Parameters

- **trigger_output** (*int*) – Number specifying type of trigger output
- **alarm_output** (*int*) – Number specifying type of alaram output
- **scanner_output** (*int*) – Number specifying the type of scanner output

Trigger output:

1. Do not output to trigger output.
2. Output a negative pulse to the trigger output.

Alarm output:

1. Finish output GO, LO.HI both set to HI level. (reset)
2. Finish output Set GO to LO level.
3. Set home output LO to LO level.
4. Terminate output HI to LO level.

Scanner - (start/stop) output:

1. Set the scanner scoot output to HI level. Output a negative pulse to the stop output.
2. Make the scanner start output low.
3. Output a HI level for the scanner start output and a negative pulse for the stop output.

set_output_format(*delimiter_format*, *block_delimiter*, *terminator*)

Sets the format and terminator of the output data output by GPIB (FMT).

Parameters

- **delimiter_format** (*int*) – Type of delimiter format
- **block_delimiter** (*int*) – Type of block delimiter
- **terminator** (*int*) – Type of termination character

The output of <EOI> (End or Identify) is output at the following timing: 1,2: Simultaneously with LF 4: Simultaneously with the last output data

If the output data format is specified as binary format, the terminator is fixed to <EOI> only and the terminator selection is ignored.

delimiter_format:

1. ASCII format with header
2. No header, ASCII format
3. Binary format

block_delimiter:

1. Make it the same as the terminator.
2. Use semicolon ;
3. Use comma ,

terminator:

1. CR, LF<EOI>
2. LF<EOI>
3. LF
4. <EOI>

1st character header:	
A)	Normal measurement data
B)	Measurement data during overrange
C)	Compliance (limiter) is working.
D)	Oscillation detection is working.
E)	[Indicates the generated data]
F)	Measurement data when an error occurs in the search measurement
26)	Measurement data is not stored in the buffer memory.

2nd character header:	
A)	A-channel data during asynchronous operation (A-channel generation data)
B)	B-channel data during asynchronous operation (B channel generation data)
I)	A-channel data for synchronous, sweeping, delayed sweep, and double synchronous sweep operations.
J)	B-channel data for synchronous, sweeping, delayed sweep, and double synchronous sweep operations.

3rd character header:	
A)	Current generation, voltage measurement (ISVM) [Current generation]
B)	Voltage generation, current measurement (VSIM) [Voltage generation]
C)	Current generation, current measurement (ISIM)
D)	Voltage generation, voltage measurement (VSVM)
E)	Current generation, external voltage measurement (IS, EXT, VM)
F)	Voltage generation, external current measurement (VS, EXT, IM)
G)	Current generation, external current measurement (IS, EXT. IM)
H)	Voltage generation, external voltage measurement (VS, EXT, VM)
26)	The measurement data is not stored in the buffer memory.

4th character header:	
A)	No operation (fixed to A)
B)	Null operation result
C)	The result of the comparison operation is GO.
D)	The result of the comparison operation is LO.
E)	The result of the comparison operation is HI.
F)	The result of null operation + comparison operation is GO.
G)	The result of null operation + comparison operation is LO.
H)	The result of null operation + comparison operation is HI.
26)	Measurement data is not stored in the buffer memory.

property service_request_enable_register

Control the contents of the service request enable register (SRER) in the form of a *SRER* `IntFlag` (`*SRE`).

> **Note**
>
> Bits other than the RQS bit are not cleared by serial polling. When *power_on_clear()* is set, status byte enable register, SESER, device operation enable register, channel operation, the enable register is cleared and no SRQ is issued.

property event_status_enable

Control the standard event status enable. (`*ESE`)

property power_on_clear

Control the power on clear flag, takes values True or False. (`*PSC`)

property device_operation_enable_register

Control the device operation output enable register (DOER) (`DOE?`).

property digital_out_enable_data

Control the contents of digital out enable data set (`DIOE`).

property status_byte_register

Measure the contents of the status byte register and MSS bits without using a serial poll (`*STB?`).

The Status Byte Register has a hierarchical structure. ERR, DOP, ESB, and COP bits, except RQS and MAV, have lower-level status registers. Each register is paired with an enable register that can be selected to output to the Status Byte register or not. The status byte register also has an enable register, which allows you to select whether or not to issue a service request SRQ.

> **Note**
>
> *STB? command can read bit 6 as MSS (logical OR of other bits).

property event_status_register

Measure the contents of the standard event status register (SESR) in the form of a *SESR* IntFlag (*ESR?).

> **Note**
>
> SESR is cleared after being read.

property device_operation_register

Measure the contents of the device operations register (DOR) in the form of a *DOR* IntFlag (DOC?).

property error_register

Measure the contents of the error register (ERR?).

property self_test

Get the result of self test after running it.

property trigger_link_function_enabled

Set a boolean that controls whether the trigger link function is enabled, takes values of True or False. (TLNK)

Type
bool

property display_enabled

Set a boolean that controls whether the display is on or off, takes values of True or False. (DISP)

Type
bool

property line_frequency

Set the used power supply frequency (LF) to 50 or 60hz. With this command, the integration time per PLC for the measurement will be one cycle of the power supply frequency you are using.

Type
int

property store_config

Set the memory area for the config to be stored at (SAV). There are five memory areas from 0 to 4 for storing.

Type
int

property load_config

Set the memory area for the config to be loaded from (RCL). There are five areas (0~4) where parameters can be loaded by the RCL command.

Type
int

set_lo_common_connection_relay(*enable*, *lo_relay=None*)

Turn the connection relay on/off between the A channel LO (internal analog common) and the LO (internal analog common) of the B channel (LTL).

Parameters

- **enable** (*bool*) – A boolean property that controls whether or not the connection relay is enabled. Valid values are True and False.
- **lo_relay** (*bool, optional*) – A boolean property that controls whether or not the internal analog common relay is enabled. Valid values are True, False and None (don't change lo relay setting).

read_measurement()

Reads the triggered value, for example triggered by the external input.

class pymeasure.instruments.advantest.advantestR624X.**SMUChannel**(*parent*, *id*, *voltage_range*, *current_range*)

Bases: *Channel*

Instantiated by main instrument class for every SMUChannel

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.

clear_measurement_buffer()

Clears the measurement data buffer (MBC).

set_output_type(*output_type*, *measurement_type*)

Sets the output method and type of the GPIB output (OFM).

Parameters

- **output_type** (int or *OutputType*) – A property that controls the type of output
- **measurement_type** (int or *MeasurementType*) – A property that controls the measurement type

> **Note**
>
> For the format of the output data, refer to *AdvantestR624X.set_output_format()*. For DC and pulse measurements, the output method is fixed to '1' (real-time output). When the output method '3' (buffering output) is specified, the measured data is not stored in memory.

property analog_input

Set the analog input terminal (ANALOG INPUT) on the rear panel ON or OFF (FL).

Type
int

1. Turn off the analog input.
2. Analog input ON, gain x1.
3. Analog input ON, gain x2.5.

property trigger_output_timing

Set the timing of the trigger output signal output from TRIGGER OUT on the rear panel (TOT). the status in the form of a *TriggerOutputSignalTiming* IntFlag.

Type
TriggerOutputSignalTiming

set_scanner_control(*output*, *interlock*)

Sets the SCANNER CONTROL (START, STOP) output signal and INTERLOCK input signal on the rear panel (SCT).

Parameters

- **output** (*int*) – A property that controls the scanner output
- **interlock** (*int*) – A property that controls the scanner interlock type

output:

1. Scanner, Turn off the control signal output.
2. Output to the scanner control signal at the start / stop of the sweep.
3. Operate / Standby Scanner, Output to the control signal.

interlock:

1. Turn off the interlock signal input.
2. Set as a stamper when the interlock signal input is HI.
3. When the interlock signal input is HI, it is on standby, and when it is LO, it is operated.

property trigger_input

Set the type of trigger input (TJM).

Type
TriggerInputType

Trigger input types	1	2	3
*TRG	O	O	X
XE 0	O	O	X
XE Channel	O	O	O
GET	O	O	X
Trigger input signal	O	X	X

O can be used, X cannot be used

Note

The sweep operation cannot be started by the trigger input signal. Be sure to start it with the 'XE' command. Once started, it is possible to advance the sweep with a trigger input signal.

property fast_mode_enabled

Set the channel response mode to fast or slow, takes values of True or False (FL).

Type
bool

property sample_hold_mode

Set the integration time of the measurement (MST).

Type
SampleHold

> **Note**
>
> - Valid only for pulse measurement and pulse sweep measurement.
> - In sample hold mode, the AD transformation is just before the fall of the pulse width.
> - The sample hold mode cannot be set during DC measurement and DC sweep measurement. When set to sample-and-hold mode, the integration time is 100 µs. However, in 2-channel synchronous operation, if one channel is in pulse generation and the other is in sample-and-hold mode, the DC measurement side also operates in sample-and-hold mode.
> - When performing pulse measurement and pulse sweep measurement, it is necessary to satisfy the restrictions on the pulse width (Tw), pulse period (Tp), and measure delay time (Td) of the WT command. If the constraint is not satisfied, the integration time is unchanged. To lengthen the integration time, first change the pulse width (Tw) and pulse period (Tp). When shortening the pulse width and pulse cycle, shorten the integration time first.

set_sample_mode(*mode*, *auto_sampling=True*)

Sets synchronous, asynchronous, tracking operation and search measurement between channels (JM).

Parameters

- **mode** (*SampleMode*) – Sample Mode
- **auto_sampling** (*bool, optional*) – Whether or not auto sampling is enabled, defaults to True

set_timing_parameters(*hold_time*, *measurement_delay*, *pulsed_width*, *pulsed_period*)

Set the hold time, measuring time, pulse width and the pulse period (WT).

Parameters

- **hold_time** (*float*) – total amount of time for the complete pulse, until next pulse comes
- **measurement_delay** (*float*) – time between measurements
- **pulsed_width** (*float*) – Time specifying the pulse width
- **pulsed_period** (*float*) – Time specifying the pulse period

> **Note**
>
> Pulse measurement has the following restrictions depending on the pulse period (Tp) setting. (For pulse sweep measurements, there are no restrictions.)
>
> - Tp < 2ms : Not measured.
> - 2ms <= Tp < 10ms : Measure once every 5 ~ 20ms.
> - 10ms <= Tp: Measured at each pulse generation.

select_for_output()

This is a query command to select a channel and to output the measurement data (FCH?). When the output channel is selected by the FCH command, the measured data of the same channel is returned until the output channel is changed by the next FCH command.

> **Note**
>
> Reading measurements with the RMM command does not affect channel specification with the FCH command. In the default state, the measurement data of channel A is output.

trigger()

Measurement trigger command for sweep, start search measurement or sweep step action (XE).

measure_voltage(*enable=True*, *internal_measurement=True*, *voltage_range=VoltageRange.AUTO*)

Sets the voltage measurement ON/OFF, measurement input, and voltage measurement range as parameters (RV).

Parameters

- **enable** (*bool, optional*) – boolean property that enables or disables voltage measurement. Valid values are True (Measure the voltage flowing at the OUTPUT terminal) and False (Measure the voltage from the rear panel -ANALOG COMMON).
- **internal_measurement** (*bool, optional*) – A boolean property that enables or disables the internal measurement.
- **voltage_range** (*VoltageRange*, optional) – Specifying voltage range

voltage_source(*source_range*, *source_value*, *current_compliance*)

Sets the source range, source value and the current compliance for the DC (constant voltage) measurement (DV).

Parameters

- **source_range** (*VoltageRange*) – Specifying source range
- **source_value** (*float*) – A number specifying the source voltage value
- **current_compliance** (*float*) – A number specifying the current compliance

> **Note**
>
> Regardless of the specified current compliance polarity, both polarities (+ and -) are set. The current compliance range is automatically set to the minimum range that includes the set value.

voltage_pulsed_source(*source_range*, *pulse_value*, *base_value*, *current_compliance*)

Sets the source range, pulse value, base value and the current compliance of the pulse (voltage) measurement (PV).

> **Note**
>
> Regardless of the specified current compliance polarity, both polarities (+ and -) are set. The current compliance range is automatically set to the minimum range that includes the set value.

property change_source_voltage

Set new target voltage (SPOT).

Type
float

> **Note**
>
> Only the DC action source value and pulse action pulse value are changed using the currently set DC action and pulse action parameters. Measure after the change and set the channel to output the measured data to the specified ch. In other words, it's the same as running the following commands:
>
> 1. DV/DI/PV/PI
> 2. XE xx
> 3. FCH xx

voltage_fixed_level_sweep(*voltage_range*, *voltage_level*, *measurement_count*, *current_compliance*, *bias=0*)

Sets the fixed level sweep (voltage) generation range, level value, current compliance and the bias value (FXV).

> **Note**
>
> Regardless of the specified current compliance polarity, both polarities (+ and -) are set. The current compliance range is automatically set to the minimum range that includes the set value.

voltage_fixed_pulsed_sweep(*voltage_range*, *pulse*, *base*, *measurement_count*, *current_compliance*, *bias=0*)

Sets the fixed pulse (voltage) sweep generation range, pulse value, base value, number of measurements, current compliance and the bias value (PXV).

> **Note**
>
> Regardless of the specified current compliance polarity, both polarities (+ and -) are set. The current compliance range is automatically set to the minimum range that includes the set value.

voltage_sweep(*sweep_mode*, *repeat*, *voltage_range*, *start_value*, *stop_value*, *steps*, *current_compliance*, *bias=0*)

Sets the sweep mode, number of repeats, source range, start value, stop value, number of steps, current compliance, and the bias value for staircase (linear/log) voltage sweep (WV).

> **Note**
>
> - Sweep mode, number of repeats, and number of steps are subject to the following restrictions.
>   - Let N = number of steps, m = 1 (one-way sweep), m = 2 (round-trip sweep).
>   - When the OFM command sets the output data output method to 1 or 2 m x number of refreshes x N <= 2048

        * m x N <= 2048 when the OFM command sets the output data output method to 3.
    - Regardless of the specified current compliance polarity, both polarities (+ and -) are set.
    - The current compliance range is automatically set to the minimum range that includes the set value.

voltage_pulsed_sweep(*sweep_mode*, *repeat*, *voltage_range*, *base*, *start_value*, *stop_value*, *steps*, *current_compliance*, *bias=0*)

Sets the sweep mode, repeat count, generation range, base value, start value, stop value, number of steps, current compliance and the bias value for a pulse wave (linear/log) voltage sweep (PWV).

Note

- The sweep mode, number of refreshes, and number of steps are subject to the following restrictions:
  - Let N = number of steps, m = l (one-way sweep), m = 2 (round-trip sweep).
    * When the OFM command sets the output data output method to 1 or 2 m x number of refreshes x N <= 2048
    * m x N <= 2048 when the OFM command sets the output data output method to 3.
- For the current compliance polarity, regardless of the specified current compliance polarity, the compliance of both polarities (+ and -) is set.
- The current compliance range is automatically set to the minimum range that includes the set value.

voltage_random_sweep(*sweep_mode*, *repeat*, *start_address*, *stop_address*, *current_compliance*, *bias=0*)

Sets the sweep mode, repeat count, start address, stop address, current compliance and the bias value of constant voltage random sweep (MDWV).

Note

- Sweep mode, number of repeats, start address and stop address are subject to the following restrictions:
  - Start address < Stop address
  - Let N = number of steps, m = l (one-way sweep), m = 2 (round-trip sweep).
    * When the OFM command sets the output data output method to 1 or 2 m x number of refreshes x N <= 2048
    * m x N <= 2048 when the OFM command sets the output data output method to 3.
- Regardless of the specified current compliance polarity, both polarities (+ and -) are set.
- The current compliance range is automatically set to the minimum range that includes the set value.

voltage_random_pulsed_sweep(*sweep_mode*, *repeat*, *start_address*, *stop_address*, *current_compliance*, *bias=0*)

Sets the sweep mode, repeat count, base value, start address, stop address, current compliance and the bias

value of the constant voltage random pulse sweep (MPWV).

> **Note**
>
> - Sweep mode, number of repeats, start address and stop address are subject to the following restrictions:
>   - Start address < Stop address
>   - Let N = number of steps, m = 1 (one-way sweep), m = 2 (round-trip sweep).
>   - When the OFM command sets the output data output method to 1 or 2 m x number of refreshes x N <= 2048
>   - m x N <= 2048 when the OFM command sets the output data output method to 3.
> - Regardless of the specified current compliance polarity, both polarities (+ and -) are set.
> - The current compliance range is automatically set to the minimum range that includes the set value.

voltage_set_random_memory(*address*, *voltage_range*, *output*, *current_compliance*)

The command stores the specified value to the randomly generated data memory (RMS).

Stored generated values are swept within the specified memory address range by the MDWV, MDWI, MPWV, MPWI commands.

current_source(*source_range*, *source_value*, *voltage_compliance*)

Sets the source range, source value, voltage compliance of the DC (constant current) measurement (DI).

Parameters

- **source_range** (*CurrentRange*) – Specifying source range
- **source_value** (*float*) – A number specifying the source current value
- **voltage_compliance** (*float*) – A number specifying the voltage compliance

> **Note**
>
> Regardless of the specified voltage compliance polarity, both polarities (+ and -) are set. The voltage compliance range is automatically set to the minimum range that includes the set value.

current_pulsed_source(*source_range*, *pulse_value*, *base_value*, *voltage_compliance*)

Sets the source range, pulse value, base value and the voltage compliance of the pulse (current) measurement (PI).

> **Note**
>
> Regardless of the specified voltage compliance polarity, both polarities (+ and -) are set. The voltage compliance range is automatically set to the minimum range that includes the set value.

property change_source_current

Set new target current (SPOT).

Type
float

> **Note**
>
> Only the DC action source value and pulse action pulse value are changed using the currently set DC action and pulse action parameters. Measure after the change and set the channel to output the measured data to the specified ch. In other words, it's the same as running the following commands:
>
> 1. DV/DI/PV/PI
> 2. XE xx
> 3. FCH xx

current_fixed_level_sweep(*current_range*, *current_level*, *measurement_count*, *voltage_compliance*, *bias=0*)

Sets the fixed level sweep (current) generation range, level value, voltage compliance and the bias value (`FXI`).

> **Note**
>
> Regardless of the specified voltage compliance polarity, both polarities (+ and -) are set. The voltage compliance range is automatically set to the minimum range that includes the set value.

current_fixed_pulsed_sweep(*current_range*, *pulse*, *base*, *measurement_count*, *voltage_compliance*, *bias=0*)

Sets the fixed pulse (current) sweep generation range, pulse value, base value, number of measurements, voltage compliance and the bias value (`PXI`).

> **Note**
>
> Regardless of the specified voltage compliance polarity, both polarities of + and - are set. The voltage compliance range is automatically set to the minimum range that includes the set value.

current_sweep(*sweep_mode*, *repeat*, *current_range*, *start_value*, *stop_value*, *steps*, *voltage_compliance*, *bias=0*)

Sets the sweep mode, number of repeats, source range, start value, stop value, number of steps, voltage compliance and bias value for the staircase (linear/log) current sweep (`WI`).

> **Note**
>
> - The sweep mode, number of refreshes, and number of steps are subject to the following restrictions:
>   - Let N = number of steps, m = 1 (one-way sweep), m = 2 (round-trip sweep).
>   - When the OFM command sets the output data output method to 1 or 2, m x number of repeats x N <= 2048.
>   - m x N <= 2048 when the OFM command sets the output data output method to 3.

- Regardless of the specified voltage compliance polarity, both polarities (+ and -) are set.
- The voltage compliance range is automatically set to the minimum range that includes the set value.

current_pulsed_sweep(*sweep_mode*, *repeat*, *current_range*, *base*, *start_value*, *stop_value*, *steps*, *voltage_compliance*, *bias=0*)

Sets the sweep mode, repeat count, generation range, base value, start value, stop value, number of steps, voltage compliance and the bias value for a pulse wave (linear/log) current sweep (PWI).

Note

- The sweep mode, number of refreshes, and number of steps are subject to the following restrictions:
  - Let N = number of steps, m = 1 (one-way sweep), m = 2 (round-trip sweep).
    - When the OFM command sets the output data output method to 1 or 2, m x number of repeats x N <= 2048.
    - m x N <= 2048 when the OFM command sets the output data output method to 3.
- Regardless of the specified voltage compliance polarity, both polarities (+ and -) are set.
- The voltage compliance range is automatically set to the minimum range that includes the set value.

measure_current(*enable=True*, *internal_measurement=True*, *current_range=CurrentRange.AUTO*)

Set the current measurement ON/OFF, measurement input, and current measurement range as parameters (RI).

Parameters

- **enable** (*bool, optional*) – boolean property that enables or disables current measurement. Valid values are True (Measure the current flowing at the OUTPUT terminal) and False (Measure the current from the rear panel -ANALOG COMMON).
- **internal_measurement** (*bool, optional*) – A boolean property that enables or disables the internal measurement.
- **current_range** (*CurrentRange*, optional) – Specifying voltage range

current_random_sweep(*sweep_mode*, *repeat*, *start_address*, *stop_address*, *current_compliance*, *bias=0*)

Sets the sweep mode, repeat count, start address, stop address, voltage compliance and the bias value of constant current random sweep (MDWI).

Note

- Sweep mode, number of repeats, start address and stop address are subject to the following restrictions:
  - Start address < Stop address
  - Let N = (stop number 1 - start number + 1), m = 1 (one-way sweep), m = 2 (round-trip sweep).

    - When the output data output method is set to 1 or 2 with the OFM command m x number of repeats x N <= 2048
    - When the output data output method is set to 3 with the OFM command m x N <= 2048
  - For the voltage compliance polarity, regardless of the specified voltage compliance polarity, both polarities of + and – are set.
  - The voltage compliance range is automatically set to the minimum range that includes the set value.

current_random_pulsed_sweep(*sweep_mode*, *repeat*, *start_address*, *stop_address*, *current_compliance*, *bias=0*)

Sets the sweep mode, repeat count, base value, start address, stop address, voltage compliance and the bias value of constant current random pulse sweep (MPWI).

Note

- Sweep mode, number of repeats, start address and stop address are subject to the following restrictions:
  - Start address < Stop address
  - Let N = (stop number 1 - start number + 1), m = 1 (one-way sweep), m = 2 (round-trip sweep).
    - When the output data output method is set to 1 or 2 with the OFM command m x number of repeats x N <= 2048
    - When the output data output method is set to 3 with the OFM command m x N <= 2048
- For the voltage compliance polarity, regardless of the specified voltage compliance polarity, both polarities of + and – are set.
- The voltage compliance range is automatically set to the minimum range that includes the set value.

current_set_random_memory(*address*, *current_range*, *output*, *voltage_compliance*)

Store the current parameters to randomly generated data memory (RMS).

Stored generated values are swept within the specified memory address range by the MDWV, MDWI, MPWV, MPWI commands.

read_random_memory(*address*)

Return memory specified by address location (RMS?).

Parameters
address (*int*) – Adress to specify memory location.

Returns
Set values returned by the device from the specified address location.

Return type
str

enable_source()

Put the specified channel into an operating state (CN).

standby()

Put the specified channel into standby state (CL).

stop()

Stops the sweep when the sweep is started by the XE command or the trigger input signal (SP).

output_all_measurements()

Output all measurements in the measurement data buffer of the specified channel (RMM?).

> **Note**
>
> For the output format, refer to *AdvantestR624X.set_output_format()*. When a memory address where no measurement data is stored is read, 999.999E+99 will be returned.

read_measurement_from_addr(*addr*)

Output only one measurement at the specified memory address from the measurement data buffer of the specified channel.

Parameters
addr (*int*) – Specifies the address to read from.

Returns
float Measurement data

> **Note**
>
> For the output format, refer to *AdvantestR624X.set_output_format()*. When a memory address where no measurement data is stored is read, 999.999E+99 will be returned.

property measurement_count

Measure the number of measurements contained in the measurement data buffer (NUB?).

property null_operation_enabled

Set a boolean that controls whether the null operation is enabled, takes values of True or False (NUG).

Type
bool

> **Note**
>
> - Null data is not rewritten even if the null operation is disabled.
> - Null data is rewritten only when null operation is changed from OFF to ON or initialized in case of DC operation or pulse operation.

set_wire_mode(*four_wire*, *lo_guard=True*)

Used to switch remote sense and to set the LO-GUARD relay ON/OFF. It operates regardless of operating state or standby state (OSL).

Parameters

- **four_wire** (*bool*) – A boolean property that enables or disables four wire measurements. Valid values are True (enables 4-wire sensing) and False (enables two-terminal sensing).
- **lo_guard** (*bool*) – A boolean property that enables or disables the LO-GUARD relay.

property auto_zero_enabled

Set the auto zero option to ON or OFF. Valid values are True (enabled) and False (disabled) (CM).

Type
bool

This command sets auto zero (automatically calibrate the zero point of the measured value operation.

1. Periodically perform auto zero.
2. Auto zero once, no periodic auto zeros thereafter.

When the auto zero mode is set to True, the following operations are performed.

- For DC operation and pulse operation:
  - At the end of one sweep, if he has exceeded the last autozero by more than 10 seconds, he will do one autozero.
  - If sweep start is specified during auto zero, the sweep will start after auto zero ends.
- Sweep operation
  - Auto zero is performed once every 10 seconds.
  - If measurement or pulse output is specified during auto zero, it will be executed after auto zero ends.

set_comparison_limits(*comparison*, *voltage_value*, *upper_limit*, *lower_limit*)

Sets the channel ON/OFF based on the measurement comparison and the data of the upper and lower limits to be compared (CMD).

Parameters

- **comparison** (*bool*) – A boolean property that controls whether or not the comparison function is enabled. Valid values are True or False.
- **voltage_value** (*bool*) – A boolean property that controls whether or not voltage or current values are passed. Valid values are True or False.
- **upper_limit** (*float*) – Number specifying the upper comparison limit
- **lower_limit** (*float*) – Number specifying the lower comparison limit

property relay_mode

Set the HI/LO relays for standby mode. This command does not operate the Operate Relay (OPM).

Type
int

1. When executing an operation only the HI side turns ON, in standby both HI and LO are turned OFF.
2. When executing an operation only the LO side turns ON, in standby both HI and LO are turned OFF.
3. When executing an operation both HI and LO turn ON, in standby both HI and LO are turned OFF.
4. When executing an operation only the HI side turns ON, in standby only the HI side is turned OFF.

property operation_register

Measure the contents of the Channel Operations Register (COR) in the form of a *COR* IntFlag (COC?).

property output_enable_register

Control the settings of the channel operation output enable register (COER) in the form of a *COR* IntFlag ?(COE?).

calibration_init()

Initialize the calibration data (CINI).

calibration_store_factor()

Store the calibration factor in the non-volatile memory (EEPROM) (CSRT).

property calibration_measured_value

Set the measured value measured by an external standard for the generated value of this instrument and start calibration (STD).

Type
float

property calibration_generation_factor

Set the increment or decrement for the generation calibration factor of the current generation range (CCS). It is used when the generated value deviates from the true value.

Type
float

property calibration_factor

Set the increment of the measurement calibration factor of the current measurement range (CCM).

Type
float

class pymeasure.instruments.advantest.advantestR624X.**SampleHold**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**SampleMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**VoltageRange**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**CurrentRange**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**SweepMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**OutputType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**TriggerInputType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**MeasurementType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class pymeasure.instruments.advantest.advantestR624X.**SequenceInterruptionType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

1. Release pause state is a valid command only in the sequence program pause state. otherwise it is ignored.
2. Pause state enters the pause state when the currently executing program ends.
3. Abort sequence program stops the sequence program when the currently executing program ends. If the currently running program is a sweep operation, interrupt the sweep operation and stop the sequence program. The output value will be the bias value.

class pymeasure.instruments.advantest.advantestR624X.**DOR**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

bit assigment for the Device Operation Register (DOR):

Bit (dec)	Description
13	Indicates that the fast tokens program is running.
12	Error in search measurement
11	End of sequence program/high-speed sequence program execution
10	Sequence program Pause state
9	Fan stop detection
8	Self-test error occurred (logic part)
7	Trigger wait state in trigger link master operation
6	Calibration mode status
5	Trigger link ON state
4	Trigger link bus error
3	Sequence program/high-speed sequence 1 program/add/de) waiting
2	Wait for sequence program wait time
1	Sequence program running
0	Synchronous operation state

class pymeasure.instruments.advantest.advantestR624X.**COR**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

bit assigment for the Channel Operations Register (COR):

Bit (dec)	Description
14	The result of the comparison operation is HI
13	The result of the comparison operation is GO
12	The result of the comparison operation is LO
11	Overheat detection
10	Overload detection
9	Oscillation detection
8	Compliance detection
7	Synchronous operation master channel
6	Measurement data output specification
5	There is measurement data
4	Self-test error occurrence (analog part)
3	Measurement data buffer full
2	Waiting for trigger
1	End of sweep
0	Operated state

class pymeasure.instruments.advantest.advantestR624X.**SRER**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

bit assigment for the Service Request Enable Register (SRER):

Bit (dec)	Description
0	none
1	ERR Set when any of QYE, DDE, EXE, or CME in the Standard Event Status Register (SESR) is set.
2	DOP Set when a bit in the device operation register for which the enable register is set to enabled is set. Cleared by reading the device operation register.
3	none
4	MAV Set when output data is set in the output queue. Cleared when output data is read.
5	ESB Set when a bit in the Standard Event Status Register (SESR) is set and the enable register is set to Enabled. Cleared by reading SESR.
6	RQS (MSS) Set when bit O to bit 5 and bit 7 of the Status Byte register are set. (this bit is read-only)
7	COP Set when a bit in the Channel Operations Register is set with the Enable Register set to Enable. Cleared by reading the Channel Operations Register.

class pymeasure.instruments.advantest.advantestR624X.**SESR**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

bit assigment for the Standard Event Status Register (SESR):

Bit (dec)	Description
0	OPC (Operation Complete) not used
1	RQC unused
2	QYE (Query Error) Set when the output queue overflows when reading without output data.
3	DDE (Device Dependent Error) Set when an error occurs in the self-test.
4	EXE (Execution Error) Set when the input data is outside the range set internally, or when the command cannot be executed.
5	CME (Command Error) Set when an undefined header or data format is wrong, or when there is a syntax error in the command.
6	URQ unused
7	PON Set when power is switched from OFF to ON.

class pymeasure.instruments.advantest.advantestR624X.**TriggerOutputSignalTiming**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

bit assigment for the timing of the trigger output signal output from TRIGGER OUT on the rear panel:

Bit (dec)	Description
5	At the end of the sweep
4	At the end of the pulse width
3	At the end of the pulse cycle
2	At the end of measurement
1	At the start of measurement
0	At the start of occurrence

Contents

General Information

The R6245/6246 Series are DC voltage/current sources and monitors having source measurement units (SMUs) with 2 isolated channels. The series covers wide source and measurement ranges. It is ideal for measurement of DC characteristics of items ranging from separate semiconductors such as bipolar transistors, MOSFETs and GaAsFETs, to ICs and power devices. Further, due to the increased measuring speed and synchronized 2-channel measurement function, device I/O characteristics can be measured with precise timing at high speed which was previously difficult to accomplish. Due to features such as the trigger link function and the sequence programming function which automatically performs a series of evaluation tests automatically, the R6245/6246 enable much more efficient evaluation tests.

There is a total of 99 commands, the majority of commands have been implemented. Device documentation is in Japanese, and the device options are enormous. The implementation is based on 6245S-GPIB-B-FHJ-8335160E01.pdf, which can be downloaded from the ADCMT website.

Examples

Initialization of the Instrument

```
from pymeasure.instruments.advantest import AdvantestR6246
from pymeasure.instruments.advantest.advantestR624X import *

smu = AdvantestR6246("GPIB::1")
```

Simple dual channel measurement example

Measurement characteristics:

Channel A: Vce = 20V Channel B: Ib = 10uA - 60uA

```
smu = AdvantestR6246("GPIB::1")
smu.reset()                                             # Set default parameters
smu.ch_A.set_sample_mode(SampleMode.PULSED_SYNC)        # Pulsed synchronized
smu.ch_A.voltage_source(source_range = VoltageRange.AUTO,
                        source_value = 20,
                        current_compliance = 0.06)
smu.ch_A.measure_current()
smu.ch_B.current_source(source_range = CurrentRange.AUTO,
                        source_value = 1E-5,            # Source current at 10 uA
                        voltage_compliance = 5)         # Voltage compliance at 5 V
smu.ch_B.measure_voltage()
smu.enable_source()                                     # Enables source A & B

for i in range(10, 60):
    k = i * 0.000001
    smu.ch_B.current_change_source = k                  # Set current from 10 uA to 60␣
↪uA

    smu.trigger()                                       # Trigger measurement
    smu.ch_A.select_for_output()
    Ic = smu.read_measurement()                         # Read channel A measurement
    smu.ch_B.select_for_output()
    Vbe = smu.read_measurement()                        # Read channel B measurement
    print(f'Ic={Ic}, Vbe={Vbe}')                        # Print measurements

smu.standby()                                           # Put channel A & B in standby
```

Program example for DC measurement

Measurement characteristics:

Function: VSIM - Source voltage and measure current Trigger voltage: 10V Current compliance: 0.5A Measurement delay time: 1ms Integration time: 1 PLC Response: Fast

After operating, the measurement is repeated 10 times with a trigger command and he prints out the results.

```
smu = AdvantestR6246("GPIB::1")
smu.reset()                                                    # Set default parameters
smu.ch_A.set_sample_mode(SampleMode.ASYNC, False)              # Asynchronous␣
↪operation and single shot sampling by trigger and command
smu.ch_A.voltage_source(source_range = VoltageRange.FIXED_BEST,
                        source_value = 10,
                        current_compliance = 0.5)              # compliance of 0.5A
smu.ch_A.measure_current()                                     # Measure current
smu.ch_A.set_timing_parameters(hold_time = 0,                  # 0 sec hold time
                               measurement_delay = 1E-3,       # 1ms delay between␣
↪measurements
                               pulsed_width = 5E-3,            # 5ms pulse width
                               pulsed_period = 10E-3)          # 10ms pulse period
smu.ch_A.sample_hold_mode = SampleHold.MODE_1PLC               # Sample at 1 power␣
↪line cycle
smu.ch_A.fast_mode_enabled = True                              # Set channel response␣
↪to fast
smu.ch_A.enable_source()                                       # Set channel in␣
↪operating state
smu.ch_A.select_for_output()                                   # Select channel for␣
↪measurement output

for i in range(1, 10):
    smu.ch_A.trigger()                                         # Trigger a measurement
    measurement = smu.read_measurement()
    print(f"NO {i} {measurement}")

smu.ch_A.standby()                                             # Put channel A in␣
↪standby mode
```

Program example for DC measurement (with external trigger)

Measurement characteristics:

Function: VSIM - Source voltage and measure current Source voltage: 10 V Base voltage 1 V Current compliance: 0.5 A Pulse width: 5 ms Pulse period: 10 ms Measurement delay time: 1 ms Integration time: 1 ms Response: Fast

After operating, an external trigger input signal is pulsed to measure the channel operation register. Reads the fixed end bit, captures the measurement data, and prints out the measurement result.

```
smu = AdvantestR6246("GPIB::1")
smu.reset()                                                    # Set default parameters

smu.ch_A.auto_zero_enabled = False
smu.ch_A.set_sample_mode(SampleMode.ASYNC, False)              # Asynchronous␣
↪operation and single shot sampling by trigger and command
smu.ch_A.voltage_pulsed_source(
      source_range = VoltageRange.FIXED_BEST,
      pulse_value = 10,
      base_value = 1,
      current_compliance = 0.5)
smu.ch_A.measure_current()                                     # Measure current
```

(continues on next page)

(continued from previous page)

```
smu.ch_A.fast_mode_enabled = True                                    # Set channel response␣
↪to fast
smu.ch_A.sample_hold_mode = SampleHold.MODE_1mS                      # Sample at 1mS
smu.ch_A.set_timing_parameters(hold_time = 0,                        # 0 sec hold time
                               measurement_delay = 1E-3,             # 1ms delay between␣
↪measurements
                               pulsed_width = 5E-3,                  # 5ms pulse width
                               pulsed_period = 10E-3)                # 10ms pulse period
smu.ch_A.trigger_input = TriggerInputType.ALL                        # Mode 1 enables the␣
↪trigger input signal
smu.ch_A.output_enable_register = COR.HAS_MEASUREMENT_DATA           # Measurement data␣
↪available
smu.service_request_enable_register = SRER.COP                       # COP Set when a bit in␣
↪the Channel Operations Register is set with the Enable Register set to Enable.
smu.ch_A.enable_source()                                             # Set channel in␣
↪operating state
smu.ch_A.select_for_output()                                         # Select channel for␣
↪measurement output

for i in range(1, 10):
    while not smu.ch_A.operation_register & COR.HAS_MEASUREMENT_DATA:
        pass

    measurement = smu.read_measurement()
    print(f"NO {i} {measurement}")

    while not smu.ch_A.operation_register & COR.WAITING_FOR_TRIGGER:
        pass

smu.ch_A.standby()                                                   # Put channel A in␣
↪standby mode
```

Program example for pulse measurement

Measurement characteristics:

Function: ISVM - Source current and measure voltage Pulse generation current: 100mA Base current: 1mA Voltage compliance: 5V Pulse width: 0 Pulse period : 0 Measurement delay time: 0 Integration time: 1ms Response: Fast

After the operation, repeat the measurement 10 times with the trigger command and print out the measurement results.

```
smu = AdvantestR6246("GPIB::1")
smu.reset()                                                          # Set default␣
↪parameters
smu.ch_A.set_sample_mode(SampleMode.ASYNC, auto_sampling = False)
smu.ch_A.current_pulsed_source(
      source_range = CurrentRange.FIXED_600mA,
      pulse_value = 0.1,                                             # 100mA
      base_value = 1E-3,                                             # 1mA
      voltage_compliance = 5)                                        # 5V
smu.ch_A.measure_voltage(voltage_range = VoltageRange.FIXED_BEST)
```

(continues on next page)

(continued from previous page)

```
smu.ch_A.fast_mode_enabled = True                                    # Set channel␣
↪response to fast
smu.ch_A.sample_hold_mode = SampleHold.MODE_1mS                      # Sample at 1mS
smu.ch_A.set_timing_parameters(hold_time = 0,                        # 0 sec hold time
                               measurement_delay = 0,                # 0 sec delay␣
↪between measurements
                               pulsed_width = 0,                     # 0 sec pulse width
                               pulsed_period = 0)                    # 0 sec pulse period
smu.ch_A.enable_source()                                             # Set channel in␣
↪operating state
smu.ch_A.select_for_output()                                         # Select channel for␣
↪measurement output

for i in range(1, 10):
    smu.ch_A.trigger()                                               # Trigger measurement

    measurement = smu.read_measurement()
    print(f"NO {i} {measurement}")

    while not smu.ch_A.operation_register & COR.WAITING_FOR_TRIGGER:
        pass

smu.ch_A.standby()                                                   # Put channel A in␣
↪standby mode
```

Fixed Level Sweep Program Example

Measurement characteristics:

function: VSVM - Voltage source and voltage measurement Level value: 15V Bias value: 0V Number of measurements: 20 times Compliance: 6mA Measuring range: Best fixed range (=60V range) Integration time: 100us Measurement delay time: 0 Hold time: 1ms Sampling mode: automatic sweep Measurement data output method: Buffering output (output of specified data)

After operating, make 20 measurements in fixed sweep. Detect the end of sweep by looking at the Channel Operation Register (COR). After the sweep is finished, read the measured data from 1 to 2 using the RMM command.

```
smu = AdvantestR6246("GPIB::1")

# First we setup our main parameters
smu.reset()                                                          # Set default parameters

smu.ch_A.set_output_type(output_type = OutputType.BUFFERING_OUTPUT_SPECIFIED,
                         measurement_type = MeasurementType.MEASURE_DATA)

smu.set_output_format(delimiter_format = 2,                          # No header, ASCII␣
↪format
                      block_delimiter = 1,                           # Make it the same as␣
↪the terminator
                      terminator = 1)                                # CR, LF<EOI>

smu.ch_A.analog_input = 1                                            # Turn off the analog␣
```

(continues on next page)

(continued from previous page)

```
↪input.

smu.set_lo_common_connection_relay(enable = True)                # Turns the connection␣
↪relay on

smu.ch_A.set_wire_mode(four_wire = False,                        # disable four wire␣
↪measurements
                       lo_guard = True)                          # enable the LO-GUARD␣
↪relay.

smu.ch_A.auto_zero_enabled = False
smu.ch_A.trigger_input = TriggerInputType.ALL                    # Mode 1 enables the␣
↪trigger input signal

# Now we set measurement specific variables
smu.ch_A.clear_measurement_buffer()
smu.ch_A.set_sample_mode(SampleMode.ASYNC, auto_sampling = True)
smu.ch_A.voltage_fixed_level_sweep(voltage_range = VoltageRange.FIXED_60V,
                                   voltage_level = 15,
                                   measurement_count = 20,       # 20 measurements
                                   current_compliance = 6E-3,    # compliance at 6mA
                                   bias = 0)
smu.ch_A.measure_voltage(voltage_range = VoltageRange.FIXED_BEST)
smu.ch_A.sample_hold_mode = SampleHold.MODE_100uS
smu.ch_A.set_timing_parameters(hold_time = 1E-3,                 # 1ms sec hold time
                               measurement_delay = 0,            # 0 sec delay between␣
↪measurements
                               pulsed_width = 0,                 # 0 sec pulse width
                               pulsed_period = 0)                # 0 sec pulse period

smu.ch_A.enable_source()                                         # Set channel in␣
↪operating state
smu.ch_A.trigger()                                               # Start the sweep

while not smu.ch_A.operation_register & COR.END_OF_SWEEP:        # Wait until the sweep␣
↪is done
    pass

# Read measurements
for i in range(1, 20):
    measurement = smu.ch_A.read_measurement_from_addr(i)
    print(i, measurement)

smu.ch_A.standby()                                               # Put channel A in␣
↪standby mode
```

7.9 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*.

If the instrument you are looking for is not here, also check *HP* for older instruments or *Keysight* for newer ones.

7.9.1 Agilent 8257D Signal Generator

class pymeasure.instruments.agilent.**Agilent8257D**(*adapter*, *name='Agilent 8257D RF Signal Generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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 percent, 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 percentage
- **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 internal 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.

7.9.2 Agilent 8722ES Vector Network Analyzer

class pymeasure.instruments.agilent.**Agilent8722ES**(*adapter*, *name='Agilent 8722ES Vector Network Analyzer'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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.

7.9.3 Agilent E4408B Spectrum Analyzer

class pymeasure.instruments.agilent.**AgilentE4408B**(*adapter*, *name='Agilent E4408B Spectrum Analyzer'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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).

7.9.4 Agilent E4980 LCR Meter

class pymeasure.instruments.agilent.**AgilentE4980**(*adapter*, *name='Agilent E4980A/AL LCR meter'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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: Resistance [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

7.9.5 Agilent E5062A Vector Network Analyzer

class pymeasure.instruments.agilent.**AgilentE5062A**(*adapter*, *name='Agilent E5062A Vector Network Analyzer'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Agilent E5062A Vector Network Analyzer

This VNA has 4 separate channels, each of which has its own sweep. The channels are stored in the `channels` dictionary. Channels can be enabled even if not displayed. Many trace-specific operations, including reading out data, happen at the channel level, but pertain to only the *active trace*.

Each channel can display up to 4 traces (controlled via the `visible_traces` parameter). Each channel also has a `display_layout` property that controls the layout of the traces in the channel. The traces are accessed via the `traces` dictionary (i.e. `vna.channels[1].traces[1]`).

The VNA supports multiple transfer formats. This API only supports the IEEE 64-bit floating point format. The VNA is configured for this format during initialization, and the data transfer format is not a publicly accessible property.

This class implements only a subset of the total E5062A functionality. The more significant missing functionality includes (TODO):

- editing the scale of the display
- markers
- calibration
- power sweeps

```
# connect to the VNA and make a measurement consisting of 10 averages
vna = AgilentE5062A("TCPIP::192.168.2.233::INSTR")
ch = vna.channels[1]            # use channel 1
ch.visible_traces = 4            # use all 4 traces
for i, (tr, parameter) in enumerate(zip(
        ch.traces.values(),
        ['S11', 'S12', 'S21', 'S22'])):
    tr.parameter = parameter

ch.averages = 10                # use 10x averaging
ch.averaging_enabled = True     # turn averaging on
ch.trigger_continuous = False   # turn off continuous triggering
vna.trigger_source = 'BUS'      # require remote trigger
vna.abort()                     # stop the current sweep
ch.restart_averaging()          # clear current averaging
for _ in range(ch.averages):
    ch.trigger_initiate()       # arm channel 1 for single sweep
    vna.trigger_single()        # send a trigger
    vna.wait_for_complete()     # wait until the sweep is complete

# see `agilentE5062A.data` for an example of saving the measurement
```

ch_1

Channel
VNAChannel

ch_2

Channel
VNAChannel

ch_3

Channel
VNAChannel

ch_4

Channel
VNAChannel

abort()

Abort the current sweep (affects the trigger fsm, see page 80 of the programmer's manual for details)

property display_layout

Control the layout of the windows (channels) on the display (str). See the list of valid options below:

- D1
- D12
- D1_2
- D112
- D1_1_2
- D123

- D1_2_3
- D12_33
- D11_23
- D13_23
- D12_13
- D1234
- D1_2_3_4
- D12_34

In general, an occurance of a number denotes the associated channel being visible and an occurence of '_' denotes a vertical split. Multiple occurences of the same number denote the channel having a larger window than the other channels. Refer to Figure 3-1 in the the programmer's manual for details.

Note that this splits windows for multiple *channels*, not traces. Basic S-param measurements most likely want to use only a single channel, but with multuple *traces* instead. See e.g. `AgilentE5062A.channels[1].visible_traces`

property output_enabled

Control whether to turn on the RF stimulus (bool). The stimulus needs to be on to perform any measurement. Beware that the stimulus is on by default! (i.e. after reset())

pop_err()

Pop an error off the error queue. Returns a tuple containing the code and error description. An error code 0 indicates success.

The Error queue can be cleared using the standard SCPI `agilentE5062A.clear()` method.

trigger()

Regardless of the trigger setting, generate a trigger (if the trigger has been initialized, i.e. is in the trigger wait state)

trigger_bus()

If the trigger source is BUS and the VNA is waiting for a trigger (the trigger has been initialized), generate a trigger.

trigger_single()

like trigger(), but the *complete* SCPI property (synchronization bit) waits for the sweep to be complete (see *agilentE5062A.wait_for_complete()*).

property trigger_source

Control the trigger source (str). From the documentation:

- INT: Uses the internal trigger to generate continuous triggers automatically (default).
- EXT: Generates a trigger when the trigger signal is inputted externally via the Ext Trig connector or the handler interface.
- MAN: Generates a trigger when the key operation of [Trigger] - Trigger is executed from the front panel.
- BUS: Generates a trigger when the `*TRG` command is executed.

wait_for_complete(*attempt=1*, *max_attempts=20*)

Wait a potentially long time for the synchronization bit. This is useful in conjunction with *agilentE5062A.trigger_single()* to wait for a single sweep to complete.

class pymeasure.instruments.agilent.agilentE5062A.**VNAChannel**(**args*, ***kwargs*)

Bases: *Channel*

A measurement channel of the E5061A/E5062A

property IF_bandwidth

Control the IF bandwidth in Hz (int), from 10 Hz to 30 kHz. Default 30 kHz.

activate()

Activate (select) the current channel. Note that the channel must first be displayed (via AgilentE5062A.display_layout) for it to be activatable.

property attenuation

Control the stimulus attenuation in dB (positive int), from 0 to 40 in incrememnts of 10. Default is 0 dB. The allowable stimulus power range is a 15 dB range: (`attenuation` - 5 dB, `attenuation` + 10 dB).

This requires the power range extension, and the command is ignored if the extension is not installed.

property averages

Control how many averages to take, from 1-999 (int). Note that `averaging_enabled` needs to be true for averaging to be enabled.

property averaging_enabled

Control whether to average the measurement data (bool).

property data

Measure the Formatted Data of the *active trace*.

Each trace consists of `scan_points` plotted either vs. frequency or in something like a smith-chart configuration. This property does not access any frequency information. So for rectangular plots, this query returns only the y-values of the trace (no frequency information). For smith- and polar- plots, this two values per data point.

This function returns a tuple containing both a primary and a secondary data numpy array. The secondary data is all zeros for all trace formats that are not smith or polar. The implication for this is that the best way to save complex S-parameter data in one go is to use a Smith (Real/Imag) or Polar (Real/Imag) trace format, (SCOMplex and POLar, respectively).

The formatted data array is settable in the VNA but not implemented in this python API.

Frequency data is accessed via the `frequencies` property.

```
# build an s-parameter matrix by measuring the trace data,
# where s_matrix[i] = [[S11, S12], [S21, S22]] @ freqs[i]
freqs = ch.frequencies
s_matrix = np.empty((freqs.size, 4), dtype=np.complex64)
for i, tr in enumerate(ch.traces.values()):
    tr.activate()
    ch.trace_format = 'POL'
    re, im = ch.data
    s_matrix[:, i] = re + 1j * im
s_matrix = s_matrix.reshape(-1, 2, 2)  # ready for use with e.g. skrf
```

property display_layout

Control the graph layout of the traces in the channel (str). Does not affect how many traces are active. See the list of valid options:

- D1

- D12
- D1_2
- D112
- D1_1_2
- D123
- D1_2_3
- D12_33
- D11_23
- D13_23
- D12_13
- D1234
- D1_2_3_4
- D12_34

In general, an occurance of a number denotes the associated trace having it's own subplot and an occurence of '_' denotes a vertical split. Multiple occurences of the same number denote the trace having a larger window than the other traces. Refer to Figure 3-1 in the the programmer's manual for details. If a trace is active but it's associated number is not present in the `display_layout` identifier, it is plotted in the first subplot.

property frequencies

Measure the frequency in Hz associated with each data point of the *active trace*. Returns a numpy array.

property power

Control the simulus power in dBm (float). The allowable range is influenced by the value of `attenuation`. (dynamic)

restart_averaging()

Clear averaging history.

property scan_points

Control the number of points used in a sweep (int). Valid range 2 - 1601.

property start_frequency

Control the start frequency in Hz (float).

property stop_frequency

Control the stop frequency in Hz (float).

property sweep_time

Control the sweep time in seconds (float). The allowable range varies on config and the set value is truncated. Note that `sweep_time_auto_enabled` needs to be `False` for changes to this property to have an effect.

property sweep_time_auto_enabled

Control whether to automatically set the sweep time (bool). You probably want this on to always keep the sweep time to a minimum (given the range, IF BW, and sweep delay time).

property sweep_type

Control the type of the sweep (string).

Setting	Description
LIN	Linear
LOG	Logarithmic
SEGM	Segment
POW	Power

Defaults to linear. Note that the API for configuring segment type sweeps is not implememented in this class.

property trace_format

Control the data format of the *active trace* of the channel (str). Default is MLOGarithmic. From the programmer's manual:

Setting	Description
MLOG	Specifies the logarithmic magnitude format.
PHAS	Specifies the phase format.
GDEL	Specifies the group delay format.
SLIN	Specifies the Smith chart format (Lin/Phase).
SLOG	Specifies the Smith chart format (Log/Phase).
SCOM	Specifies the Smith chart format (Real/Imag).
SMIT	Specifies the Smith chart format (R+jX).
SADM	Specifies the Smith chart format (G+jB).
PLIN	Specifies the polar format (Lin).
PLOG	Specifies the polar format (Log).
POr	Specifies the polar format (Re/Im).
MLI	Specifies the linear magnitude format.
SWR	Specifies the SWR format.
REAL	Specifies the real format.
IMAG	Specifies the imaginary format.
UPH	Specifies the expanded phase format.
PPH	Specifies the positive phase format.

property trigger_continuous

Control whether the channel triggers continuously (bool). If not, after a sweep is complete, the global trigger enters the hold state, and will not respond to triggers.

trigger_initiate()

If the trigger is in the hold state (i.e. `trigger_continuous` is off and the measurement has completed), re-initialize the trigger for a single measurement.

property visible_traces

Control the number of traces visible in the channel (int).

class pymeasure.instruments.agilent.agilentE5062A.**VNATrace**(*parent*, *id*)

Bases: *Channel*

A trace (of a channel) of the E5061A/E5062A

activate()

Select this trace.

property parameter

Control the measurement parameter of the trace (str). Can be S11, S21, S12, or S22

7.9.6 Agilent 34410A Multimeter

class pymeasure.instruments.agilent.**Agilent34410A**(*adapter*, *name='HP/Agilent/Keysight 34410A Multimeter'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

Get AC current, in Amps

property current_dc

Get DC current, in Amps

property resistance

Get Resistance, in Ohms

property resistance_4w

Get Four-wires (remote sensing) resistance, in Ohms

property voltage_ac

Get AC voltage, in Volts

property voltage_dc

Get DC voltage, in Volts

7.9.7 HP/Agilent/Keysight 34450A Digital Multimeter

class pymeasure.instruments.agilent.**Agilent34450A**(*adapter*, *name='HP/Agilent/Keysight 34450A Multimeter'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

Get a capacitance measurement in Farads, based on the active *mode*.

property capacitance_auto_range

Control auto ranging for capacitance.

property capacitance_range

Control 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

Get a continuity measurement in Ohms, based on the active *mode*.

property current

Get a DC current measurement in Amps, based on the active *mode*.

property current_ac

Get an AC current measurement in Amps, based on the active *mode*.

property current_ac_auto_range

Control auto ranging for AC current.

property current_ac_range

Control 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

Control 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

Control auto ranging for DC current.

property current_range

Control 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

Control 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

Get a diode measurement in Volts, based on the active *mode*.

property frequency

Get a frequency measurement in Hz, based on the active *mode*.

property frequency_aperture

Control 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

Control whether auto ranging for AC current in frequency measurements is enabled.

property frequency_current_range

Control 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

Control whether auto ranging for AC voltage in frequency measurements is enabled.

property frequency_voltage_range

Control 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 mode

Control the measurement mode of the multimeter. Can be "current", "ac current", "voltage", "ac voltage", "resistance", "4w resistance", "current frequency", "voltage frequency", "continuity", "diode", "temperature", or "capacitance".

property resistance

Get a resistance measurement in Ohms for 2-wire configuration, based on the active *mode*.

property resistance_4w

Get a resistance measurement in Ohms for 4-wire configuration, based on the active *mode*.

property resistance_4w_auto_range

Control auto ranging for 4-wire resistance.

property resistance_4w_range

Control 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

Control 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

Control auto ranging for 2-wire resistance.

property resistance_range

Control 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

Control 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

Get a temperature measurement in Celsius, based on the active *mode*.

property voltage

Get a DC voltage measurement in Volts, based on the active *mode*.

property voltage_ac

Get an AC voltage measurement in Volts, based on the active *mode*.

property voltage_ac_auto_range

Control auto ranging for AC voltage.

property voltage_ac_range

Control 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

Control 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

Control auto ranging for DC voltage.

property voltage_range

Control 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

Control 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).

7.9.8 Agilent 4155/4156 Semiconductor Parameter Analyzer

class pymeasure.instruments.agilent.agilent4156.**Agilent4156**(*adapter*, *name='Agilent 4155/4156 Semiconductor Parameter Analyzer'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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",
           "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
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

property analyzer_mode

Control 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

Control the 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()

Disable 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

Control the 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

Control the integration time.

- Values: SHORT, MEDIUM, LONG

```
instr.integration_time = "MEDIUM"
```

measure(*period='INF'*, *points=100*)

Perform a single measurement and wait 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()

Stop the ongoing measurement.

```
instr.stop()
```

class pymeasure.instruments.agilent.agilent4156.**AgilentMeasurementChannel**(*parent*, *id*)

Bases: *Channel*

Offers some common properties.

property channel_function

Control the channel function.

- Values: VAR1, VAR2, VARD or CONS.

property channel_mode

Control the channel mode.(dynamic)

property disable

Set the settings of this channel to default value.

```
instr.smu1.disable
```

reset_settings()

Reset the settings of this channel to default value.

```
instr.smu1.reset_settings()
```

property voltage_name

Control 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.**SMU**(*parent*, *id*, ***kwargs*)

Bases: *AgilentMeasurementChannel*

SMU of Agilent 4155/4156 Semiconductor Parameter Analyzer

property compliance

Control 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

Control 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

Control 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.voltage_name = "Vbase"
```

property series_resistance

Control the series resistance of SMU<n>.

- Values: `0OHM`, `10KOHM`, `100KOHM`, or `1MOHM`

```
instr.smu1.series_resistance = "10KOHM"
```

class pymeasure.instruments.agilent.agilent4156.**VAR1**(*parent*, ***kwargs*)

Bases: *VARX*

Class to handle all the specific definitions needed for VAR1. Most common methods are inherited from base class.

property spacing

Control the sweep type of VAR1.

- Values: `LINEAR`, `LOG10`, `LOG25`, `LOG50`.

class pymeasure.instruments.agilent.agilent4156.**VAR2**(*adapter*, ***kwargs*)

Bases: *VARX*

Class to handle all the specific definitions needed for VAR2. Common methods are imported from base class.

property points

Control 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**(*parent*, *id='VARD'*, ***kwargs*)

Bases: *Channel*

Class to handle all the definitions needed for VARD. VARD is always defined in relation to VAR1.

property channel_mode

Control the channel mode.

property compliance

Control the sweep COMPLIANCE value of VARD.

```
instr.vard.compliance = 0.1
```

property offset

Control 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

Control 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**(*parent*, *id*, ***kwargs*)

Bases: *Channel*

Base class to define sweep variable settings.

property channel_mode

Control the channel mode.

property compliance

Control the sweep COMPLIANCE value.

```
instr.var1.compliance = 0.1
```

property start

Control the sweep START value.

```
instr.var1.start = 0
```

property step

Control the sweep STEP value.

```
instr.var1.step = 0.1
```

property stop

Control the sweep STOP value.

```
instr.var1.stop = 3
```

class pymeasure.instruments.agilent.agilent4156.**VMU**(*parent*, *id*, ***kwargs*)

Bases: *AgilentMeasurementChannel*

VMU of Agilent 4155/4156 Semiconductor Parameter Analyzer

class pymeasure.instruments.agilent.agilent4156.**VSU**(*parent*, *id*, ***kwargs*)

Bases: *AgilentMeasurementChannel*

VSU of Agilent 4155/4156 Semiconductor Parameter Analyzer

property constant_value

Control the constant source value of VSU<n>.

```
instr.vsu1.constant_value = 0
```

7.9.9 Agilent 4294A Precision Impedance Analyzer

class pymeasure.instruments.agilent.**Agilent4294A**(*adapter*, *name='Agilent 4294A Precision Impedance Analyzer'*, *read_termination='\n'*, *write_termination='\n'*, *timeout=5000*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Agilent 4294A Precision Impedance Analyzer

property active_trace

Control the active trace

get_data(*path=None*)

Get the measurement data from the instrument after completion.

Parameters
path – Path for optional data export to CSV.

Returns
Pandas Dataframe

property measurement_type

Control the measurement type. See MEASUREMENT_TYPES

property num_points

Control the number of points measured at each sweep

save_graphics(*path=''*)

Save graphics on the screen to a file on the local computer. Adapted from: https://www.keysight.com/se/en/lib/software-detail/programming-examples/4294a-data-transfer-program-excel-vba-1645196.html

property start_frequency

Control the start frequency in Hz

property stop_frequency

Control the stop frequency in Hz

property title

Control the title of the active trace

7.9.10 Agilent 33220A Arbitrary Waveform Generator

class pymeasure.instruments.agilent.**Agilent33220A**(*adapter*, *name='Agilent 33220A Arbitrary Waveform generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

Control the amplitude of the output waveform. Depending on amplitude unit, the unit is Volt (peak-to-peak), Volt (RMS) or dBm. The limits depend on the configured output termination (50 Ohm to High Impedance changes by a factor of 2) and the offset. Supplying out-of-bound values may silently be clipped by the device (float, in V or dBm)

property amplitude_unit

Control the units of the amplitude (string, strict from VPP, VRMS, DBM).

beep()

Causes a system beep.

property beeper_state

Control the state of the beeper (bool).

property burst_mode

Control the burst mode (string, strict from TRIG, TRIGGERED, GAT, GATED).

property burst_ncycles

Control the number of cycles to be output when a burst is triggered (int, strict from 1 to 50000).

property burst_state

Control whether the burst mode is on (bool).

property frequency

Control the frequency of the output waveform. Depending on the output shape, the supported frequency range changes. Supplying out-of-bound values may silently be clipped by the device (float, in Hz)

property offset

Control the voltage offset of the output waveform. This is limited by the amplitude and output termination. Supplying out-of-bound values may silently be clipped by the device. (float, in V)

property output

Control the output of the function generator (bool).

property pulse_dutycycle

Control the duty cycle of a pulse waveform function in percent (float, strict from 0 to 100).

property pulse_hold

Control if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform (string, strict from WIDT, WIDTH, DCYC, DCYCLE).

property pulse_period

Control the period of a pulse waveform function in seconds (float, strict from 200e-9 to 2e3). The period 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

Control 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.

property pulse_width

Control the width of a pulse waveform function in seconds (float, strict from 20e-9 to 2e3).

property ramp_symmetry

Control the symmetry percentage for the ramp waveform (float, strict from 0 to 100).

property remote_local_state

Set the remote/local state of the function generator (string, strict from LOC, LOCAL, REM, REMOTE, RWL, RWLOCK).

property shape

Control the output waveform (string, strict from SINUSOID, SIN, SQUARE, SQU, RAMP, PULSE, PULS, NOISE, NOIS, DC, USER).

property square_dutycycle

Control the duty cycle of a square waveform function (float, strict from 20 to 80).

trigger()

Send a trigger signal to the function generator.

property trigger_source

Control the trigger source (string, strict from IMM, IMMEDIATE, EXT, EXTERNAL, BUS).

property trigger_state

Control whether the output is triggered (bool).

property voltage_high

Control the upper voltage of the output waveform. The limits depend on the output termination. Supplying out-of-bound values may silently be clipped by the device. (float, in V)

property voltage_low

Control the lower voltage of the output waveform. The limits depend on the output termination. Supplying out-of-bound values may silently be clipped by the device. (float, in V)

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.

7.9.11 Agilent 33500 Function/Arbitrary Waveform Generator Family

class pymeasure.instruments.agilent.**Agilent33500**(*adapter*, *name='Agilent 33500 Function/Arbitrary Waveform generator family'*, ***kwargs*)

Bases: *SCPIMixin*, *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.channels[1].shape = 'SIN'           # Sets channel 1 output signal shape␣
```

(continues on next page)

(continued from previous page)

```
↪to sine
generator.frequency = 1e3                # Sets default channel output frequency to␣
↪1 kHz
generator.channels[1].frequency = 1e3          # Sets channel 1 output frequency to␣
↪1 kHz
generator.channels[2].amplitude = 5            # Sets channel 2 output amplitude to␣
↪5 Vpp
generator.channels[2].output = 'on'            # Enables channel 2 output

generator.channels[1].shape = 'ARB'            # Set channel 1 shape to arbitrary
generator.channels[1].arb_srate = 1e6          # Set channel 1 sample rate to 1MSa/s

generator.channels[1].data_volatile_clear()    # Clear channel 1 volatile internal␣
↪memory
generator.channels[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.channels[1].arb_file = 'test'        # Select the transmitted waveform 'test
↪'
```

ch_1

Channel
Agilent33500Channel

ch_2

Channel
Agilent33500Channel

property amplitude

Control the voltage amplitude in Volts (float).(dynamic)

property amplitude_unit

Control the amplitude units (string, strictly 'VPP', 'VRMS', or 'DBM').(dynamic)

property arb_advance

Control how the device advances from data point to data point (str). Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

property arb_file

Control the arbitrary signal to use from the volatile memory of the device.(dynamic)

property arb_filter

Control the filter setting for arbitrary signals (str). Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

property arb_srate

Control the sample rate of the currently selected arbitrary signal in Sa/s (float). Valid values range from 1 µSa/s to 250 MSa/s (maximum range, device-dependent).

beep()

Causes a system beep.

property burst_mode

Control the burst mode type (str, strictly 'TRIG<GERED>' or 'GAT<ED>').(dynamic)

property burst_ncycles

Control the number of cycles to be output when a burst is triggered (int).(dynamic)

property burst_period

Control the period of subsequent bursts in seconds (float).(dynamic)

property burst_state

Control the burst mode state (bool).(dynamic)

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

Set text to be displayed on the front panel of the device (string).

property ext_trig_out

Control whether the trigger out signal is active (bool).

property frequency

Control the waveform frequency in Hz (float). Depends on the specified shape.(dynamic)

property offset

Control the voltage offset in Volts (float).(dynamic)

property output

Control the output state (bool).(dynamic)

property output_load

Control the expected load resistance in Ohms (str or float). The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm.(dynamic)

property phase

Control the waveform phase in degrees (float, from -360 to 360).

phase_sync()

Synchronize the phase of all channels.

property pulse_dutycycle

Control the pulse duty cycle in percent (float, from 0 to 100).(dynamic)

property pulse_hold

Control whether pulse width or duty cycle is maintained when the period or frequency of the waveform is changed (str).(dynamic)

property pulse_period

Control the pulse period in seconds (float). Overwrites frequency. If the period is shorter than the pulse width + the edge time, the edge time and pulse width are adjusted.(dynamic)

property pulse_transition

Control the pulse edge time (both rising and falling) in seconds (float).(dynamic)

property pulse_width

Control the pulse width in seconds (float).(dynamic)

property ramp_symmetry

Control the ramp waveform symmetry in percent (float, from 0 to 100).(dynamic)

property shape

Control the output waveform shape (str).(dynamic)

property square_dutycycle

Control the square wave duty cycle in percent (float).(dynamic)

trigger()

Send a trigger signal to the function generator.

property trigger_source

Control the trigger source (str).

property voltage_high

Control the upper voltage level in Volts (float).(dynamic)

property voltage_low

Control the lower voltage level in Volts (float).(dynamic)

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.

7.9.12 Agilent 33521A Function/Arbitrary Waveform Generator

class pymeasure.instruments.agilent.**Agilent33521A**(*adapter*, ***kwargs*)

Bases: *Agilent33500*

Represents the Agilent 33521A Function/Arbitrary Waveform Generator.

This documentation page shows only methods different from the parent class *Agilent33500*.

ch_1

Channel
Agilent33500Channel

ch_2

Channel
Agilent33500Channel

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.

class pymeasure.instruments.agilent.agilent33500.**Agilent33500Channel**(*parent*, *id*)

Bases: *Channel*

Implementation of a base Agilent 33500 channel

property amplitude

Control the voltage amplitude in Volts (float).(dynamic)

property amplitude_unit

Control the amplitude units (string, strictly 'VPP', 'VRMS', or 'DBM').(dynamic)

property arb_advance

Control how the device advances from data point to data point (str). Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

property arb_file

Control the arbitrary signal to use from the volatile memory of the device.(dynamic)

property arb_filter

Control the filter setting for arbitrary signals (str). Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

property arb_srate

Control the sample rate of the currently selected arbitrary signal in Sa/s (float). Valid values range from 1 µSa/s to 250 MSa/s (maximum range, device-dependent).

property burst_mode

Control the burst mode type (str, strictly 'TRIG<GERED>' or 'GAT<ED>').(dynamic)

property burst_ncycles

Control the number of cycles to be output when a burst is triggered (int).(dynamic)

property burst_period

Control the period of subsequent bursts in seconds (float).(dynamic)

property burst_state

Control the burst mode state (bool).(dynamic)

data_arb(*arb_name*, *data_points*, *data_format='DAC'*)

Uploads an arbitrary trace into the volatile memory of the device for a given channel.

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 for a given channel.

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 memory.

property frequency

Control the waveform frequency in Hz (float). Depends on the specified shape.(dynamic)

property offset

Control the voltage offset in Volts (float).(dynamic)

property output

Control the output state (bool).(dynamic)

property output_load

Control the expected load resistance in Ohms (str or float). The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm.(dynamic)

property phase

Control the waveform phase in degrees (float, from -360 to 360).

property pulse_dutycycle

Control the pulse duty cycle in percent (float, from 0 to 100).(dynamic)

property pulse_hold

Control whether pulse width or duty cycle is maintained when the period or frequency of the waveform is changed (str).(dynamic)

property pulse_period

Control the pulse period in seconds (float). Overwrites frequency. If the period is shorter than the pulse width + the edge time, the edge time and pulse width are adjusted.(dynamic)

property pulse_transition

Control the pulse edge time (both rising and falling) in seconds (float).(dynamic)

property pulse_width

Control the pulse width in seconds (float).(dynamic)

property ramp_symmetry

Control the ramp waveform symmetry in percent (float, from 0 to 100).(dynamic)

property shape

Control the output waveform shape (str).(dynamic)

property square_dutycycle

Control the square wave duty cycle in percent (float).(dynamic)

property voltage_high

Control the upper voltage level in Volts (float).(dynamic)

property voltage_low

Control the lower voltage level in Volts (float).(dynamic)

7.9.13 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'

continues on next page

Table 1 – continued from previous page

Command	Property/Method
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
```

(continues on next page)

(continued from previous page)

```
# 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)
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 because 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,␣
```

(continues on next page)

(continued from previous page)

```
↪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
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*, *name='Agilent B1500 Semiconductor Parameter Analyzer'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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 Execution 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 instrument.

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. Halves 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** (*float*) – 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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

ADC Mode

AUTO = 0

MANUAL = 1

PLC = 2

TIME = 3

class pymeasure.instruments.agilent.agilentB1500.**AutoManual**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Auto/Manual selection

AUTO = 0

MANUAL = 1

class pymeasure.instruments.agilent.agilentB1500.**MeasMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Measurement Mode

SPOT = 1

STAIRCASE_SWEEP = 2

SAMPLING = 10

class pymeasure.instruments.agilent.agilentB1500.**MeasOpMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Sweep Mode

LINEAR_SINGLE = 1

LOG_SINGLE = 2

LINEAR_DOUBLE = 3

LOG_DOUBLE = 4

class pymeasure.instruments.agilent.agilentB1500.**SamplingMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Output after sampling

BASE = 1

BIAS = 2

class pymeasure.instruments.agilent.agilentB1500.**StaircaseSweepPostOutput**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Output after staircase sweep

START = 1

STOP = 2

class pymeasure.instruments.agilent.agilentB1500.**CompliancePolarity**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Compliance polarity

AUTO = 0

MANUAL = 1

class pymeasure.instruments.agilent.agilentB1500.**WaitTimeType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: *CustomIntEnum*

Wait time type

SMU_SOURCE = 1

SMU_MEASUREMENT = 2

CMU_MEASUREMENT = 3

7.9.14 Agilent 4284A LCR Meter

class pymeasure.instruments.agilent.**Agilent4284A**(*adapter*, *name='Agilent 4284A LCR meter'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Agilent 4284A precision LCR meter.

```
agilent = Agilent4284A("GPIB::1::INSTR")

agilent.reset()                         # Return instrument settings to default
                                          values
agilent.frequency = 10e3                # Set frequency to 10 kHz
agilent.voltage = 0.02                  # Set AC voltage to 20 mV
```

(continues on next page)

(continued from previous page)

```
agilent.mode = 'ZTR'                        # Set impedance mode to measure
                                              impedance magnitude [Ohm] and phase
                                              [rad]
agilent.sweep_measurement(
    'frequency', [1e4, 1e3, 100]            # Perform frequency sweep measurement
)                                           # at 10 kHz, 1 kHz, and 100 Hz
agilent.enable_high_power()                 # Enable upper current, voltage, and
                                              bias limits, if properly configured.

agilent.correction.open_enabled = True      # Enable the OPEN correction
agilent.correction.spot1.enabled = True     # Enable the correction for SPOT1
agilent.correction.spot1.frequency = 10e3   # Set SPOT1 frequency to 10 kHz
agilent.correction.spot1.measure_open()     # Measure the OPEN structure for SPOT1
```

correction

Channel
Agilent4284ACorrection

property ac_current

Control AC current level in Amps. Valid range is 50 uA to 20 mA for default, 50 uA to 200 mA in high-power mode.(dynamic)

property ac_voltage

Control AC voltage level in Volts. Range is 5 mV to 2 V for default, 5 mV to 20 V in high-power mode.(dynamic)

property auto_range_enabled

Control whether the impedance auto range is enabled.

property bias_current

Control the DC bias current in Amps. Requires Option 001 (power amplifier / DC bias) to be installed. Maximum is 100 mA.(dynamic)

property bias_enabled

Control whether DC bias is enabled.

property bias_voltage

Control the DC bias voltage in Volts. Maximum is 2 V by default, 40 V in high-power mode.(dynamic)

property frequency

Control AC frequency in Hertz, from 20 Hz to 1 MHz.(dynamic)

property high_power_enabled

Control whether high power mode is enabled. Enabling requires option 001 (power amplifier / DC bias) to be installed.

property impedance_mode

Control impedance measurement function.

- 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]
- LPQ: Parallel inductance [H] and quality factor [number]
- LPD: Parallel inductance [H] and dissipation 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: Resistance [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 impedance_range

Control the impedance measurement range. The 4284A will select an appropriate measurement range for the setting value.

sweep_measurement(*sweep_mode*, *sweep_values*)

Run list sweep measurement using sequential trigger.

Parameters

- **sweep_mode** (*str*) – parameter to sweep across. Must be one of *frequency*, *voltage*, *current*, *bias_voltage*, or *bias_current*.
- **sweep_values** – list of parameter values to sweep across.

Returns

values as configured with *impedance_mode* and list of sweep parameters in format ([val A], [val B], [sweep_values])

trigger()

Execute a bus trigger, regardless of trigger state. Can be used when *trigger_source* is set to *BUS*. Returns result of triggered measurement.

property trigger_continuous_enabled

Control whether trigger state automatically returns to WAIT FOR TRIGGER after measurement.

property trigger_delay

Control trigger delay in seconds. Valid range is 0 to 60, with 1 ms resolution.

trigger_immediate()

Execute a bus trigger, regardless of trigger state. Can be used when *trigger_source* is set to *BUS*. Measurement result must be retrieved with *FETCH?* command.

trigger_initiate()

Change the trigger state from IDLE to WAIT FOR TRIGGER for one trigger sequence.

property trigger_source

Control trigger mode. Valid options are *INT*, *EXT*, *BUS*, or *HOLD*.

class pymeasure.instruments.agilent.agilent4284A.**Agilent4284ACorrection**(*parent*, *id*)

Bases: *Channel*

A class representing the correction functions.

property cable_length

Control the correction setting for cable length in meters, (int strictly 0, 1, 2 or 4).

property load_enabled

Control the LOAD correction (bool).

property load_function

Control the spot LOAD standard.

Type
str, strictly in CPD, CPQ, CPG, CPRP, CSD, CSQ, CSRS, LPQ, LPD, LPG, LPRP, LSD, LSQ, LSRS, RX, ZTD, ZTR, GB, YTD, YTR

See *Agilent4284A.impedance_mode* for detailed explanation.

measure_load()

Measure the LOAD correction standard.

measure_open()

Measure the OPEN correction standard.

measure_short()

Measure the SHORT correction standard.

property open_enabled

Control the OPEN correction (bool).

property short_enabled

Control the SHORT correction (bool).

class pymeasure.instruments.agilent.agilent4284A.**Agilent4284ASpot**(*parent*, *id*)

Bases: *Channel*

A class representing the spot correction functions.

property enabled

Control this spot correction (bool).

property frequency

Control the frequency of this spot in Hz.

property load_function

Control the LOAD standard of this spot.

Type
str, strictly in CPD, CPQ, CPG, CPRP, CSD, CSQ, CSRS, LPQ, LPD, LPG, LPRP, LSD, LSQ, LSRS, RX, ZTD, ZTR, GB, YTD, YTR

See *Agilent4284A.impedance_mode* for detailed explanation.

measure_load()

Measure the LOAD correction standard of this spot.

measure_open()

Measure the OPEN correction standard of this spot.

measure_short()

Measure the SHORT correction standard of this spot.

7.9.15 Agilent B2980 Series, Femto/Picoammeter Electrometer/High Resistance Meter

class pymeasure.instruments.agilent.**AgilentB2981**(*adapter*, *name='Agilent/Keysight B2980A/B series'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

A class representing the Agilent/Keysight B2981A/B.

The B2981 is a Femto/Picoammeter.

abort()

Abort the all actions.

abort_acquisition()

Abort action 'ACQuire'.

arm()

Send an immediate arm trigger for all actions.

When the status of all actions is initiated, the arm trigger causes a layer change from arm to trigger.

arm_acquisition()

Send an immediate arm trigger for action 'ACQuire'.

When the status of action 'ACQuire' is initiated, the arm trigger causes a layer change from arm to trigger.

property arm_acquisition_bypass_once_enabled

Control the bypass for the event detector in the arm layer for action 'ACQuire' (bool).

property arm_acquisition_count

Control the arm trigger count for action 'ACQuire'.

Type

- int, strictly from `1` to `100000` or
- str, strictly in `MIN`, `MAX`, `DEF`, `INF`

`INF` is equivalent to `2147483647`

property arm_acquisition_delay

Control the arm trigger delay for action 'ACQuire' in seconds.

Type

- float, strictly from `0` to `1E5` or
- str, strictly in `MIN`, `MAX`, `DEF`

property arm_acquisition_output_enabled

Control the arm trigger output for action 'ACQuire' (bool).

It is for the status change between the idle state and the arm layer.

property arm_acquisition_output_signal

Control the arm trigger output for action 'ACQuire'.

Type
str, strictly in INT1, INT2, LAN, TOUT, EXT1 to EXT7

- INT1 or INT2 selects the internal bus 1 or 2.
- LAN selects a LAN port.
- EXT1 to EXT7 selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TOUT selects the BNC Trigger Out.

It is for the status change between the idle state and the arm layer.

property arm_acquisition_source

Control the arm trigger source for action 'ACQuire'.

Type
str, strictly in AINT, BUS, TIM, INT1, INT2, LAN, TIN, EXT1 to EXT7

- AINT automatically selects the arm source most suitable for the present operating mode by using internal algorithms.
- BUS selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- TIM selects a signal internally generated every interval set by the arm_timer command.
- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by the arm_source_lan_id command.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property arm_acquisition_source_lan_id

Control the source for LAN arm triggers for action 'ACQuire'.

Type
str, strictly from LAN0 to LAN7

property arm_acquisition_timer

Control the timer interval of the arm trigger source for action 'ACQuire' in seconds.

Type

- float, strictly from 1E-5 to 1E5
- str, strictly in MIN, MAX, DEF

property arm_all_bypass_once_enabled

Set the bypass for the event detector in the arm layer for action 'ALL' (bool).

property arm_all_count

Set the arm trigger counter for action 'ALL'.

Type

- int, strictly from 1 to 100000 or
- str, strictly in MIN, MAX, DEF, INF

INF is equivalent to 2147483647.

property arm_all_delay

Set the arm trigger delay for action 'ALL' in seconds.

Type

- float, strictly from 0 to 1E5 or
- str, strictly in MIN, MAX, DEF

property arm_all_output_enabled

Set the arm trigger output for action 'ALL' (bool).

It is for the status change between the idle state and the arm layer.

property arm_all_output_signal

Set the arm trigger output for action 'ALL'.

Type
str, strictly in INT1, INT2, LAN, TOUT, EXT1 to EXT7

- INT1 and INT2 selects the internal bus 1 or 2.
- LAN selects a LAN port.
- EXT1 to EXT7 selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TOUT selects the BNC Trigger Out.

It is for the status change between the idle state and the arm layer.

property arm_all_source

Set the arm trigger source for action 'ALL'.

Type
str, strictly in AINT, BUS, TIM, INT1, INT2, LAN, TIN, EXT1 to EXT7

- AINT automatically selects the arm source most suitable for the present operating mode by using internal algorithms.
- BUS selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- TIM selects a signal internally generated every interval set by attr:*arm_all_timer*.
- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by *arm_all_source_lan_id*.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property arm_all_source_lan_id

Set the source for LAN arm triggers for action 'ALL'.

Type
str, strictly from LAN0 to LAN7

property arm_all_timer

Set the timer interval of the arm trigger source for action 'ALL' in seconds.

Type

- float, strictly from 1E-5 to 1E5 or
- str, strictly in MIN, MAX, DEF

property current

Measure current with a spot measurement in Amps.

Returns
float

property current_range

Control the range for the current measurement in Amps.

Type

- float, strictly from 2E-12 to 20E-3 or
- str, strictly in MIN, MAX, DEF, UP, DOWN

property data_buffer_size

Get the data buffer size (int).

init()

Initiate a trigger for all actions.

init_acquisition()

Initiate a trigger for action 'ACQuire'.

property input_enabled

Control the instrument input relay (bool).

property interlock_enabled

Get the interlock status (bool).

property measure

Measure the defined parameter(s) with a spot measurement.

Returns
float or list of float

property trigger_acquisition_bypass_once_enabled

Control the bypass for the event detector in the trigger layer for action 'ACQuire' (bool).

property trigger_acquisition_count

Control the trigger count for action 'ACQuire'.

Type

- int, strictly from 1 to 100000 or
- str, strictly in MIN, MAX, DEF, INF

INF is equivalent to 2147483647.

property trigger_acquisition_delay

Control the trigger delay for action 'ACQuire' in seconds.

Type

- float, strictly from 0 to 1E5 or
- str, strictly in MIN, MAX, DEF

property trigger_acquisition_is_idle

Get the idle status of action 'ACQuire' (bool).

The command waits until the status is changed to idle.

property trigger_acquisition_output_enabled

Control the trigger output for action 'ACQuire' (bool).

It is for the status change between the idle state and the trigger layer.

property trigger_acquisition_output_signal

Control the trigger signal output for action 'ACQuire'.

Type
str, strictly in INT1, INT2, LAN, TOUT, EXT1 to EXT7

- INT1 and INT2 selects the internal bus 1 or 2.
- LAN selects a LAN port.
- EXT1 to EXT7 selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TOUT: selects the BNC Trigger Out.

It is for the status change between the idle state and the trigger layer.

property trigger_acquisition_source

Control the trigger source for action 'ACQuire'.

Type
str, strictly in AINT, BUS, TIM, INT1, INT2, LAN, TIN, EXT1 to EXT7

- AINT automatically selects the trigger source most suitable for the present operating mode by using internal algorithms.
- BUS selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- TIM selects a signal internally generated every interval set by the *trigger_acquisition_timer*.
- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by *trigger_acquisition_source_lan_id*.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property trigger_acquisition_source_lan_id

Control the source for LAN triggers for action 'ACQuire'.

Type
str, strictly from LAN0 to LAN7

property trigger_acquisition_timer

Control the timer interval of the trigger source for action 'ACQuire' in seconds.

Type

- float, strictly from 1E-5 to 1E5 or
- str, strictly in MIN, MAX, DEF

property trigger_all_bypass_once_enabled

Set the bypass for the event detector in the trigger layer for action 'ALL' (bool).

property trigger_all_count

Set the trigger count for action 'ALL'.

Type

- int, strictly from 1 to 100000 or
- str, strictly in MIN, MAX, DEF, INF

INF is equivalent to 2147483647.

property trigger_all_delay

Set the trigger delay for action 'ALL' in seconds.

Type

- float, strictly from 0 to 1E5 or
- str, strictly in MIN, MAX, DEF

property trigger_all_is_idle

Get the trigger idle status for action 'ALL' (bool).

The command waits until the status is changed to idle.

property trigger_all_output_enabled

Set the trigger output for action 'ALL' (bool).

It is for the status change between the idle state and the trigger layer.

property trigger_all_output_signal

Set the trigger signal output for action 'ALL'.

Type
str, strictly in INT1, INT2, LAN, TOUT, EXT1 to EXT7

- INT1 and INT2 selects the internal bus 1 or 2.
- LAN selects a LAN port.
- EXT1 to EXT7 selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TOUT selects the BNC Trigger Out.

It is for the status change between the idle state and the trigger layer.

property trigger_all_source

Set the trigger source for action 'ALL'.

Type
str, strictly in AINT, BUS, TIM, INT1, INT2, LAN, TIN, EXT1 to EXT7

- AINT automatically selects the trigger source most suitable for the present operating mode by using internal algorithms.
- BUS selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- TIM selects a signal internally generated every interval set by *trigger_all_timer*.
- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by *trigger_all_source_lan_id*.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property trigger_all_source_lan_id

Set the source for LAN triggers for action 'ALL'.

Type
str, strictly from LAN0 to LAN7

property trigger_all_timer

Set the timer interval of the trigger source for action 'ALL' in seconds.

Type

- float, strictly from 1E-5 to 1E5 or
- str, strictly in MIN, MAX, DEF

property zero_corrected

Control the zero correction function (bool).

class pymeasure.instruments.agilent.**AgilentB2983**(*adapter*, *name='Agilent/Keysight B2980A/B series'*, ***kwargs*)

Bases: *AgilentB2981*, *BatteryMixin*

A class representing the Agilent/Keysight B2983A/B.

The B2983 is a Femto/Picoammeterwith with battery operation.

class pymeasure.instruments.agilent.**AgilentB2985**(*adapter*, *name='Agilent/Keysight B2980A/B series'*, ***kwargs*)

Bases: *AgilentB2981*

A class representing the Agilent/Keysight B2985A/B.

The B2985 is a Femto/Picoammeter and Electrometer/High Resistance Meter.

abort_transient()

Abort action 'TRANSient'.

`arm_transient()`

Send an immediate arm trigger for action 'TRANSient'.

When the status of the specified action is initiated, the arm trigger causes a layer change from arm to trigger.

`property arm_transient_bypass_once_enabled`

Control the bypass for the event detector in the arm layer for action 'TRANSient' (bool).

`property arm_transient_count`

Control the arm trigger count for action 'TRANSient'.

Type

- int, strictly from `1` to `100000` or
- str, strictly in `MIN`, `MAX`, `DEF`, `INF`

`INF` is equivalent to `2147483647`.

`property arm_transient_delay`

Control the arm trigger delay for action 'TRANSient' in seconds.

Type

- float, strictly from `0` to `1E5` or
- str, strictly in `MIN`, `MAX`, `DEF`

`property arm_transient_output_enabled`

Control the arm trigger output for action 'TRANSient' (bool).

It is for the status change between the idle state and the arm layer.

`property arm_transient_output_signal`

Control the arm trigger output for action 'TRANSient'.

Type
str, strictly in `INT1`, `INT2`, `LAN`, `TOUT`, `EXT1` to `EXT7`

- `INT1` and `INT2` selects the internal bus 1 or 2.
- `LAN` selects a LAN port.
- `EXT1` to `EXT7` selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- `TOUT` selects the BNC Trigger Out.

It is for the status change between the idle state and the arm layer.

`property arm_transient_source`

Control the arm trigger source for action 'TRANSient'.

Type
str, strictly in `AINT`, `BUS`, `TIM`, `INT1`, `INT2`, `LAN`, `TIN`, `EXT1` to `EXT7`

- `AINT` automatically selects the arm source most suitable for the present operating mode by using internal algorithms.
- `BUS` selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- `TIM` selects a signal internally generated every interval set by *arm_transient_timer*.

- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by *arm_transient_source_lan_id*.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property arm_transient_source_lan_id

Control the source for LAN arm triggers for action 'TRANSient'.

Type
str, strictly from LAN0 to LAN7

property arm_transient_timer

Control the timer interval of the arm trigger source for action 'TRANSient' in seconds.

Type

- float, strictly from 1E-5 to 1E5
- str, strictly in MIN, MAX, DEF

property charge

Measure charge with a spot measurement in Coulombs.

Returns
float

property charge_range

Control the range for the charge measurement in Coulombs.

Type

- float, strictly from 2E-9 to 2E-6 or
- str, strictly in MIN, MAX, DEF, UP, DOWN

property function

Control the measurement function.

Type
str, strictly in CURR, CHAR, VOLT, RES

property humidity

Measure the relative humidity in percent.

Returns
float

init_transient()

Initiate a trigger for action 'TRANSient'.

property resistance

Measure resistance with a spot measurement in Ohms.

Returns
float

property resistance_range

Control the range for the resistance measurement.

Type

- float, strictly from 1E6 to 1E15 or
- str, strictly in MIN, MAX, DEF, UP, DOWN

property source_enabled

Control the voltage source output relay (bool).

property source_low_state

Control the voltage source low terminal state.

Type
str, strictly FLO or COMM

- FLO: low terminal is floating.
- COMM: low terminal is connected to Common.

property source_off_state

Control the voltage source off condition.

Type
str, strictly in ZERO, HIZ, NORM

HIGH Z	• Output relay: off (open) • The voltage source setting is not changed. • This status is available only when the 20 V range is used.
NORMAL	• Output voltage: 0 V • Output relay: off (open)
ZERO	• Output voltage: 0 V in the present voltage range

property source_voltage

Control the source voltage in Volts.

Type
float, strictly from -1050 to 1050

For voltages > 20V the interlock must be closed.

Range	Resolution	Output voltage	Maximum current
20 V	700 µV	-21 V V 21 V	+/-20 mA
1000 V	35 mV	-1 V V 1000 V	+/-1.0 mA
-1000 V	35 mV	-1000 V V 1 V	+/-1.0 mA

property source_voltage_range

Control the source voltage range.

Type

- float, strictly from -1000 to 1000 or
- str, strictly in MIN, MAX, DEF

property temperature

Measure the temperature.

Returns
float

The unit of temperature is controlled by *temperature_unit*:

property temperature_sensor

Control the tempertature sensor.

Type
str, strictly in TC, HSEN

- TC selects the thermocouple.
- HSEN selects the temperature sensor within the humidity sensor.

property temperature_unit

Control the tempertature unit.

Type
str, strictly in C, CEL, F, FAR, K

- C, CEL selects degree Celsius.
- F, FAR selects degree Fahrenheit.
- K selects Kelvin.

property trigger_transient_bypass_once_enabled

Control the bypass for the event detector in trigger layer for action 'TRANSient' (bool).

property trigger_transient_count

Control the trigger count for action 'TRANSient'.

Type

- int, strictly from 1 to 100000 or
- str, strictly in MIN, MAX, DEF, INF

INF is equivalent to 2147483647.

property trigger_transient_delay

Control the trigger delay for action 'TRANSient' in seconds.

Type

- float, strictly from 0 to 1E5 or
- str, strictly in MIN, MAX, DEF

property trigger_transient_is_idle

Get idle the status of action 'TRANSient' (bool).

The command waits until the status is changed to idle.

property trigger_transient_output_enabled

Control the trigger output for action 'TRANSient' (bool).

It is for the status change between the idle state and the trigger layer.

property trigger_transient_output_signal

Control the trigger signal output for action 'TRANSient'.

Type
str, strictly in INT1, INT2, LAN, TOUT, EXT1 to EXT7

- INT1, INT2 selects the internal bus 1 or 2.
- LAN selects a LAN port.
- EXT1 to EXT7 selects the GPIO pin n, which is an output port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TOUT selects the BNC Trigger Out.

It is for the status change between the idle state and the trigger layer.

property trigger_transient_source

Control the trigger source for action 'TRANSient'.

Type
str, strictly in AINT, BUS, TIM, INT1, INT2, LAN, TIN, EXT1 to EXT7

- AINT automatically selects the trigger source most suitable for the present operating mode by using internal algorithms.
- BUS selects the remote interface trigger command such as the group execute trigger (GET) and the TRG command.
- TIM selects a signal internally generated every interval set by *trigger_transient_timer*.
- INT1 and INT2 selects a signal from the internal bus 1 or 2.
- LAN selects the LXI trigger specified by *trigger_transient_source_lan_id*.
- EXT1 to EXT7 selects a signal from the GPIO pin n, which is an input port of the Digital I/O D-sub connector on the rear panel. n = 1 to 7.
- TIN selects the BNC Trigger In.

property trigger_transient_source_lan_id

Control the source for LAN triggers for action 'TRANSient'.

Type
str, strictly from LAN0 to LAN7

property trigger_transient_timer

Control the timer interval of the trigger source for action 'TRANSient'.

Type

- float, strictly from 1E-5 to 1E5 or

- str, strictly in `MIN`, `MAX`, `DEF`

property voltage

Measure voltage with a spot (one-shot) measurement in Volts.

Returns
float

property voltage_range

Control the range for the voltage measurement in Volts.

Type

- float, strictly from `2` to `20` or
- str, strictly in `MIN`, `MAX`, `DEF`, `UP`, `DOWN`

class pymeasure.instruments.agilent.**AgilentB2987**(*adapter*, *name='Agilent/Keysight B2980A/B series'*, ***kwargs*)

Bases: *AgilentB2985*, *BatteryMixin*

A class representing the Agilent/Keysight B2987A/B.

The B2987 is a Femto/Picoammeter and Electrometer/High Resistance Meter with battery operation.

class pymeasure.instruments.agilent.agilentB298x.**BatteryMixin**

Bases: `object`

A class representing the B2983/7 battery functions.

property battery_cycles

Get the battery cycle count (int).

property battery_level

Get the percentage of the remaining battery capacity (int).

property battery_selftest_passed

Get the battery self-test result (bool).

7.9.16 Agilent E5270B SMU mainframe

class pymeasure.instruments.agilent.**AgilentE5270B**(*adapter*, *name='Agilent E5270B'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

A class representing the Agilent E5270B 8 slot SMU mainframe.

It supports the following plug-in modules:

Name	Type	Slots	Range	Resolution
E5280B	HPSMU	2	±200 V, ±1 A	2 µV, 10 fA
E5281B	MPSMU	1	±100 V, ±100 mA	0.5 µV, 10 fA
E5287A	HRSMU	1	±100 V, ±100 mA	0.5 µV, 1 fA
E5288A	ASU	—	±100 V, ±100 mA	0.5 µV, 0.1 fA

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

`get_error_message`(*error_code*)

Return the error message for to the specified error code (str).

`property options`

Get the installed SMUs (list of str).

Each list entry contains the module name and its revision, e.g. `E5281B,0` An empty slot returns `0,0`

`class pymeasure.instruments.agilent.agilentE5270B.SMUChannel`(*parent*, *id*)

Bases: *Channel*

A class representing the Agilent E5270B SMU channel.

`property current`

Measure the current in Amps (float).

`property current_setpoint`

Set range, output current and voltage compliance (int, float, float).

```
inst = AgilentE5270B("GPIB0::17::INSTR")
inst.smu1.current_setpoint = (0, 1.32e-3, 0.5)  # (range, value, compliance)
# Set SMU1 to output 1.32 mA in autorange with 0.5 V voltage compliance
```

Ranges:

- `0`: Auto ranging
- `8`: 1 pA limited auto ranging for E5287A+E5288A
- `9`: 10 pA limited auto ranging for E5287A
- `10`: 100 pA limited auto ranging for E5287A
- `11`: 1 nA limited auto ranging
- `12`: 10 nA limited auto ranging
- `13`: 100 nA limited auto ranging
- `14`: 1 A limited auto ranging
- `15`: 10 A limited auto ranging
- `16`: 100 A limited auto ranging
- `17`: 1 mA limited auto ranging
- `18`: 10 mA limited auto ranging
- `19`: 100 mA limited auto ranging
- `20`: 1 A limited auto ranging for E5280B

`property enabled`

Set the channel output state (bool).

`property voltage`

Measure the voltage in Volts (float).

property voltage_setpoint

Set range, output voltage and current compliance (int, float, float)

```
inst = AgilentE5270B("GPIB0::17::INSTR")
inst.smu1.voltage_setpoint = (12, 14.42, 0.05)  # (range, value, compliance)
# Set SMU1 to output 14.42 V in 20 V range with 50 mA current compliance
```

Ranges:

- `0`: Auto ranging
- `5`: 0.5 V limited auto ranging for E5281B/E5287A
- `50`: 5 V limited auto ranging for E5281B/E5287A
- `11` or `20`: 2 V limited auto ranging
- `12` or `200`: 20 V limited auto ranging
- `13` or `400`: 40 V limited auto ranging
- `14` or `1000`: 100 V limited auto ranging
- `15` or `2000`: 200 V limited auto ranging for E5280B

7.10 Aim-TTI

This section contains specific documentation on the Aim-TTI instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.10.1 Aim-TTI PL Series Power Supplies

class pymeasure.instruments.aimtti.aimttiPL.**PLBase**(*adapter*, *name='AimTTI PL'*, ***kwargs*)

Bases: `SCPIUnknownMixin`, *Instrument*

Control AimTTI PL series power supplies. Model number ending with -P or P(G) support this remote interface.

Documentation:
https://resources.aimtti.com/manuals/New_PL+PL-P_Series_Instruction_Manual-Iss18.pdf

PL-series devices:
https://www.aimtti.com/product-category/dc-power-supplies/aim-plseries

The default value for the timeout argument is set to 5000ms.

```
psu = PL303QMDP("ASRL7::INSTR")
psu.reset()
psu.ch_2.voltage = 1.2
psu.ch_2.output_enabled = True
...
psu.ch_2.output_enabled = False
psu.local()
```

property all_outputs_enabled

Control whether all sources are enabled simultaneously, takes values True or False.

local()

Go to local. Make sure all output are disabled first.

class pymeasure.instruments.aimtti.aimttiPL.**PLChannel**(*parent*, *id*, *voltage_range: list = None*, *current_range: list = None*)

Bases: *Channel*

A channel of AimTTI PL series power supplies.

Channels of the power supply. The channels are number from right-to-left, starting at 1.

property current

Measure the output readback current for this output channel in Amps.

property current_limit

Control the current limit in Amps.(dynamic)

property current_range

Control the current range of the channel. Low (500/800mA) range, or High range. Output must be switched off before changing range.

property output_enabled

Control whether the source is enabled, takes values True or False.

property voltage

Measure the output readback voltage for this output channel in Volts.

property voltage_setpoint

Control the output voltage of this channel. With verify: the operation is completed when the parameter being adjusted reaches the required value to within ±5% or ±10 counts.(dynamic)

class pymeasure.instruments.aimtti.aimttiPL.**PL068P**(*adapter*, *name='AimTTI PL068-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

class pymeasure.instruments.aimtti.aimttiPL.**PL155P**(*adapter*, *name='AimTTI PL155-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

class pymeasure.instruments.aimtti.aimttiPL.**PL303P**(*adapter*, *name='AimTTI PL303-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

class pymeasure.instruments.aimtti.aimttiPL.**PL601P**(*adapter*, *name='AimTTI PL601-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

class pymeasure.instruments.aimtti.aimttiPL.**PL303QMDP**(*adapter*, *name='AimTTI PL303QMD-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

ch_2

Channel
PLChannel

class pymeasure.instruments.aimtti.aimttiPL.**PL303QMTP**(*adapter*, *name='AimTTI PL303QMT-P'*, ***kwargs*)

Bases: *PLBase*

ch_1

Channel
PLChannel

ch_2

Channel
PLChannel

ch_3

Channel
PLChannel

7.10.2 Aim-TTi Electronic Loads

class pymeasure.instruments.aimtti.ld400p.**LD400P**(*adapter*, *name='LD400'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Interface for the LD400P electronic DC load from Aim-TTi.

Note: the LD400 edition doesn't have digital interfacing, only the LD400P does.

The interface is described by the LD400[P] manual.

The class `SCPIMixin` is inherited, however the commands *options*, next_error` and *check_errors* do not work for this device.

check_errors()

Not available.

property current

Get the measured load current (float).

property input_enabled

Control the input state of the digital load, can be `True` (on) or `False` (off).

property level_a

Control the A level of the current control mode (float).

property level_b

Control the B level of the current control mode (float).

property level_select

Control the selected level of the digital load. Can be one of: `("A", "B", "T", "V", "E")`, corresponding to (respectively) Level A, Level B, Transient, Ext Voltage and Ext TTL (string).

property mode

Control the mode of the digital load. Can be one of: `("C", "P", "R", "G", "V")`, corresponding to (respectively) constant current, power, resistance, conductance, or voltage (string).

property next_error

Get next error - Not available.

property options

Get options - Not available.

classmethod remove_unit_suffix(*msg: str*) → str

Remove known unit suffixes from a msg.

The units of some replies depend on the mode of the device, hence all known units are removed.

property voltage

Get the measured source input voltage (float).

7.11 AJA International

This section contains specific documentation on the AJA International instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.11.1 AJA DCXS-750 or 1500 DC magnetron sputtering power supply

class pymeasure.instruments.aja.**DCXS**(*adapter*, *name='AJA DCXS sputtering power supply'*, ***kwargs*)

Bases: *Instrument*

AJA DCXS-750 or 1500 DC magnetron sputtering power supply with multiple outputs

Connection to the device is made through an RS232 serial connection. The communication settings are fixed in the device at 38400, one stopbit, no parity. The device's communication protocol uses single character commands and fixed length replies, both without any terminator.

Parameters

- **adapter** – pyvisa resource name of the instrument or adapter instance
- **name** (*string*) – The name of the instrument.
- **kwargs** – Any valid key-word argument for Instrument

property active_gun

Control the active gun number.

ask(*command*, *query_delay=None*, ***kwargs*)

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.
- ****kwargs** – Keyword arguments passed to the read method.

Returns

String returned by the device without read_termination.

property current

Measure the output current in mA.

property deposition_time_min

Control the minutes part of deposition time. Can be set only when 'enabled' is False.

property deposition_time_sec

Control the seconds part of deposition time. Can be set only when 'enabled' is False.

property enabled

Control the on/off state of the power supply

property fault_code

Get the error code from the power supply.

property id

Get the power supply type identifier.

property material

Control the material name of the sputter target.

property power

Measure the actual output power in W.

property ramp_time

Control the ramp time in seconds. Can be set only when 'enabled' is False.

read(*reply_length=-1*, ***kwargs*)

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

property regulation_mode

Control the regulation mode of the power supply.

property remaining_deposition_time_min

Get the minutes part of remaining deposition time.

property remaining_deposition_time_sec

Get the seconds part of remaining deposition time.

property setpoint

Control the setpoint value. Units are determined by regulation mode (power -> W, voltage -> V, current -> mA).

property shutter_delay

Control the shutter delay in seconds. Can be set only when 'enabled' is False.

property shutter_state

Get the status of the gun shutters. 0 for closed and 1 for open shutters.

property software_version

Get the software revision of the power supply firmware.

property voltage

Measure the output voltage in V.

7.12 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*.

7.12.1 Ametek 7270 DSP Lockin Amplifier

class pymeasure.instruments.ametek.**Ametek7270**(*adapter*, *name='Ametek DSP 7270'*, *read_termination='\x00'*, *write_termination='\x00'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

This is the class for the Ametek DSP 7270 lockin amplifier

In this instrument, some measurements are defined only for specific modes, called Reference modes, see *set_reference_mode()* and will raise errors if called incorrectly

property adc1

Get the input value of ADC1 in Volts (float).

property adc2

Get the input value of ADC2 in Volts (float).

property adc3

Get the input value of ADC3 in Volts (float).

property adc4

Get the input value of ADC4 in Volts (float).

ask(*command*, *query_delay=None*)

Send a command and read the response, stripping white spaces.

Usually the properties use the *values()* method that adds a strip call, however several methods use directly the result from ask to be cast into some other types. It should therefore also add the strip here, as all responses end with a newline character.

property auto_gain

Control whether automatic gain is enabled (bool).

check_set_errors()

mandatory to be used for property setter

The Ametek protocol expect the default null character to be read to check the property has been correctly set. With default termination character set as Null character, this turns out as an empty string to be read.

property dac1

Control the output value on DAC1 in Volts (float truncated to values from -10 to 10).

property dac2

Control the output value on DAC2 in Volts (float truncated to values from -10 to 10).

property dac3

Control the output value on DAC3 in Volts (float truncated to values from -10 to 10).

property dac4

Control the output value on DAC4 in Volts (float truncated to values from -10 to 10).

property frequency

Control the lock-in frequency in Hz (float, truncated_range, values from 0 to 2.5e5).

property harmonic

Control the reference harmonic mode (integer truncated to values from 1 to 127).

property id

Get the instrument ID and firmware version

property mag

Get magnitude in Volts (float).

property phase

Control the reference harmonic phase in degrees (float, modular_range, values from 0 to 360).

property sensitivity

Control the sensitivity range in Volts (float truncated to values from 2 nV to 1 V).(dynamic)

set_channel_A_mode()

Sets instrument to channel A mode – assuming it is in voltage mode

set_current_mode(*low_noise=False*)

Sets instrument to current control mode with either low noise or high bandwidth

set_differential_mode(*lineFiltering=True*)

Sets instrument to differential mode – assuming it is in voltage mode

set_reference_mode(*mode: int = 0*)

Set the instrument in Single, Dual or harmonic mode.

Parameters

mode – the integer specifying the mode: 0 for Single, 1 for Dual harmonic, and 2 for Dual reference.

set_voltage_mode()

Sets instrument to voltage control mode

shutdown()

Ensures the instrument in a safe state

property slope

Control the filter slope in dB/octave (integer truncated to values 6, 12, 18, 24).

property theta

Get signal phase in degrees (float).

property time_constant

Control the time constant in seconds (float truncated to values from 10 microseconds to 100,000 seconds).

property voltage

Control the voltage in Volts (float, truncated_range, values from 0 to 5).

property x

Get X value in Volts (float).

property x1

Get first harmonic X value in Volts (float).

property x2

Get second harmonic X value in Volts (float).

property xy

Get both X and Y values in Volts (float, tuple).

property y

Get Y value in Volts (float).

property y1

Get first harmonic Y value in Volts (float).

property y2

Get second harmonic Y value in Volts (float).

7.13 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*.

7.13.1 AMI 430 Power Supply

class pymeasure.instruments.ami.**AMI430**(*adapter*, *name='AMI superconducting magnet power supply.'*, ***kwargs*)

Bases: *SCPIMixin*, *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
```

(continues on next page)

(continued from previous page)

```
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

Control the coil constant in kGauss/A. (float)

disable_persistent_switch()

Disables the persistent switch.

enable_persistent_switch()

Enables the persistent switch.

property field

Get the field in kGauss of the magnet.

has_persistent_switch_enabled()

Returns a boolean if the persistent switch is enabled.

property magnet_current

Get the current in Amps of the magnet.

property magnet_status

Get the magnet status.

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

Control the current ramping rate in A/s. (float)

property ramp_rate_field

Control the field ramping rate in kGauss/s. (float)

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

Get the field in kGauss of the magnet.

property supply_current

Get the current in Amps of the power supply.

property target_current

Control the target current in A for the magnet. (float)

property target_field

Control the target field in kGauss for the magnet. (float)

property voltage_limit

Control the voltage limit for charging/discharging the magnet. (float)

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.

7.14 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*.

7.14.1 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*, *name='Anaheim Automation Stepper Motor Controller'*, *address=0*, *encoder_enabled=False*, ***kwargs*)

Bases: *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.

7.15 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*.

7.15.1 Anapico APSIN12G Signal Generator

class pymeasure.instruments.anapico.**APSIN12G**(*adapter*, *name='Anapico APSIN12G Signal Generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Anapico APSIN12G Signal Generator with option 9K, HP and GPIB.

property blanking

Control the blanking of output power when frequency is changed. ON makes the output to be blanked (off) while changing frequency.

disable_rf()

Disables the RF output.

enable_rf()

Enables the RF output.

property frequency

Control the output frequency in Hz. (float)

property power

Control the output power in dBm. (float)

property reference_output

Control the 10MHz reference output from the synth. (str)

7.16 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*.

7.16.1 Andeen Hagerling AH2500A capacitance bridge

class pymeasure.instruments.andeenhagerling.**AH2500A**(*adapter*, *name=None*, *timeout=3000*, *write_termination='\n'*, *read_termination='\n'*, ***kwargs*)

Bases: *Instrument*

Andeen Hagerling 2500A Precision Capacitance Bridge implementation

property caplossvolt

Get the result of 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

Get the 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

Control 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.

7.16.2 Andeen Hagerling AH2700A capacitance bridge

class pymeasure.instruments.andeenhagerling.**AH2700A**(*adapter*, *name='Andeen Hagerling 2700A Precision Capacitance Bridge'*, *timeout=5000*, ***kwargs*)

Bases: *AH2500A*

Andeen Hagerling 2700A Precision Capacitance Bridge implementation

property caplossvolt

Get the result of a single capacitance, loss measurement and return the values in units of pF and nS. The used measurement voltage is returned as third value.

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clears the instrument status byte

property complete

Get the synchronization bit.

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 config

Get the configuration

property frequency

Control 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

Get the instrument identification

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get 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

Get the status byte and Master Summary Status bit.

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

Control 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.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.17 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*.

7.17.1 Anritsu MG3692C Signal Generator

class pymeasure.instruments.anritsu.**AnritsuMG3692C**(*adapter*, *name='Anritsu MG3692C Signal Generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Anritsu MG3692C Signal Generator

disable()

Disables the signal output.

enable()

Enables the signal output.

property frequency

Control the output frequency in Hz. This property can be set. (float)

property output

Control the signal output state. (bool)

property power

Control the output power in dBm. (float))

shutdown()

Shuts down the instrument, putting it in a safe state.

7.17.2 Anritsu MS9710C Optical Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS9710C**(*adapter*, *name='Anritsu MS9710C Optical Spectrum Analyzer'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Anritsu MS9710C Optical Spectrum Analyzer.

property analysis
Control Analysis Control

property analysis_result
Get anaysis result from current scan.

property average_point
Control number of averages to take on each point (2-1000), or OFF

property average_sweep
Control 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
Get the data condition of data memory register A. Starting wavelength, and a sampling point (l1, l2, n).

property data_memory_a_size
Get the number of points sampled in data memory register A.

property data_memory_a_values
Get the binary data from memory register A.

property data_memory_b_condition
Get the data condition of data memory register B. Starting wavelength, and a sampling point (l1, l2, n).

property data_memory_b_size
Get the number of points sampled in data memory register B.

property data_memory_b_values
Get the binary data from memory register B.

property data_memory_select
Control Memory Data Select.

property dip_search
Control Dip Search Mode

property ese2
Control Extended Event Status Enable Register 2

property esr2
Control Extended Event Status Register 2

property level_lin
Control Level Linear Scale (/div)

property level_log
Control Level Log Scale (/div)

property level_opt_attn
Control Optical Attenuation Status (ON/OFF)

property level_scale
Get Current Level Scale

property measure_mode
Get the current Measure Mode the OSA is in.

measure_peak()

Measure the peak and return the trace marker.

property peak_search

Control Peak Search Mode

read_memory(*slot='A'*)

Read the scan saved in a memory slot.

property resolution

Control Resolution (nm)

property resolution_actual

Control Resolution Actual (ON/OFF)

property resolution_vbw

Control Video Bandwidth Resolution

property sampling_points

Control number of sampling points

single_sweep(***kwargs*)

Perform a single sweep and wait for completion.

property trace_marker

Control the trace marker with a wavelength. Returns the trace wavelength and power.

property trace_marker_center

Set 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

Control Center Wavelength of Spectrum Scan in nm.

property wavelength_marker_value

Control Wavelength Marker Value (wavelength or freq.?)

property wavelength_span

Control Wavelength Span of Spectrum Scan in nm.

property wavelength_start

Control Wavelength Start of Spectrum Scan in nm.

property wavelength_stop

Control Wavelength Stop of Spectrum Scan in nm.

property wavelength_value_in

Control Wavelength value in Vacuum or Air

property wavelengths

Get a numpy array of the current wavelengths of scans.

7.17.3 Anritsu MS9740A Optical Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS9740A**(*adapter*, *name='Anritsu MS9740A Optical Spectrum Analyzer'*, ***kwargs*)

Bases: *AnritsuMS9710C*

Anritsu MS9740A Optical Spectrum Analyzer.

property average_sweep

Control number of averages to make on a sweep (1-1000), with 1 being a single (non-averaged) sweep

property data_memory_select

Control Memory Data Select.

repeat_sweep(*n=20*, *delay=0.5*)

Perform a single sweep and wait for completion.

property resolution

Control Resolution (nm)

property resolution_vbw

Control Video Bandwidth Resolution

property sampling_points

Control number of sampling points

7.17.4 Anritsu MS2090A Handheld Spectrum Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS2090A**(*adapter*, *name='Anritsu MS2090A Handheld Spectrum Analyzer'*, ***kwargs*)

Bases: *SCPIMixin*, *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.

7.17.5 Anritsu MS464xB Vector Network Analyzer

class pymeasure.instruments.anritsu.**AnritsuMS4642B**(*adapter*, *name='Anritsu MS464xB Vector Network Analyzer'*, *active_channels=16*, *installed_ports=4*, *traces_per_channel=None*, ***kwargs*)

Bases: *AnritsuMS464xB*

A class representing the Anritsu MS4642B Vector Network Analyzer (VNA).

This VNA has a frequency range from 10 MHz to 20 GHz and is part of the *AnritsuMS464xB* family of instruments; for documentation, for documentation refer to this base class.

class pymeasure.instruments.anritsu.**AnritsuMS4644B**(*adapter*, *name='Anritsu MS464xB Vector Network Analyzer'*, *active_channels=16*, *installed_ports=4*, *traces_per_channel=None*, ***kwargs*)

Bases: *AnritsuMS464xB*

A class representing the Anritsu MS4644B Vector Network Analyzer (VNA).

This VNA has a frequency range from 10 MHz to 40 GHz and is part of the *AnritsuMS464xB* family of instruments; for documentation, for documentation refer to this base class.

class pymeasure.instruments.anritsu.**AnritsuMS4645B**(*adapter*, *name='Anritsu MS464xB Vector Network Analyzer'*, *active_channels=16*, *installed_ports=4*, *traces_per_channel=None*, ***kwargs*)

Bases: *AnritsuMS464xB*

A class representing the Anritsu MS4645B Vector Network Analyzer (VNA).

This VNA has a frequency range from 10 MHz to 50 GHz and is part of the *AnritsuMS464xB* family of instruments; for documentation, for documentation refer to this base class.

class pymeasure.instruments.anritsu.**AnritsuMS4647B**(*adapter*, *name='Anritsu MS464xB Vector Network Analyzer'*, *active_channels=16*, *installed_ports=4*, *traces_per_channel=None*, ***kwargs*)

Bases: *AnritsuMS464xB*

A class representing the Anritsu MS4647B Vector Network Analyzer (VNA).

This VNA has a frequency range from 10 MHz to 70 GHz and is part of the *AnritsuMS464xB* family of instruments; for documentation, for documentation refer to this base class.

class pymeasure.instruments.anritsu.**AnritsuMS464xB**(*adapter*, *name='Anritsu MS464xB Vector Network Analyzer'*, *active_channels=16*, *installed_ports=4*, *traces_per_channel=None*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

A class representing the Anritsu MS464xB Vector Network Analyzer (VNA) series.

This family consists of the MS4642B, MS4644B, MS4645B, and MS4647B, which are represented in their respective classes (*AnritsuMS4642B*, *AnritsuMS4644B*, *AnritsuMS4645B*, *AnritsuMS4647B*), that only differ in the available frequency range.

They can contain up to 16 instances of *MeasurementChannel* (depending on the configuration of the instrument), that are accessible via the *channels* dict or directly via *ch_* + the channel number.

Parameters

- **active_channels** (*int (1-16) or str ("auto")*) – defines the number of active channels (default=16); if active_channels is "auto", the instrument will be queried for the number of active channels.
- **installed_ports** (*int (1-4) or str ("auto")*) – defines the number of installed ports (default=4); if "auto" is provided, the instrument will be queried for the number of ports
- **traces_per_channel** (*int (1-16) or str ("auto") or None*) – defines the number of traces that is assumed for each channel (between 1 and 16); if not provided, the maximum number is assumed; "auto" is provided, the instrument will be queried for the number of traces of each channel.

property active_channel

Control the active channel.

property bandwidth_enhancer_enabled

Control the state of the IF bandwidth enhancer.

property binary_data_byte_order

Control the binary numeric I/O data byte order.

valid values are:

value	description
NORM	The most significant byte (MSB) is first
SWAP	The least significant byte (LSB) is first

check_errors()

Read all errors from the instrument.

Returns
list of error entries

copy_data_file(*from_filename*, *to_filename*)

Copy a file on the VNA HDD.

Parameters

- **from_filename** (*str*) – full filename including pat
- **to_filename** (*str*) – full filename including path

create_directory(*dir_name*)

Create a directory on the VNA HDD.

Parameters
dir_name (*str*) – directory name

property data_drawing_enabled

Control whether data drawing is enabled (True) or not (False).

property datablock_header_format

Control the way the arbitrary block header for output data is formed.

Valid values are:

value	description
0	A block header with arbitrary length will be sent.
1	The block header will have a fixed length of 11 characters.
2	No block header will be sent. Not IEEE 488.2 compliant.

property datablock_numeric_format

Control format for numeric I/O data representation.

Valid values are:

value	description
ASCII	An ASCII number of 20 or 21 characters long with floating point notation.
8byte	8 bytes of binary floating point number representation limited to 64 bits.
4byte	4 bytes of floating point number representation.

property datafile_frequency_unit

Control the frequency unit displayed in a SNP data file.

Valid values are HZ, KHZ, MHZ, GHZ.

property datafile_include_heading

Control whether a heading is included in the data files.

property datafile_parameter_format

Control the parameter format displayed in an SNP data file.

Valid values are:

value	description
LINPH	Linear and Phase.
LOGPH	Log and Phase.
REIM	Real and Imaginary Numbers.

delete_data_file(*filename*)

Delete a file on the VNA HDD.

Parameters
filename (*str*) – full filename including path

delete_directory(*dir_name*)

Delete a directory on the VNA HDD.

Parameters
dir_name (*str*) – directory name

property display_layout

Control the channel display layout in a Row-by-Column format.

Valid values are: R1C1, R1C2, R2C1, R1C3, R3C1, R2C2C1, R2C1C2, C2R2R1, C2R1R2, R1C4, R4C1, R2C2, R2C3, R3C2, R2C4, R4C2, R3C3, R5C2, R2C5, R4C3, R3C4, R4C4. The number following the R indicates the number of rows, following the C the number of columns; e.g. R2C2 results in a 2-by-2 layout. The options that contain two C's or R's result in asymmetric layouts; e.g. R2C1C2 results in a layout with 1 channel on top and two channels side-by-side on the bottom row.

property event_status_enable_bits

Control the Standard Event Status Enable Register bits.

The register can be queried using the *query_event_status_register()* method. Valid values are between 0 and 255. Refer to the instrument manual for an explanation of the bits.

property external_trigger_delay

Control the delay time of the external trigger in seconds.

Valid values are between 0 [s] and 10 [s] in steps of 1e-9 [s] (i.e. 1 ns).

property external_trigger_edge

Control the edge type of the external trigger.

Valid values are POS (for positive or leading edge) or NEG (for negative or trailing edge).

property external_trigger_handshake

Control status of the external trigger handshake.

property external_trigger_type

Control the type of trigger that will be associated with the external trigger.

Valid values are POIN (for point), SWE (for sweep), CHAN (for channel), and ALL.

property hold_function_all_channels

Control the hold function of all channels.

Valid values are:

value	description
CONT	Perform continuous sweeps on all channels
HOLD	Hold the sweep on all channels
SING	Perform a single sweep and then hold all channels

load_data_file(*filename*)

Load a data file from the VNA HDD into the VNA memory.

Parameters
filename (*str*) – full filename including path

load_data_file_to_memory(*filename*)

Load a data file to a memory trace.

Parameters
filename (*str*) – full filename including path

property manual_trigger_type

Control the type of trigger that will be associated with the manual trigger.

Valid values are POIN (for point), SWE (for sweep), CHAN (for channel), and ALL.

property max_number_of_points

Control the maximum number of points the instrument can measure in a sweep.

Note that when this value is changed, the instrument will be rebooted. Valid values are 25000 and 100000. When 25000 points is selected, the instrument supports 16 channels with 16 traces each; when 100000 is selected, the instrument supports 1 channel with 16 traces.

property number_of_channels

Control the number of displayed (and therefore accessible) channels.

When the system is in 25000 points mode, the number of channels can be 1, 2, 3, 4, 6, 8, 9, 10, 12, or 16; when the system is in 100000 points mode, the system only supports 1 channel. If a value is provided that is not valid in the present mode, the instrument is set to the next higher channel number.

property number_of_ports

Get the number of instrument test ports.

query_event_status_register()

Query the value of the Standard Event Status Register.

Note that querying this value, clears the register. Refer to the instrument manual for an explanation of the returned value.

read_datafile(*channel*, *sweep_points*, *datafile_freq*, *datafile_par*, *filename*)

Read a data file from the VNA.

Parameters

- **channel** (*int*) – Channel Index
- **sweep_points** (*int*) – number of sweep point as an integer
- **datafile_freq** (*DataFileFrequencyUnits*) – Data file frequency unit
- **datafile_par** (*DataFileParameter*) – Data file parameter format
- **filename** (*str*) – full path of the file to be saved

property remote_trigger_type

Control the type of trigger that will be associated with the remote trigger.

Valid values are POIN (for point), SWE (for sweep), CHAN (for channel), and ALL.

return_to_local()

Returns the instrument to local operation.

property service_request_enable_bits

Control the Service Request Enable Register bits.

Valid values are between 0 and 255; setting 0 performs a register reset. Refer to the instrument manual for an explanation of the bits.

store_image(*filename*)

Capture a screenshot to the file specified.

Parameters

filename (*str*) – full filename including path

trigger()

Trigger a continuous sweep from the remote interface.

trigger_continuous()

Trigger a continuous sweep from the remote interface.

trigger_single()

Trigger a single sweep with synchronization from the remote interface.

property trigger_source

Control the source of the sweep/measurement triggering.

Valid values are:

value	description
AUTO	Automatic triggering
MAN	Manual triggering
EXTT	Triggering from rear panel BNC via the GPIB parser
EXT	External triggering port
REM	Remote triggering

update_channels(*number_of_channels=None*, ***kwargs*)

Create or remove channels to be correct with the actual number of channels.

Parameters

number_of_channels (*int*) – optional, if given, defines the desired number of channels.

class pymeasure.instruments.anritsu.anritsuMS464xB.**MeasurementChannel**(**args*, *frequency_range=None*, *traces=None*, ***kwargs*)

Bases: *Channel*

Represents a channel of Anritsu MS464xB VNA.

Contains 4 instances of *Port* (accessible via the *ports* dict or directly *pt_* + the port number) and up to 16 instances of *Trace* (accessible via the *traces* dict or directly *tr_* + the trace number).

Parameters

- **frequency_range** (*list of floats*) – defines the number of installed ports (default=4).
- **traces** (*int (1-16) or str ("auto") or None*) – defines the number of traces that is assumed for the channel (between 1 and 16); if not provided, the maximum number is assumed; "auto" is provided, the instrument will be queried for the number of traces.

activate()

Set the indicated channel as the active channel.

property active_trace

Set the active trace on the indicated channel.

property application_type

Control the application type of the specified channel.

Valid values are TRAN (for transmission/reflection), NFIG (for noise figure measurement), PULS (for PulseView).

property average_count

Control the averaging count for the indicated channel.

The channel must be turned on. Valid values are between 1 and 1024.

property average_sweep_count

Get the averaging sweep count for the indicated channel.

property average_type

Control the averaging type to for the indicated channel.

Valid values are POIN (point-by-point) or SWE (sweep-by-sweep)

property averaging_enabled

Control whether the averaging is turned on for the indicated channel.

property bandwidth

Control the IF bandwidth for the indicated channel.

Valid values are between 1 [Hz] and 1E6 [Hz] (i.e. 1 MHz). The system will automatically select the closest IF bandwidth from the available options (1, 3, 10 … 1E5, 3E5, 1E6).

property calibration_enabled

Control whether the RF correction (calibration) is enabled for indicated channel.

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

clear_average_count()

Clear and restart the averaging sweep count of the indicated channel.

property cw_mode_enabled

Control the state of the CW sweep mode of the indicated channel.

property cw_number_of_points

Control the CW sweep mode number of points of the indicated channel.

Valid values are between 1 and 25000 or 100000 depending on the maximum points setting.

property display_layout

Control the trace display layout in a Row-by-Column format for the indicated channel.

Valid values are: R1C1, R1C2, R2C1, R1C3, R3C1, R2C2C1, R2C1C2, C2R2R1, C2R1R2, R1C4, R4C1, R2C2, R2C3, R3C2, R2C4, R4C2, R3C3, R5C2, R2C5, R4C3, R3C4, R4C4. The number following the R indicates the number of rows, following the C the number of columns; e.g. R2C2 results in a 2-by-2 layout. The options that contain two C's or R's result in asymmetric layouts; e.g. R2C1C2 results in a layout with 1 trace on top and two traces side-by-side on the bottom row.

property frequency_CW

Control the CW frequency of the indicated channel in hertz.

Valid values are between 1E7 [Hz] (i.e. 10 MHz) and 4E10 [Hz] (i.e. 40 GHz). (dynamic)

property frequency_center

Control the center value of the sweep range of the indicated channel in hertz.

Valid values are between 1E7 [Hz] (i.e. 10 MHz) and 4E10 [Hz] (i.e. 40 GHz). (dynamic)

property frequency_span

Control the span value of the sweep range of the indicated channel in hertz.

Valid values are between 2 [Hz] and 4E10 [Hz] (i.e. 40 GHz). (dynamic)

property frequency_start

Control the start value of the sweep range of the indicated channel in hertz.

Valid values are between 1E7 [Hz] (i.e. 10 MHz) and 4E10 [Hz] (i.e. 40 GHz). (dynamic)

property frequency_stop

Control the stop value of the sweep range of the indicated channel in hertz.

Valid values are between 1E7 [Hz] (i.e. 10 MHz) and 4E10 [Hz] (i.e. 40 GHz). (dynamic)

property hold_function

Control the hold function of the specified channel.

valid values are:

value	description
CONT	Perform continuous sweeps on all channels
HOLD	Hold the sweep on all channels
SING	Perform a single sweep and then hold all channels

property number_of_points

Control the number of measurement points in a frequency sweep of the indicated channel.

Valid values are between 1 and 25000 or 100000 depending on the maximum points setting.

property number_of_traces

Control the number of traces on the specified channel

Valid values are between 1 and 16.

property sweep_mode

Control the sweep mode for Spectrum Analysis on the indicated channel.

Valid options are VNA (for a VNA-like mode where the instrument will only measure at points in the frequency list) or CLAS (for a classical mode, where the instrument will scan all frequencies in the range).

property sweep_time

Control the sweep time of the indicated channel.

Valid values are between 2 and 100000.

property sweep_type

Control the sweep type of the indicated channel.

Valid options are:

value	description
LIN	Frequency-based linear sweep
LOG	Frequency-based logarithmic sweep
FSEGM	Segment-based sweep with frequency-based segments
ISEGM	Index-based sweep with frequency-based segments
POW	Power-based sweep with either a CW frequency or swept-frequency
MFGC	Multiple frequency gain compression

update_frequency_range(*frequency_range*)

Update the values-attribute of the frequency-related dynamic properties.

Parameters
frequency_range (*list*) – the frequency range that the instrument is capable of.

update_traces(*number_of_traces=None*)

Create or remove traces to be correct with the actual number of traces.

Parameters
number_of_traces (*int*) – optional, if given defines the desired number of traces.

class pymeasure.instruments.anritsu.anritsuMS464xB.**Trace**(*parent*, *id*)

Bases: *Channel*

Represents a trace within a *MeasurementChannel* of the Anritsu MS464xB VNA.

activate()

Set the indicated trace as the active one.

property measurement_parameter

Control the measurement parameter of the indicated trace.

Valid values are any S-parameter (e.g. S11, S12, S41) for 4 ports, or one of the following:

value	description
Sxx	S-parameters (1-4 for both x)
MIX	Response Mixed Mode
NFIG	Noise Figure trace response (only with option 41 or 48)
NPOW	Noise Power trace response (only with option 41 or 48)
NTEMP	Noise Temperature trace response (only with option 41 or 48)
AGA	Noise Figure Available Gain trace response (only with option 48)
IGA	Noise Figure Insertion Gain trace response (only with option 48)

class pymeasure.instruments.anritsu.anritsuMS464xB.**Port**(*parent*, *id*)

Bases: *Channel*

Represents a port within a *MeasurementChannel* of the Anritsu MS464xB VNA.

property power_level

Control the power level (in dBm) of the indicated port on the indicated channel.

7.18 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*.

7.18.1 Attocube ANC300 Motion Controller

class pymeasure.instruments.attocube.anc300.**ANC300Controller**(*adapter=None*, *name='attocube ANC300 Piezo Controller'*, *axisnames=''*, *passwd=''*, *query_delay=0.05*, ***kwargs*)

Bases: *Instrument*

Attocube ANC300 Piezo stage controller with several axes

Parameters

- **adapter** – The VISA resource name of the controller (e.g. "TCPIP::<address>::<port>::SOCKET") or a created Adapter. The instruments default communication port is 7230.
- **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** – default delay between sending and reading in s (default 0.05)
- **host** – host address of the instrument (e.g. 169.254.0.1)

  Deprecated since version 0.11.2: The 'host' argument is deprecated. Use 'adapter' argument instead.
- **kwargs** – Any valid key-word argument for VISAAdapter

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

Returns
List of error entries.

property controllerBoardVersion

Get the serial number of the controller board.

ground_all()

Grounds all axis of the controller.

handle_deprecated_host_arg(*adapter*, *kwargs*)

This function formats user input to the __init__ function to be compatible with the current definition of the __init__ function. This is used to support outdated (deprecated) code. and separated out to make it easier to remove in the future. To whoever removes this: This function should be removed and the *adapter* argument in the __init__ method should be made non-optional.

Parameters
kwargs (*dict*) – keyword arguments passed to the __init__ function, including the deprecated *host* argument.

Return str
resource string for the VISAAdapter

read()

Read after setting a value.

stop_all()

Stop all movements of the axis.

property version

Get the version number and instrument identification.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None means query_delay.

class pymeasure.instruments.attocube.anc300.**Axis**(*parent*, *id*)

Bases: *Channel*

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

Measure the saved capacity value in nF of the axis.

property frequency

Control the frequency of the stepping motion in Hertz from 1 to 10000 Hz.

insert_id(*command*)

Insert the channel id in a command replacing *placeholder*.

Add axis id to a command string at the correct position after the initial command, but before a potential value.

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

Control 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 is blocking and 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

move_raw(*steps*)

Move 'steps' steps in the direction given by the sign of the argument. This method assumes the mode of the axis is set to 'stp' and it is non-blocking, i.e. it will return immediately after sending the command.

Parameters
steps – integer value of steps to be performed. A positive sign corresponds to upwards steps, a negative sign to downwards steps. The values of +/-inf trigger a continuous movement. The axis can be halted by the stop method.

property offset_voltage

Control offset voltage in Volts from 0 to 150 V.

property output_voltage

Measure the output voltage in volts.

property pattern_down

Control step down pattern of the piezo drive. 256 values ranging from 0 to 255 representing 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

Control step up pattern of the piezo drive. 256 values ranging from 0 to 255 representing 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

Get the serial number of the axis.

property stepd

Set the steps downwards for N steps. Mode must be 'stp' and N must be positive. 0 causes a continuous movement until stop is called.

Deprecated since version 0.13.0: Use meth:*move_raw* instead.

property stepu

Set the steps upwards for N steps. Mode must be 'stp' and N must be positive. 0 causes a continuous movement until stop is called.

Deprecated since version 0.13.0: Use meth:*move_raw* instead.

stop()

Stop any motion of the axis

property voltage

Control the amplitude of the stepping voltage in volts from 0 to 150 V.

7.19 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*.

7.19.1 BK Precision 9130B DC Power Supply

class pymeasure.instruments.bkprecision.**BKPrecision9130B**(*adapter*, *name='BK Precision 9130B Source'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the BK Precision 9130B DC Power Supply interface for interacting with the instrument.

property channel

Control which channel is selected. Can only take values [1, 2, 3]. (int)

property current

Control the current of the selected channel. (float)

property source_enabled

Control whether the source is enabled. (bool)

property voltage

Control voltage of the selected channel. (float)

7.20 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*.

7.20.1 Danfysik 8500 Power Supply

class pymeasure.instruments.danfysik.**Danfysik8500**(*adapter*, *name='Danfysik 8500 Current Supply'*, ***kwargs*)

Bases: *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

Control the actual current in Amps. This property can be set through *current_ppm*.

property current_ppm

Control the current in parts per million..

property current_setpoint

Get 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

Get 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

Control the polarity of the current supply, being either -1 or 1. This property can be set by supplying 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

Get 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

Get a list of human-readable strings that contain the instrument status information, based on *status_hex*.

property status_hex

Get 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

7.21 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*.

7.21.1 Delta Elektronica SM7045D Power source

class pymeasure.instruments.deltaelektronika.**SM7045D**(*adapter*, *name='Delta Elektronika SM 70-45 D'*, ***kwargs*)

Bases: `SCPIUnknownMixin`, *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.

7.22 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*.

7.22.1 Edwards nxds vacuum pump

class pymeasure.instruments.edwards.**Nxds**(*adapter*, *name='Edwards NXDS Vacuum Pump'*, ***kwargs*)

Bases: *Instrument*

Represents the Edwards nXDS (10i) Vacuum Pump and provides a low-level interaction with the instrument. This could potentially work with Edwards pump that has a RS232 interface. This instrument is constructed to only start and stop pump.

property enable

Set the pump enabled state with default settings.

7.23 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*.

7.23.1 Euro Test HPP120256 High Voltage Power Supply

class pymeasure.instruments.eurotest.**EurotestHPP120256**(*adapter*, *name='Euro Test High Voltage DC Source model HPP-120-256'*, *query_delay=0.1*, *write_delay=0.4*, *timeout=5000*, ***kwargs*)

Bases: *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 **EurotestHPP120256Status**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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.

ask(*command*)

Overrides Instrument ask method for including query_delay time on parent call. :param command: Command string to be sent to the instrument. :returns: String returned by the device without read_termination.

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

Get the identification of the instrument (string)

property kill_enabled

Control the instrument kill enable (boolean).

property lam_status

Get the instrument lam status (string).

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

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

Get 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_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(*command*, ***kwargs*)

Overrides Instrument write method for including write_delay time after the parent call.

Parameters
command – command string to be sent to the instrument

7.24 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*.

7.24.1 Fluke 7341 Temperature bath

class pymeasure.instruments.fluke.**Fluke7341**(*adapter*, *name='Fluke 7341'*, ***kwargs*)

Bases: *Instrument*

Represents the compact constant temperature bath from Fluke.

property id

Get the instrument model.

read()

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

Extract the value from the response string.

Responses are in the format "*type*: *value optional information*". Optional information is for example the unit (degree centigrade or Fahrenheit).

property set_point

Control the temperature setpoint (float from -40 to 150 °C) The unit is as defined in property *unit*.

property temperature

Measure the current bath temperature. The unit is as defined in property *unit*.

property unit

Control the temperature unit: *c* for Celsius and *f* for Fahrenheit`.

7.25 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*.

7.25.1 F.W. Bell 5080 Handheld Gaussmeter

class pymeasure.instruments.fwbell.**FWBell5080**(*adapter*, *name='F.W. Bell 5080 Handheld Gaussmeter'*, ***kwargs*)

Bases: *SCPIMixin*, *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.

check_errors()
Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()
Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()
Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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 field

Measure the field in the appropriate units (float).

fields(*samples=1*)

Returns a numpy array of field samples for a given sample number.

Parameters
samples – The number of samples to preform

property id

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property range

Control the maximum field range in the active units (int). 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.

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

Get the status byte and Master Summary Status bit.

property units

Get the field units (str), 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.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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 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*.

7.26.1 Heidenhain ND287 Position Display Unit

class pymeasure.instruments.heidenhain.**ND287**(*adapter*, *name='Heidenhain ND287'*, *units='mm'*, ***kwargs*)

Bases: *Instrument*

Represents the Heidenhain ND287 position display unit used to readout and display absolute position measured by Heidenhain encoders.

check_errors()

Method to read an error status message and log when an error is detected.

Returns

String with the error message as its contents.

property id

Get the string identification property for the device.

property position

Measure the encoder's current position (float). Note that the get_process performs a mapping from the returned value to a float measured in the units specified by *ND287.units*. The get_process is modified dynamically as this mapping changes slightly between different units.(dynamic)

property status

Get the encoder's status bar

property units

Control the unit of measure set on the device. Valid values are 'mm' and 'inch' Note that this parameter can only be set manually on the device. So this argument only ensures that the instance units and physical device settings match. I.e., this property does not change any physical device setting.

7.27 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*.

7.27.1 HCP TC038 crystal oven

class pymeasure.instruments.hcp.**TC038**(*adapter*, *name='TC038'*, *address=1*, *timeout=1000*, ***kwargs*)

Bases: *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_set_errors()

Check for errors after having set a property.

Called if `check_set_errors=True` is set for that property.

property information

Get the information about the device and its capabilities.

property monitored_value

Measure 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

Control the setpoint of the temperature controller in °C.

property temperature

Measure the current temperature in °C.

write(*command*)

Send a *command* in its own protocol.

7.27.2 HCP TC038D crystal oven

class pymeasure.instruments.hcp.**TC038D**(*adapter*, *name='TC038D'*, *address=1*, *timeout=1000*, ***kwargs*)

Bases: *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_set_errors()

Check for errors after having set a property.

Called if `check_set_errors=True` is set for that property.

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

Control the setpoint of the oven in °C.

property temperature

Measure 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).

7.28 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*.

If the instrument you are looking for is not here, also check *Agilent* or *Keysight* for newer instruments.

7.28.1 HP 33120A Arbitrary Waveform Generator

class pymeasure.instruments.hp.**HP33120A**(*adapter*, *name='Hewlett Packard 33120A Function Generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Hewlett Packard 33120A Arbitrary Waveform Generator and provides a high-level interface for interacting with the instrument.

property amplitude

Control 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*. (float)

property amplitude_units

Control the units of the amplitude, which can take the values Vpp, Vrms, dBm, and default. (str)

beep()

Causes a system beep.

property burst_count

Control the number of cycles per burst (1 to 50,000 cycles)

property burst_enabled

Control state of burst modulation

property burst_phase

Control the starting phase angle of a burst (-360 to +360 degrees)

property burst_rate

Control the burst rate in Hz fo an internal burst source

property burst_source

Control internal or external gate source for burst modulation

property frequency

Control 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*. (float)

property max_amplitude

Get the maximum *amplitude* in Volts for the given shape

property max_burst_count

Get the maximum *burst_count*

property max_burst_phase

Get the maximum *burst_phase*

property max_burst_rate

Get the maximum *burst_rate*

property max_frequency

Get the maximum *frequency* in Hz for the given shape

property max_offset

Get the maximum *offset* in Volts for the given shape

property min_amplitude

Get the minimum *amplitude* in Volts for the given shape

property min_burst_count

Get the minimum *burst_count*

property min_burst_phase

Get the minimum *burst_phase*

property min_burst_rate

Get the minimum *burst_rate*

property min_frequency

Get the minimum *frequency* in Hz for the given shape

property min_offset

Get the minimum *offset* in Volts for the given shape

property offset

Control 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

Control the shape of the wave, which can take the values: sinusoid, square, triangle, ramp, noise, dc, and user. (str)

7.28.2 HP 34401A Multimeter

class pymeasure.instruments.hp.**HP34401A**(*adapter*, *name='HP 34401A'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the HP / Agilent / Keysight 34401A Multimeter and provides a high-level interface for interacting with the instrument.

```
dmm = HP34401A("GPIB::1")
dmm.function_ = "DCV"
print(dmm.reading)  # -> Single float reading

dmm.nplc = 0.02
dmm.autozero_enabled = False
dmm.trigger_count = 100
dmm.trigger_delay = "MIN"
print(dmm.reading)  # -> Array of 100 very fast readings
```

property auto_input_impedance_enabled

Control if automatic input resistance mode is enabled.

Only valid for dc voltage measurements. When disabled (default), the input resistance is fixed at 10 MOhms for all ranges. With AUTO ON, the input resistance is set to >10 GOhms for the 100 mV, 1 V, and 10 V ranges.

property autorange

Control the autorange state for the currently active function.

property autozero_enabled

Control the autozero state.

beep()

This command causes the multimeter to beep once.

property beeper_enabled

Control whether the beeper is enabled.

property current_ac

AC current, in Amps

Deprecated since version 0.12: Use the `function_` and `reading` properties instead.

property current_dc

DC current, in Amps

Deprecated since version 0.12: Use the `function_` and `reading` properties instead.

property detector_bandwidth

Control the lowest frequency expected in the input signal in Hertz.

Valid values: 3, 20, 200, “MIN”, “MAX”.

property display_enabled

Control the display state.

property displayed_text

Control the text displayed on the multimeter’s display.

The text can be up to 12 characters long; any additional characters are truncated my the multimeter.

property function_

Control the measurement function.

Allowed values: “DCV”, “DCV_RATIO”, “ACV”, “DCI”, “ACI”, “R2W”, “R4W”, “FREQ”, “PERIOD”, “CONTINUITY”, “DIODE”.

property gate_time

Control the gate time (or aperture time) for frequency or period measurements.

Valid values: 0.01, 0.1, 1, “MIN”, “MAX”. Specifically: 10 ms (4.5 digits), 100 ms (default; 5.5 digits), or 1 second (6.5 digits).

init_trigger()

Set the state of the triggering system to “wait-for-trigger”.

Measurements will begin when the specified trigger conditions are satisfied after this command is received.

property nplc

Control the integration time in number of power line cycles (NPLC).

Valid values: 0.02, 0.2, 1, 10, 100, "MIN", "MAX". This command is valid only for dc volts, ratio, dc current, 2-wire ohms, and 4-wire ohms.

property range_

Control the range for the currently active function.

For frequency and period measurements, ranging applies to the signal's input voltage, not its frequency

property reading

Take a measurement of the currently selected function.

Reading this property is equivalent to calling *init_trigger()*, waiting for completion and fetching the reading(s).

property remote_control_enabled

Control whether remote control is enabled.

property remote_lock_enabled

Control whether the beeper is enabled.

property resistance

Resistance, in Ohms

Deprecated since version 0.12: Use the `function_` and `reading` properties instead.

property resistance_4w

Four-wires (remote sensing) resistance, in Ohms

Deprecated since version 0.12: Use the `function_` and `reading` properties instead.

property resolution

Control the resolution of the measurements.

Not valid for frequency, period, or ratio. Specify the resolution in the same units as the measurement function, not in number of digits. Results in a "Settings Conflict" error if autorange is enabled. MIN selects the smallest value accepted, which gives the most resolution. MAX selects the largest value accepted which gives the least resolution.

property sample_count

Controls the number of samples per trigger event.

Valid values: 1 to 50000, "MIN", "MAX".

property scpi_version

The SCPI version of the multimeter.

property self_test_result

Initiate a self-test of the multimeter and return the result.

Be sure to set an appropriate connection timeout, otherwise the command will fail.

property stored_reading

Measure the reading(s) currently stored in the multimeter's internal memory.

Reading this property will NOT initialize a trigger. If you need that, use the *reading* property instead.

property stored_readings_count

The number of readings currently stored in the internal memory.

property terminals_used

Query the multimeter to determine if the front or rear input terminals are selected.

Returns "FRONT" or "REAR".

property trigger_auto_delay_enabled

Control the automatic trigger delay state.

If enabled, the delay is determined by function, range, integration time, and ac filter setting. Selecting a specific trigger delay value automatically turns off the automatic trigger delay.

property trigger_count

Control the number of triggers accepted before returning to the "idle" state.

Valid values: 1 to 50000, "MIN", "MAX", "INF". The INFinite parameter instructs the multimeter to continuously accept triggers (you must send a device clear to return to the "idle" state).

property trigger_delay

Control the trigger delay in seconds.

Valid values (incl. floats): 0 to 3600 seconds, "MIN", "MAX".

trigger_single_autozero()

Trigger an autozero measurement.

Consequent autozero measurements are disabled.

property trigger_source

Control the trigger source.

Valid values: "IMM", "BUS", "EXT" The multimeter will accept a software (bus) trigger, an immediate internal trigger (this is the default source), or a hardware trigger from the rear-panel Ext Trig (external trigger) terminal.

property voltage_ac

AC voltage, in Volts

Deprecated since version 0.12: Use the `function_` and `reading` properties instead.

write(*command*)

Write a command to the instrument.

7.28.3 HP 3437A System-Voltmeter

class pymeasure.instruments.hp.**HP3437A**(*adapter*, *name='Hewlett-Packard HP3437A'*, ***kwargs*)

Bases: *HPLegacyInstrument*

Represents the Hewlett Packard 3737A system voltmeter and provides a high-level interface for interacting with the instrument.

class SRQ(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Enum element for SRQ mask bit decoding

property SRQ_mask

Return current SRQ mask, this property can be set,

bit assignment 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 always 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 `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 `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,

Possible 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

7.28.4 HP 3478A Multimeter

class pymeasure.instruments.hp.**HP3478A**(*adapter*, *name='Hewlett-Packard HP3478A'*, ***kwargs*)

Bases: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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 `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`.

7.28.5 HP 8116A 50 MHz Pulse/Function Generator

class pymeasure.instruments.hp.**HP8116A**(*adapter*, *name='Hewlett-Packard 8116A'*, ***kwargs*)

Bases: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum of the digits used with the autovernier (see *HP8116A.start_autovernier()*).

class Direction(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum of the directions used with the autovernier (see *HP8116A.start_autovernier()*).

GPIB_trigger()

Initiate trigger via low-level GPIB-command (aka GET - group execute trigger).

property amplitude

Control the amplitude of the output in V (strict float from 10e-3 to 16). The allowed amplitude range 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

Control whether the autovernier is enabled (bool).

property burst_number

Control the number of periods generated in a burst (strict int from 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

Control whether the complement of the signal is generated (bool).

property complete

Get whether the measurement is complete (bool).

property control_mode

Control the control mode of the instrument. Possible values are 'off', 'FM', 'AM', 'PWM', 'VCO'.

property duty_cycle

Control the duty cycle of the output in percent (float). 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

Control the frequency of the output in Hz (strict float from 1e-3 to 52.5e6).

property haversine_enabled

Control whether a haversine/havertriangle signal is generated when in 'triggered', 'internal_burst' or 'external_burst' operating mode.

property high_level

Control the high level of the output in V (strict float from -7.9 to 8). The allowed high level range must be at least 10 mV greater than the low level.

property limit_enabled

Control whether parameter limiting is enabled (bool).

property low_level

Control the low level of the output in V (strict float from -8 to 7.9). The allowed low level range must be at least 10 mV less than the high level.

property offset

Control the offset of the output in V (strit float from -7.95 to 7.95). The allowed offset range is also limited by the amplitude.

property operating_mode

Control 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

Get the device options installed. The only possible option is 001.

property output_enabled

Control whether the output is enabled (bool).

property pulse_width

Control the pulse width in s (strict float from 8e-9 to 999e-3). The pulse width may not be larger than the period.

property repetition_rate

Control the repetition rate in s (i.e. the time between bursts) in 'internal_burst' mode (strict float from 20e-9 to 999e-3).

reset()

Initatiate a reset (like a power-on reset) of the 8116A.

property shape

Control 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

Get the status byte of the 8116A as a `Status` IntFlag-type enum.

property sweep_marker_frequency

Control the frequency marker in both sweep modes in Hz (strict float from 1e-3 to 52.5e6). At this frequency, the marker output switches from low to high.

property sweep_start

Control the start frequency in both sweep modes in Hz (strict float from 1e-3 to 52.5e6).

property sweep_stop

Control the stop frequency in both sweep modes in Hz (strict float from 1e-3 to 52.5e6).

property sweep_time

Control the sweep time per decade in both sweep modes in s (float). The sweep time is selectable in a 1-2-5 sequence between 10 ms and 500 s.

property trigger_slope

Control 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.

7.28.6 HP 8560A / 8561B Spectrum Analyzer

Every unit is used in the base unit, so for time it is s (Seconds), frequency in Hz (Hertz) etc…

Generic Specific Attributes & Methods

Content

General

HP856Xx.**preset**()

Set the spectrum analyzer to a known, predefined state.

'preset' does not affect the contents of any data or trace registers or stored preselector data. 'preset' does not clear the input or output data buffers;

HP856Xx.**attenuation**

Control the input attenuation in decade steps from 10 to 70 db (type 'int') or set to 'AUTO' and 'MAN'(ual)

Type: `str`, `int`

```
instr.attenuation = 'AUTO'
instr.attenuation = 60
```

HP856Xx.**amplitude_unit**

Control the amplitude unit with a selection of the following parameters: string 'DBM', 'DBMV', 'DBUV', 'V', 'W', 'AUTO', 'MAN' or use the enum *AmplitudeUnits*

Type: str

```
instr.amplitude_unit = 'dBmV'
instr.amplitude_unit = AmplitudeUnits.dBmV
```

HP856Xx.**trigger_mode**

Control the trigger mode. Selected trigger conditions must be met in order for a sweep to occur. For the available modes refer to *TriggerMode*. When any trigger mode other than free run is selected, a "T" appears on the left edge of the display.

HP856Xx.**detector_mode**

Control the IF detector used for acquiring measurement data. This is normally a coupled function, in which the spectrum analyzer selects the appropriate detector mode. Four modes are available: normal, positive, negative, and sample.

Type: str

Takes a representation of the detector mode, either from *DetectionModes* or use 'NEG', 'NRM', 'POS', 'SMP'

```
instr.detector_mode = DetectionModes.SMP
instr.detector_mode = 'NEG'

if instr.detector_mode == DetectionModes.SMP:
    pass
```

HP856Xx.**coupling**

Control the input coupling of the spectrum analyzer. AC coupling protects the input of the analyzer from damaging dc signals, while limiting the lower frequency-range to 100 kHz (although the analyzer will tune down to 0 Hz with signal attenuation).

Type: str

Takes a representation of the coupling mode, either from *CouplingMode* or use 'AC' / 'DC'

```
instr.coupling = 'AC'
instr.coupling = CouplingMode.DC

if instr.coupling == CouplingMode.DC:
    pass
```

HP856Xx.**set_auto_couple**()

Set the video bandwidth, resolution bandwidth, input attenuation, sweep time, and center frequency step-size to coupled mode.

These functions can be recoupled individually or all at once. The spectrum analyzer chooses appropriate values for these functions. The video bandwidth and resolution bandwidth are set according to the coupled ratios stored under *resolution_bandwidth_to_span_ratio* and *video_bandwidth_to_resolution_bandwidth*. If no ratios are chosen, default ratios (1.0 and 0.011, respectively) are used instead.

HP856Xx.**set_linear_scale**()

Set the spectrum analyzers display to linear amplitude scale.

Measurements made on a linear scale can be read out in any units.

HP856Xx.**logarithmic_scale**

Control the logarithmic amplitude scale. When in linear mode, querying 'logarithmic_scale' returns a "0". Allowed values are 0, 1, 2, 5, 10

Type: `int`

```
if instr.logarithmic_scale:
    pass

# set the scale to 10 db per division
instr.logarithmic_scale = 10
```

HP856Xx.**threshold**

Control the minimum amplitude level and clips data at this value. Default value is -90 dBm. See also - *marker_threshold* does not clip data below its threshold

Type: `str, float` range -200 to 30

> **Note**
>
> When a trace is in max-hold mode, if the threshold is raised above any of the trace data, the data below the threshold will be permanently lost.

HP856Xx.**set_title**(*string*)

Sets character data in the title area of the display, which is in the upper-right corner.

A title can be up to two rows of sixteen characters each, Carriage return and line feed characters are not allowed.

HP856Xx.**status**

Get the decimal equivalent of the bits set in the status byte (see the RQS and SRQ commands). STB is equivalent to a serial poll command. The RQS and associated bits are cleared in the same way that a serial poll command would clear them.

HP856Xx.**check_done**()

Return when all commands in a command string entered before :meth:'check_done' has been completed. Sending a *trigger_sweep()* command before 'check_done' ensures that the spectrum analyzer will complete a full sweep before continuing on in a program. Depending on the timeout a timeout error from the adapter will raise before the spectrum analyzer can finish due to an extreme long sweep time.

```
instr.trigger_sweep()

# wait for a full sweep and than 'do_something'
instr.check_done()
do_something()
```

HP856Xx.**request_service**(*input*)

Triggers a service request. This command allows you to force a service request and test a program designed to handle service requests. However, the service request can be triggered only if it is first masked using the *request_service_conditions* command.

Parameters
input (*StatusRegister*) – Bits to emulate a service request

HP856Xx.**errors**

Get a list of errors present (of type *ErrorCode*). An empty list means there are no errors. Reading 'errors' clears all HP-IB errors. For best results, enter error data immediately after querying for errors.

Type: *ErrorCode*

```
errors = instr.errors
if len(errors) > 0:
    print(errors[0].code)

for error in errors:
    print(error)

if ErrorCode(112) in errors:
    print("yeah")
```

Example result of this python snippet:

```
112
ErrorCode("??CMD?? - Unrecognized command")
ErrorCode("NOP NUM - Command cannot have numeric units")
yeah
```

HP856Xx.**save_state**(*inp*)

Saves the currently displayed instrument state in the specified state register.

Parameters

- **inp** – State to be recalled: either storage slot 0 … 9 or 'LAST' or 'PWRON'
- **inp** – str, int

```
instr.preset()
instr.center_frequency = 300e6
instr.span = 20e6
instr.save_state("PWRON")
```

HP856Xx.**recall_state**(*inp*)

Set to the display a previously saved instrument state. See *save_state()*.

Parameters

- **inp** – State to be recalled: either storage slot 0 … 9 or 'LAST' or 'PWRON'
- **inp** – str, int

```
instr.save_state(7)
instr.preset()
instr.recall_state(7)
```

HP856Xx.**request_service_conditions**

Control a bit mask that specifies which service requests can interrupt a program sequence.

```
instr.request_service_conditions = StatusRegister.ERROR_PRESENT | StatusRegister.
↪TRIGGER

print(instr.request_service_conditions)
StatusRegister.ERROR_PRESENT|TRIGGER
```

HP856Xx.**set_maximum_hold = <function HP856Xx.set_maximum_hold>**

HP856Xx.**set_minimum_hold = <function HP856Xx.set_minimum_hold>**

HP856Xx.**reference_level_calibration**

Control the calibration of the reference level remotely and retuns the current calibration. To calibrate the reference level, connect the 300 MHz calibration signal to the RF input. Set the center frequency to 300 MHz, the frequency span to 20 MHz, and the reference level to -10 dBm. Use the RLCAL command to move the input signal to the reference level. When the signal peak falls directly on the reference-level line, the reference level is calibrated. Storing this value in the analyzer in EEROM can be done only from the front panel. The RLCAL command, when queried, returns the current value.

Type: `float`

```
# connect cal signal to rf input
instr.preset()
instr.amplitude_unit = AmplitudeUnits.DBM
instr.center_frequency = 300e6
instr.span = 100e3
instr.reference_level = 0
instr.trigger_sweep()

instr.peak_search(PeakSearchMode.High)
level = instr.marker_amplitude
rlcal = instr.reference_level_calibration - int((level + 10) / 0.17)
instr.reference_level_calibration = rlcal
```

HP856Xx.**reference_offset**

Control an offset applied to all amplitude readouts (for example, the reference level and marker amplitude). The offset is in dB, regardless of the selected scale and units. The offset can be useful to account for gains of losses in accessories connected to the input of the analyzer. When this function is active, an “R” appears on the left edge of the display.

Type: `int`

HP856Xx.**reference_level**

Control the reference level, or range level when in normalized mode. (Range level functions the same as reference level.) The reference level is the top horizontal line on the graticule. For best measurement accuracy, place the peak of a signal of interest on the reference-level line. The spectrum analyzer input attenuator is coupled to the reference level and automatically adjusts to avoid compression of the input signal. Refer also to *amplitude_unit*. Minimum reference level is -120.0 dBm or 2.2 uV

Type: `float`

HP856Xx.**display_line**

Control the horizontal display line for use as a visual aid or for computational purposes. The default value is 0 dBm.

Type: `float`

Takes a value with the unit of *amplitude_unit*

```
instr.display_line = -10

if instr.display_line == 0:
    pass
```

HP856Xx.**protect_state_enabled**

Control the storing of any new data in the state or trace registers. If set to 'True', the registers are "locked"; the data in them cannot be erased or overwritten, although the data can be recalled. To "unlock" the registers, and store new data, set 'protect_state_enabled' to off by selecting 'False' as the parameter.

Type: `bool`

HP856Xx.**mixer_level**

Control the maximum signal level that is at the input mixer. The attenuator automatically adjusts to ensure that this level is not exceeded for signals less than the reference level. From -80 to -10 DB.

Type: `int`

HP856Xx.**frequency_counter_mode_enabled**

Set the device into a frequency counter mode that counts the frequency of the active marker or the difference in frequency between two markers. If no marker is active, 'frequency_counter_mode_enabled' places a marker at the center of the trace and counts that marker frequency. The frequency counter provides a more accurate frequency reading; it pauses at the marker, counts the value, then continues the sweep. To adjust the frequency counter resolution, use the 'frequency_counter_resolution' command. To return the counter value, use the 'marker_frequency' command.

```
instr.frequency_counter_mode_enabled = True
```

HP856Xx.**frequency_counter_resolution**

Control the resolution of the frequency counter. Refer to the 'frequency_counter_mode' command. The default value is 10 kHz.

Type `int`

```
# activate frequency counter mode
instr.frequency_counter_mode = True

# adjust resolution to 1 Hz
instr.frequency_counter_resolution = 1

if instr.frequency_counter_resolution:
    pass
```

HP856Xx.**adjust_all**()

Activate the local oscillator (LO) and intermediate frequency (IF) alignment routines. These are the same routines that occur when is switched on. Commands following 'adjust_all' are not executed until after the analyzer has finished the alignment routines.

HP856Xx.**adjust_if**

Control the automatic IF adjustment. This function is normally on. Because the IF is continuously adjusting, executing the IF alignment routine is seldom necessary. When the IF adjustment is not active, an "A" appears on the left side of the display.

- *"FULL"* IF adjustment is done for all IF settings.
- *"CURR"* IF adjustment is done only for the IF settings currently displayed.

- *False* turns the continuous IF adjustment off.
- *True* reactivates the continuous IF adjustment.

Type: bool, str

HP856Xx.**hold**()

Freeze the active function at its current value.

If no function is active, no operation takes place.

HP856Xx.**annotation_enabled**

Set the display annotation off or on.

Type: bool

HP856Xx.**set_crt_adjustment_pattern**()

Activate a CRT adjustment pattern, shown in Figure 5-3. Use the X POSN, Y POSN, and TRACE ALIGN adjustments (available from the rear panel) to align the display. Use X POSN and Y POSN to move the display horizontally and vertically, respectively. Use TRACE ALIGN to straighten a tilted display. To remove the pattern from the screen, execute the *preset()* command.

HP856Xx.**display_parameters**

Get the location of the lower left (P1) and upper right (P2) vertices as a tuple of the display window.

Type: tuple

```
repr(instr.display_parameters)
(72, 16, 712, 766)
```

HP856Xx.**firmware_revision**

Get the revision date code of the spectrum analyzer firmware.

Type: datetime.date

HP856Xx.**graticule_enabled**

Control the display graticule. Switch it either on or off.

Type: bool

```
instr.graticule = True

if instr.graticule:
    pass
```

HP856Xx.**serial_number**

Get the spectrum analyzer serial number.

HP856Xx.**id**

Get the identification of the device with software and hardware revision (e.g. HP8560A,002, H03)

Type: str

```
print(instr.id)
HP8560A,002,H02
```

HP856Xx.**elapsed_time**

Get the elapsed time (in hours) of analyzer operation. This value can be reset only by Hewlett-Packard.

Type: int

```
print(elapsed_time)
1998
```

Demodulation

HP856Xx.**demodulation_mode**

Control the demodulation mode of the spectrum analyzer. Either AM or FM demodulation, or turns the demodulation — off. Place a marker on a desired signal and then set *demodulation_mode*; demodulation takes place on this signal. If no marker is on, *demodulation_mode* automatically places a marker at the center of the trace and demodulates the frequency at that marker position. Use the volume and squelch controls to adjust the speaker and listen.

Type: `str`

Takes a representation of the demodulation mode, either from *DemodulationMode* or use 'OFF', 'AM', 'FM'

```
instr.demodulation_mode = 'AC'
instr.demodulation_mode = DemodulationMode.AM

if instr.demodulation_mode == DemodulationMode.FM:
    instr.demodulation_mode = Demodulation.OFF
```

HP856Xx.**demodulation_agc_enabled**

Control the demodulation automatic gain control (AGC). The AGC keeps the volume of the speaker relatively constant during AM demodulation. AGC is available only during AM demodulation and when the frequency span is greater than 0 Hz.

Type: `bool`

```
instr.demodulation_agc = True

if instr.demodulation_agc:
    instr.demodulation_agc = False
```

HP856Xx.**demodulation_time**

Control the amount of time that the sweep pauses at the marker to demodulate a signal. The default value is 1 second. When the frequency span equals 0 Hz, demodulation is continuous, except when between sweeps. For truly continuous demodulation, set the frequency span to 0 Hz and the trigger mode to single sweep (see TM). Minimum 100 ms to maximum 60 s

Type: `float`

```
# set the demodulation time to 1.2 seconds
instr.demodulation_time = 1.2

if instr.demodulation_time == 10:
    pass
```

HP856Xx.**squelch**

Control the squelch level for demodulation. When this function is on, a dashed line indicating the squelch level appears on the display. A marker must be active and above the squelch line for demodulation to occur. Refer to the *demodulation_mode* command. The default value is -120 dBm.

Type: `str,int`

```
instr.preset()
instr.start_frequency = 88e6
instr.stop_frequency = 108e6

instr.peak_search(PeakSearchMode.High)
instr.demodulation_time = 10

instr.squelch = -60
instr.demodulation_mode = DemodulationMode.FM
```

Frequency

HP856Xx.**start_frequency**

Control the start frequency and set the spectrum analyzer to start-frequency/ stop-frequency mode. If the start frequency exceeds the stop frequency, the stop frequency increases to equal the start frequency plus 100 Hz. The center frequency and span change with changes in the start frequency.

Type: `float`

```
instr.start_frequency = 300.5e6
if instr.start_frequency == 200e3:
    print("Correct frequency")
```

(dynamic)

HP856Xx.**stop_frequency**

Control the stop frequency and set the spectrum analyzer to start-frequency/ stop-frequency mode. If the stop frequency is less than the start frequency, the start frequency decreases to equal the stop frequency minus 100 Hz. The center frequency and span change with changes in the stop frequency.

Type: `float`

```
instr.stop_frequency = 300.5e6
if instr.stop_frequency == 200e3:
    print("Correct frequency")
```

(dynamic)

HP856Xx.**center_frequency**

Control the center frequency in hertz and sets the spectrum analyzer to center frequency / span mode.

The span remains constant; the start and stop frequencies change as the center frequency changes.

Type: `float`

```
instr.center_frequency = 300.5e6
if instr.center_frequency == 200e3:
    print("Correct frequency")
```

(dynamic)

HP856Xx.**frequency_offset**

Control an offset added to the displayed absolute-frequency values, including marker-frequency values.

It does not affect the frequency range of the sweep, nor does it affect relative frequency readouts. When this function is active, an "F" appears on the left side of the display. Changes all the following frequency measurements.

Type: `float`

```
instr.frequency_offset = 2e6
if instr.frequency_offset == 2e6:
    print("Correct frequency")
```

(dynamic)

HP856Xx.**frequency_reference_source**

Control the frequency reference source. Select either the internal frequency reference (INT) or supply your own external reference (EXT). An external reference must be 10 MHz (+100 Hz) at a minimum amplitude of 0 dBm. Connect the external reference to J9 (10 MHz REF IN/OUT) on the rear panel. When the external mode is selected, an "X" appears on the left edge of the display.

Type: `str`

Takes element of *FrequencyReference* or use 'INT', 'EXT'

```
instr.frequency_reference_source = 'INT'
instr.frequency_reference_source = FrequencyReference.EXT

if instr.frequency_reference_source == FrequencyReference.INT:
    instr.frequency_reference_source = FrequencyReference.EXT
```

HP856Xx.**span**

Control the frequency span. The center frequency does not change with changes in the frequency span; start and stop frequencies do change. Setting the frequency span to 0 Hz effectively allows an amplitude-versus-time mode in which to view signals. This is especially useful for viewing modulation. Querying SP will leave the analyzer in center frequency /span mode.

HP856Xx.**set_full_span**()

Set the spectrum analyzer to the full frequency span as defined by the instrument.

The full span is 2.9 GHz for the HP 8560A. For the HP 8561B, the full span is 6.5 GHz.

HP856Xx.**frequency_display_enabled**

Get the state of all annotations that describes the spectrum analyzer frequency. returns 'False' if no annotations are shown and vice versa 'True'. This includes the start and stop frequencies, the center frequency, the frequency span, marker readouts, the center frequency step-size, and signal identification to center frequency. To retrieve the frequency data, query the spectrum analyzer.

Type: `bool`

```
if instr.frequency_display:
    print("Frequencies get displayed")
```

Resolution Bandwidth

HP856Xx.**resolution_bandwidth**

Control the resolution bandwidth. This is normally a coupled function that is selected according to the ratio selected by the RBR command. If no ratio is selected, a default ratio (0.011) is used. The bandwidth, which ranges from 10 Hz to 2 MHz, may also be selected manually.

Type: `str, dec`

HP856Xx.**resolution_bandwidth_to_span_ratio**

Control the coupling ratio between the resolution bandwidth and the frequency span. When the frequency span is changed, the resolution bandwidth is changed to satisfy the selected ratio. The ratio ranges from 0.002 to 0.10. The "UP" and "DN" parameters adjust the ratio in a 1, 2, 5 sequence. The default ratio is 0.011.

Video

HP856Xx.**video_trigger_level**

Control the video trigger level when the trigger mode is set to VIDEO (refer to the *trigger_mode* command). A dashed line appears on the display to indicate the level. The default value is 0 dBm. Range -220 to 30.

Type: `float`

HP856Xx.**video_bandwidth_to_resolution_bandwidth**

Control the coupling ratio between the video bandwidth and the resolution bandwidth. Thus, when the resolution bandwidth is changed, the video bandwidth changes to satisfy the ratio. The ratio ranges from 0.003 to 3 in a 1, 3, 10 sequence. The default ratio is 1. When a new ratio is selected, the video bandwidth changes to satisfy the new ratio—the resolution bandwidth does not change value.

HP856Xx.**video_bandwidth**

Control the video bandwidth. This is normally a coupled function that is selected according to the ratio selected by the VBR command. (If no ratio is selected, a default ratio, 1.0, is used instead.) Video bandwidth filters (or smooths) post-detected video information. The bandwidth, which ranges from 1 Hz to 3 MHz, may also be selected manually. If the specified video bandwidth is less than 300 Hz and the resolution bandwidth is greater than or equal to 300 Hz, the IF detector is set to sample mode. Reducing the video bandwidth or increasing the number of video averages will usually smooth the trace by about as much for the same total measurement time. Reducing the video bandwidth to one-third or less of the resolution bandwidth is desirable when the number of video averages is above 25. For the case where the number of video averages is very large, and the video bandwidth is equal to the resolution bandwidth, internal mathematical limitations allow about 0.4 dB overresponse to noise on the logarithmic scale. The overresponse is negligible (less than 0.1 dB) for narrower video bandwidths.

Type: `int`

HP856Xx.**video_average**

Control the video averaging function. Video averaging smooths the displayed trace without using a narrow bandwidth. 'video_average' sets the IF detector to sample mode (see the DET command) and smooths the trace by averaging successive traces with each other. If desired, you can change the detector mode during video averaging. Video averaging is available only for trace A, and trace A must be in clear-write mode for 'video_average' to operate. After 'video_average' is executed, the number of sweeps that have been averaged appears at the top of the analyzer screen. Using video averaging allows you to view changes to the entire trace much faster than using narrow video filters. Narrow video filters require long sweep times, which may not be desired. Video averaging, though requiring more sweeps, uses faster sweep times; in some cases, it can produce a smooth trace as fast as a video filter.

Type: `str, int`

FFT & Measurements

HP856Xx.**create_fft_trace_window**(*trace*, *window_mode*)

Creates a window trace array for the fast Fourier transform (FFT) function.

The trace-window function creates a trace array according to three built-in algorithms: UNIFORM, HANNING, and FLATTOP. When used with the FFT command, the three algorithms give resultant passband shapes that represent a compromise among amplitude uncertainty, sensitivity, and frequency resolution. Refer to the FFT command description for more information.

Parameters

- **trace** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **window_mode** (*str*) – A representation of the window mode, either from *WindowType* or use 'HANNING', 'FLATTOP' or 'UNIFORM'

HP856Xx.**get_power_bandwidth**(*trace*, *percent*)

Measure the combined power of all signal responses contained in a trace array. The command then computes the bandwidth equal to a percentage of the total power. For example, if 100% is specified, the power bandwidth equals the current frequency span. If 50% is specified, trace elements are eliminated from either end of the array, until the combined power of the remaining trace elements equals half of the total power computed. The frequency span of these remaining trace elements is the power bandwidth output to the controller.

Parameters

- **trace** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **percent** (*float*) – Percentage of total power 0 . . . 100 %

```
# reset spectrum analyzer
instr.preset()

# set to single sweep mode
instr.sweep_single()

instr.center_frequency = 300e6
instr.span = 1e6

instr.maximum_hold()

instr.trigger_sweep()

if instr.done:
    pbw = instr.power_bandwidth(Trace.A, 99.0)
    print("The power bandwidth at 99 percent is %f kHz" % (pbw / 1e3))
```

HP856Xx.**do_fft**(*source*, *destination*, *window*)

Calculate and show a discrete Fourier transform.

The FFT command performs a discrete Fourier transform on the source trace array and stores the logarithms of the magnitudes of the results in the destination array. The maximum length of any of the traces is 601 points. FFT is designed to be used in transforming zero-span amplitude-modulation information into the frequency domain. Performing an FFT on a frequency sweep will not provide time-domain results. The FFT results are displayed on the spectrum analyzer in a logarithmic amplitude scale. For the horizontal dimension, the frequency at the left side of the graph is 0 Hz, and at the right side is Finax- Fmax is equal to 300 divided by sweep time. As an

example, if the sweep time of the analyzer is 60 ms, Fmax equals 5 kHz. The FFT algorithm assumes that the sampled signal is periodic with an integral number of periods within the time-record length (that is, the sweep time of the analyzer). Given this assumption, the transform computed is that of a time waveform of infinite duration, formed of concatenated time records. In actual measurements, the number of periods of the sampled signal within the time record may not be integral. In this case, there is a step discontinuity at the intersections of the concatenated time records in the assumed time waveform of infinite duration. This step discontinuity causes measurement errors, both amplitude uncertainty (where the signal level appears to vary with small changes in frequency) and frequency resolution (due to filter shape factor and sidelobes). Windows are weighting functions that are applied to the input data to force the ends of that data smoothly to zero, thus reducing the step discontinuity and reducing measuremen errors.

Parameters

- **source** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **destination** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **window** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B

Trace

HP856Xx.**view_trace**(*trace*)

Display the current contents of the selected trace, but does not update the contents. View mode may be executed before a sweep is complete when *sweep_single()* and *trigger_sweep()* are not used.

Parameters

trace (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B

Raises

- **TypeError** – Type isn't 'string'
- **ValueError** – Value is 'TRA' nor 'TRB'

HP856Xx.**get_trace_data_a**()

Get the data of trace A as a list.

The function returns the 601 data points as a list in the amplitude format. Right now it doesn't support the linear scaling due to the manual just being wrong.

HP856Xx.**get_trace_data_b**()

Get the data of trace B as a list.

The function returns the 601 data points as a list in the amplitude format. Right now it doesn't support the linear scaling due to the manual just being wrong.

HP856Xx.**set_trace_data_a**

Set the trace data of trace A.

> **Warning**
>
> The string based method this attribute is using takes its time. Something around 5000ms timeout at the adapter seems to work well.

HP856Xx.**set_trace_data_b**

Set the trace data of trace B also allows to write the data.

> **Warning**
>
> The string based method this attribute is using takes its time. Something around 5000ms timeout at the adapter seems to work well.

HP856Xx.**trace_data_format**

Control the format used to input and output trace data (see the TRA/TRB command, You must specify the desired format when transferring data from the spectrum analyzer to a computer; this is optional when transferring data to the analyzer.

Type: `str` or `TraceDataFormat`

> **Warning**
>
> Only needed for manual read out of trace data. Don't use this if you don't know what You are doing.

HP856Xx.**save_trace**(*trace*, *number*)

Saves the selected trace in the specified trace register.

Parameters

- **trace** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **number** (*int*) – Storage location from 0 … 7 where to store the trace

```
instr.preset()
instr.center_frequency = 300e6
instr.span = 20e6

instr.save_trace(Trace.A, 7)
instr.preset()

# reload - at 7 stored trace - to Trace B
instr.recall_trace(Trace.B, 7)
```

HP856Xx.**recall_trace**(*trace*, *number*)

Recalls previously saved trace data to the display. See *save_trace()*. Either as Trace A or Trace B.

Parameters

- **trace** (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B
- **number** (*int*) – Storage location from 0 … 7 where to store the trace

```
instr.preset()
instr.center_frequency = 300e6
instr.span = 20e6

instr.save_trace(Trace.A, 7)
```

(continues on next page)

(continued from previous page)

```
instr.preset()

# reload - at 7 stored trace - to Trace B
instr.recall_trace(Trace.B, 7)
```

HP856Xx.**clear_write_trace**(*trace*)

Set the chosen trace to clear-write mode. This mode sets each element of the chosen trace to the bottom-screen value; then new data from the detector is put in the trace with each sweep.

```
instr.clear_write_trace('TRA')
instr.clear_write_trace(Trace.A)
```

Parameters

trace (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B

Raises

- **TypeError** – Type isn't 'string'
- **ValueError** – Value is 'TRA' nor 'TRB'

HP856Xx.**subtract_display_line_from_trace_b**()

Subtract the display line from trace B and places the result in dBm (when in log mode) in trace B, which is then set to view mode.

In linear mode, the results are in volts.

HP856Xx.**exchange_traces**()

Exchange the contents of trace A with those of trace B.

If the traces are in clear-write or max-hold mode, the mode is changed to view. Otherwise, the traces remain in their initial mode.

HP856Xx.**blank_trace**(*trace*)

Blank the chosen trace from the display. The current contents of the trace remain in the trace but are not updated.

```
instr.blank_trace('TRA')
instr.blank_trace(Trace.A)
```

Parameters

trace (*str*) – A representation of the trace, either from *Trace* or use 'TRA' for Trace A or 'TRB' for Trace B

Raises

- **TypeError** – Type isn't 'string'
- **ValueError** – Value is 'TRA' nor 'TRB'

HP856Xx.**trace_a_minus_b_plus_dl_enabled**

Control subtraction of trace B from trace A and addition to the display line, and stores the result in dBm (when in log mode) in trace A. When in linear mode, the result is in volts. If trace A is in clear-write or max-hold mode, this function is continuous. When this function is active, an "M" appears on the left side of the display.

Type: `bool`

> **Warning**
>
> The displayed amplitude of each trace element falls in one of 600 data points. There are 10 points of over-range, which corresponds to one-sixth of a division Kg of overrange. When adding or subtracting trace data, any results exceeding this limit are clipped at the limit.

HP856Xx.**trace_a_minus_b_enabled**

Control subtraction of the contents of trace B from trace A. It places the result, in dBm (when in log mode), in trace A. When in linear mode, the result is in volts. If trace A is in clear-write or max-hold mode, this function is continuous. When AMB is active, an "M" appears on the left side of the display. `trace_a_minus_b_plus_dl` overrides AMB.

Type: `bool`

> **Warning**
>
> The displayed amplitude of each trace element falls in one of 600 data points. There are 10 points of over-range, which corresponds to one-sixth of a division Kg of overrange. When adding or subtracting trace data, any results exceeding this limit are clipped at the limit.

Marker

HP856Xx.**search_peak**(*mode*)

Place a marker on the highest point on a trace, the next-highest point, the next-left peak, or the next-right peak. The default is 'HI' (highest point). The trace peaks must meet the criteria of the marker threshold and peak excursion functions in order for a peak to be found. See also the `peak_threshold` and *peak_excursion* commands.

Parameters
mode (*str*) – Takes 'HI', 'NH', 'NR', 'NL' or the enumeration *PeakSearchMode*

```
instr.search_peak('NL')
instr.search_peak(PeakSearchMode.NextHigh)
```

HP856Xx.**marker_amplitude**

Get the amplitude of the active marker. If no marker is active, MKA places a marker at the center of the trace and returns that amplitude value. In the *amplitude_unit()* unit.

Type: `float`

```
level = instr.marker_amplitude
unit = instr.amplitude_unit
print("Level: %f %s" % (level, unit))
```

HP856Xx.**set_marker_to_center_frequency**()

Set the center frequency to the frequency value of an active marker.

HP856Xx.**marker_delta**

Control a second marker on the trace. The parameter value specifies the distance in frequency or time (when in zero span) between the two markers. If queried - returns the frequency or time of the second marker.

Type: `float`

```
# place second marker 1 MHz apart from the first marker
instr.marker_delta = 1e6

# print frequency of second marker in case it got moved automatically
print(instr.marker_delta)
```

HP856Xx.**marker_frequency**

Control the frequency of the active marker. Default units are in Hertz.

Type: `float`

```
# place marker no. 1 at 100 MHz
instr.marker_frequency = 100e6

# print frequency of the marker in case it got moved automatically
print(instr.marker_frequency)
```

(dynamic)

HP856Xx.**set_marker_minimum**()

Place an active marker on the minimum signal detected on a trace.

HP856Xx.**marker_noise_mode_enabled**

Control the detector mode to sample and compute the average of 32 data points (16 points on one side of the marker, the marker itself, and 15 points on the other side of the marker). This average is corrected for effects of the log or linear amplifier, bandwidth shape factor, IF detector, and resolution bandwidth. If two markers are on (whether in 'marker_delta' mode or 1/marker delta mode), 'marker_noise_mode_enabled' works on the active marker and not on the anchor marker. This allows you to measure signal-to-noise density directly. To query the value, use the 'marker_amplitude' command.

Type: `bool`

```
# activate signal-to-noise density mode
instr.marker_noise_mode_enabled = True

# get noise density by `marker_amplitude`
print("Signal-to-noise density: %d dbm / Hz" % instr.marker_amplitude)
```

HP856Xx.**deactivate_marker**(*all_markers=False*)

Turn off the active marker or, if specified, turn off all markers.

Parameters

all_markers (*bool*) – If True the call deactivates all markers, if false only the currently active marker (optional)

```
# place first marker at 300 MHz
instr.marker_frequency = 300e6

# place second marker 2 MHz apart from first
instr.marker_delta = 2e6

# deactivate active marker (delta marker)
instr.deactivate_marker()
```

(continues on next page)

(continued from previous page)

```
# deactivate all markers
instr.deactivate_marker(all_markers=True)
```

HP856Xx.**marker_threshold**

Control the minimum amplitude level from which a peak on the trace can be detected. The default value is -130 dBm. See also the *peak_excursion* command. Any portion of a peak that falls below the peak threshold is used to satisfy the peak excursion criteria. For example, a peak that is equal to 3 dB above the threshold when the peak excursion is equal to 6 dB will be found if the peak extends an additional 3 dB or more below the threshold level. Maximum 30 db to minimum -200 db.

Type: `signed int`

```
instr.marker_threshold = -70
if instr.marker_threshold > -80:
    pass
```

HP856Xx.**peak_excursion**

Control what constitutes a peak on a trace. The chosen value specifies the amount that a trace must increase monotonically, then decrease monotonically, in order to be a peak. For example, if the peak excursion is 10 dB, the amplitude of the sides of a candidate peak must descend at least 10 dB in order to be considered a peak (see Figure 5-4) The default value is 6 dB. In linear mode, enter the marker peak excursion as a unit-less number. Any portion of a peak that falls below the peak threshold is also used to satisfy the peak excursion criteria. For example, a peak that is equal to 3 dB above the threshold when the peak excursion is equal to 6 dB will be found if the peak extends an additional 3 dB or more below the threshold level.

Type: `float`

```
instr.peak_excursion = 2
if instr.peak_excursion == 2:
    pass
```

HP856Xx.**set_marker_to_reference_level**()

Set the reference level to the amplitude of an active marker.

If no marker is active, 'marker_to_reference_level' places a marker at the center of the trace and uses that marker amplitude to set the reference level.

HP856Xx.**set_marker_delta_to_span**()

Set the frequency span equal to the frequency difference between two markers on a trace.

The start frequency is set equal to the frequency of the left- most marker and the stop frequency is set equal to the frequency of the right-most marker.

HP856Xx.**set_marker_to_center_frequency_step_size**()

Set the center frequency step-size equal to the frequency value of the active marker.

HP856Xx.**marker_time**

Control the marker's time value. Default units are seconds.

Type: `float`

```
# set marker at sweep time corresponding second two
instr.marker_time = 2

if instr.marker_time == 2:
    pass
```

HP856Xx.**marker_signal_tracking_enabled**

Control whether the center frequency follows the active marker.

This is done after every sweep, thus maintaining the marker value at the center frequency. This allows you to "zoom in" quickly from a wide span to a narrow one, without losing the signal from the screen. Or, use 'marker_signal_tracking_enabled' to keep a slowly drifting signal centered on the display. When this function is active, a "K" appears on the left edge of the display.

Type: `bool`

Diagnostic Values

HP856Xx.**sampling_frequency**

Get the sampling oscillator frequency corresponding to the current start frequency. Diagnostic Attribute

Type: `float`

HP856Xx.**lo_frequency**

Get the first local oscillator frequency corresponding to the current start frequency. Diagnostic Attribute

Type: `float`

HP856Xx.**mroll_frequency**

Get the main roller oscillator frequency corresponding to the current start frequency, except then the resolution bandwidth is less than or equal to 100 Hz.

Diagnostic Attribute

Type: `float`

HP856Xx.**oroll_frequency**

Get the offset roller oscillator frequency corresponding to the current start frequency, except when the resolution bandwidth is less than or equal to 100 Hz.

Diagnostic Attribute

Type: `float`

HP856Xx.**xroll_frequency**

Get the transfer roller oscillator frequency corresponding to the current start frequency, except when the resolution bandwidth is less than or equal to 100 Hz.

Diagnostic Attribute

Type: `float`

HP856Xx.**sampler_harmonic_number**

Get the sampler harmonic number corresponding to the current start frequency.

Diagnostic Attribute

Type: `int`

Sweep

HP856Xx.**sweep_single = <function HP856Xx.sweep_single>**

HP856Xx.**sweep_time**

Control the sweep time. This is normally a coupled function which is automatically set to the optimum value allowed by the current instrument settings. Alternatively, you may specify the sweep time. Note that when the specified sweep time is too fast for the current instrument settings, the instrument is no longer calibrated and the message 'MEAS UNCAL' appears on the display. The sweep time cannot be adjusted when the resolution bandwidth is set to 10 Hz, 30 Hz, or 100 Hz.

Type: `str, float`

Real from 50E—3 to 100 when the span is greater than 0 Hz; 50E—6 to 60 when the span equals 0 Hz. When the resolution bandwidth is <100 Hz, the sweep time cannot be adjusted.

HP856Xx.**sweep_couple**

Control the sweep couple mode which is either a stimulus-response or spectrum-analyzer auto-coupled sweep time. In stimulus-response mode, auto-coupled sweep times are usually much faster for swept-response measurements. Stimulus-response auto-coupled sweep times are typicallly valid in stimulus-response measurements when the system's frequency span is less than 20 times the bandwidth of the device under test.

Type: `str` or *SweepCoupleMode*

HP856Xx.**sweep_output**

Control the sweep-related signal that is available from J8 on the rear panel. FAV provides a dc ramp of 0.5V/GHz. RAMP provides a 0—10 V ramp corresponding to the sweep ramp that tunes the first local oscillator (LO). For the HP 8561B, in multiband sweeps one ramp is provided for each frequency band.

Type: `str` or *SweepOut*

HP856Xx.**set_continuous_sweep = <function HP856Xx.set_continuous_sweep>**

HP856Xx.**trigger_sweep**()

Command the spectrum analyzer to take one full sweep across the trace display. Commands following TS are not executed until after the analyzer has finished the trace sweep. This ensures that the instrument is set to a known condition before subsequent commands are executed.

Normalization

HP856Xx.**normalize_trace_data_enabled**

Control the normalization routine for stimulus-response measurements. This function subtracts trace B from trace A, offsets the result by the value of the normalized reference position (*normalized_reference_level*), and displays the result in trace A. 'normalize_trace_data_enabled' is intended for use with the *store_open()* and *store_short()* or *store_thru()* commands. These functions are used to store a reference trace into trace B. Refer to the respective command descriptions for more information. Accurate normalization occurs only if the reference trace and the measured trace are on-screen. If any of these traces are off-screen, an error message will be displayed. If the error message ERR 903 A > DLMT is displayed, the range level (RL) can be adjusted to move the measured response within the displayed measurement range of the analyzer. If ERR 904 B > DLMT is displayed, the calibration is invalid and a thru or open/short calibration must be performed. If active (ON), the 'normalize_trace_data' command is automatically turned off with an instrument preset (IP) or at power on.

Type: `bool`

HP856Xx.**normalized_reference_level**

Control the normalized reference level. It is intended to be used with the `normalize_trace_data` command. When using 'normalized_reference_level', the input attenuator and IF step gains are not affected. This function is a trace-offset function enabling the user to offset the displayed trace without introducing hardware-switching errors into the stimulus-response measurement. The unit of measure for 'normalized_reference_level' is dB. In absolute power mode (dBm), reference level (*reference_level*) affects the gain and RF attenuation settings of the instrument, which affects the measurement or dynamic range. In normalized mode (relative power or dB-measurement mode), NRL offsets the trace data on-screen and does not affect the instrument gain or attenuation settings. This allows the displayed normalized trace to be moved without decreasing the measurement accuracy due to changes in gain or RF attenuation. If the measurement range must be changed to bring trace data on-screen, then the range level should be adjusted. Adjusting the range-level normalized mode has the same effect on the instrument settings as does reference level in absolute power mode (normalize off).

Type: `int`

```
# reference level in case of normalization to -30 DB
instr.normalized_reference_level = -30

if instr.normalized_reference_level == -30:
    pass
```

HP856Xx.**normalized_reference_position**

Control the normalized reference-position that corresponds to the position on the graticule where the difference between the measured and calibrated traces resides. The dB value of the normalized reference-position is equal to the normalized reference level. The normalized reference-position may be adjusted between 0.0 and 10.0, corresponding to the bottom and top graticule lines, respectively.

Type: `float`

```
instr.normalized_reference_position = 5.5

if instr.normalized_reference_position == 5.5:
    pass
```

Open/Short Calibration (Reflection)

HP856Xx.**recall_open_short_average**()

Set the internally stored open/short average reference trace into trace B. The instrument state is also set to the stored open/short reference state.

```
instr.preset()
instr.sweep_single()
instr.start_frequency = 300e3
instr.stop_frequency = 1e9

instr.source_power_enabled = True
instr.sweep_couple = SweepCoupleMode.StimulusResponse
instr.source_peak_tracking()

input("CONNECT OPEN. PRESS CONTINUE WHEN READY TO STORE.")
instr.trigger_sweep()
instr.done()
instr.store_open()
```

(continues on next page)

(continued from previous page)

```
input("CONNECT SHORT. PRESS CONTINUE WHEN READY TO STORE AND AVERAGE.")
instr.trigger_sweep()
instr.done()
instr.store_short()

input("RECONNECT DUT. PRESS CONTINUE WHEN READY.")
instr.trigger_sweep()
instr.done()

instr.normalize = True

instr.trigger_sweep()
instr.done()

instr.normalized_reference_position = 8
instr.trigger_sweep()

instr.preset()
# demonstrate recall of open/short average trace
instr.recall_open_short_average()
instr.trigger_sweep()
```

HP856Xx.**store_open**()

Save the current instrument state and trace A into nonvolatile memory.

This command must be used in conjunction with the *store_short()* command and must precede the *store_short()* command. The data obtained during the store open procedure is averaged with the data obtained during the *store_short()* procedure to provide an open/short calibration. The instrument state (that is, instrument settings) must not change between the *store_open()* and *store_short()* operations in order for the open/short calibration to be valid. Refer to the *store_short()* command description for more information.

HP856Xx.**store_short**()

Take currently displayed trace A data and averages this data with previously stored open data, and stores it in trace B.

This command is used in conjunction with the *store_open()* command and must be preceded by it for proper operation. Refer to the *store_open()* command description for more information. The state of the open/short average trace is stored in state register #8.

Thru Calibration

HP856Xx.**store_thru**()

Store a thru-calibration trace into trace B and into the nonvolatile memory of the spectrum analyzer.

The state of the thru information is stored in state register #9.

HP856Xx.**recall_thru**()

Recalls the internally stored thru-reference trace into trace B.

The instrument state is also set to the stored thru-reference state.

HP8560A Specific Attributes & Methods

class pymeasure.instruments.hp.**HP8560A**(*adapter*, *name='Hewlett-Packard HP8560A'*, ***kwargs*)

Bases: HP856Xx

Represents the HP 8560A Spectrum Analyzer and provides a high-level interface for interacting with the instrument.

```
from pymeasure.instruments.hp import HP8560A
from pymeasure.instruments.hp.hp856Xx import AmplitudeUnits

sa = HP8560A("GPIB::1")

sa.amplitude_unit = AmplitudeUnits.DBUV
sa.start_frequency = 299.5e6
sa.stop_frequency = 300.5e6

print(sa.marker_amplitude)
```

activate_source_peak_tracking()

Activate a routine which automatically adjusts both the coarse and fine-tracking adjustments to obtain the peak response of the tracking generator on the spectrum-analyzer display. Tracking peak is not necessary for resolution bandwidths greater than or equal to 300 kHz. A thru connection should be made prior to peaking in order to ensure accuracy.

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_leveling_control

Control if internal or external leveling is used with the built-in tracking generator. Takes either 'INT', 'EXT' or members of enumeration SourceLevelingControlMode

Type: str

```
instr.preset()
instr.sweep_single()
instr.center_frequency = 300e6
instr.span = 1e6

instr.source_power = -5

instr.trigger_sweep()
instr.source_leveling_control = SourceLevelingControlMode.External

if ErrorCode(900) in instr.errors:
    print("UNLEVELED CONDITION. CHECK LEVELING LOOP.")
```

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_power

Control the built-in tracking generator's output power.

Type: `str, float`

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_power_enabled

Set the built-in tracking generator on or off. See *source_power*

property source_power_offset

Control the offset of the displayed power of the built-in tracking generator so that it is equal to the measured power at the input of the spectrum analyzer. This function may be used to take into account system losses (for example, cable loss) or gains (for example, preamplifier gain) reflecting the actual power delivered to the device under test.

Type: `int`

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_power_step

Control the step size of the source power level, source power offset, and power-sweep range functions. Range: 0.1 … 12.75 DB with 0.05 steps.

Type: `float`

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_power_sweep

Control the power-sweep function, where the output power of the tracking generator is swept over the power-sweep range chosen. The starting source power level is set using the *source_power* command. The output power of the tracking generator is swept according to the sweep rate of the spectrum analyzer.

Type: `str, float`

> **Note**
>
> Only available with an HP 8560A Option 002.

property source_power_sweep_enabled

Set the power sweep active or inactive. See *source_power_sweep*.

property tracking_adjust_coarse

Control the coarse adjustment to the frequency of the built-in tracking-generator oscillator. Once enabled,

this adjustment is made in digital-to-analogconverter (DAC) values from 0 to 255. For fine adjustment, refer to the *tracking_adjust_fine* command description.

Type: int

> **Note**
>
> Only available with an HP 8560A Option 002.

property tracking_adjust_fine

Control the fine adjustment of the frequency of the built-in tracking-generator oscillator. Once enabled, this adjustment is made in digital-to-analogconverter (DAC) values from 0 to 255. For coarse adjustment, refer to the *tracking_adjust_coarse* command description.

Type: int

> **Note**
>
> Only available with an HP 8560A Option 002.

HP8561B Specific Attributes & Methods

class pymeasure.instruments.hp.**HP8561B**(*adapter*, *name='Hewlett-Packard HP8561B'*, ***kwargs*)

Bases: HP856Xx

Represents the HP 8561B Spectrum Analyzer and provides a high-level interface for interacting with the instrument.

```
from pymeasure.instruments.hp import 8561B
from pymeasure.instruments.hp.hp856Xx import AmplitudeUnits

sa = HP8560A("GPIB::1")

sa.amplitude_unit = AmplitudeUnits.DBUV
sa.start_frequency = 6.4e9
sa.stop_frequency = 6.5e9

print(sa.marker_amplitude)
```

property conversion_loss

Control the compensation for losses outside the instrument when in external mixer mode (such as losses within connector cables, external mixers, etc.). 'conversion_loss' specifies the mean conversion loss for the current harmonic band. In a full frequency band (such as band K), the mean conversion loss is defined as the minimum loss plus the maximum loss for that band divided by two. Adjusting for conversion loss allows the system to remain calibrated (that is, the displayed amplitude values have the conversion loss incorporated into them). The default value for any band is 30 dB. The spectrum analyzer must be in external-mixer mode in order for this command to work. When in internal-mixer mode, querying 'conversion_loss' returns a zero.

property harmonic_number_lock

Control the lock to a chosen harmonic so only that harmonic is used to sweep an external frequency band. To select a frequency band, use the 'fullband' command; it selects an appropriate harmonic for the desired band. To change the harmonic number, use 'harmonic_number_lock'. Note that 'harmonic_number_lock'

also works in internal-mixing modes. Once 'fullband' or 'harmonic_number_lock' are set, only center frequencies and spans that fall within the frequency band of the current harmonic may be entered. When the 'set_full_span' command is activated, the span is limited to the frequency band of the selected harmonic.

property harmonic_number_lock_enabled

Set the harmonic number locking active or inactive. See *harmonic_number_lock*.

property mixer_bias

Set the bias for an external mixer that requires diode bias for efficient mixer operation. The bias, which is provided on the center conductor of the IF input, is activated when MBIAS is executed. A "+" or "—" appears on the left edge of the spectrum analyzer display, indicating that positive or negative bias is on. When the bias is turned off, MBIAS is set to 0. Default units are in milliamps.

property mixer_bias_enabled

Control the bias for an external mixer. See *mixer_bias*.

property mixer_mode

Control the mixer mode. Select either the internal mixer or supply an external mixer. Takes enum 'MixerMode' or string 'INT', 'EXT'

peak_preselector()

Peaks the preselector in the HP 8561B Spectrum Analyzer.

Make sure the entire frequency span is in high band, set the desired trace to clear-write mode, place a marker on a desired signal, then execute PP. The peaking routine zooms to zero span, peaks the preselector tracking, then returns to the original position. To read the new preselector peaking number, use the PSDAC command. Commands following PP are not executed until after the analyzer has finished peaking the preselector.

property preselector_dac_number

Control the preselector peak DAC number. For use with an HP 8561B Spectrum Analyzer.

Type: `int`

set_fullband(*band*)

Select a commonly-used, external-mixer frequency band, as shown in the table. The harmonic lock function *harmonic_number_lock* is also set; this locks the harmonic of the chosen band. External-mixing functions are not available with an HP 8560A Option 002. Takes frequency band letter as string.

Table 2: Title

Frequency Band	Frequency (GHz)	Range	Mixing Harmonic	Conversion Loss
K	18.0 — 26.5		6	30 dB
A	26.5 — 40.0		8	30 dB
Q	33.0—50.0		10	30 dB
U	40.0—60.0		10	30 dB
V	50.0—75.0		14	30 dB
E	60.0—-90.0		16	30 dB
W	75.0—110.0		18	30 dB
F	90.0—140.0		24	30 dB
D	110.0—170.0		30	30 dB
G	140.0—220.0		36	30 dB
Y	170.0—260.0		44	30 dB
J	220.0—325.0		54	30 dB

set_signal_identification_to_center_frequency()

Set the center frequency to the frequency obtained from the command SIGID.

SIGID must be in AUTO mode and have found a valid result for this command to execute properly. Use SIGID on signals greater than 18 GHz {i.e., in external mixing mode). SIGID and IDCF may also be used on signals less than 6.5 GHz in an HP 8561B.

property signal_identification

Control the signal identification for identifying signals for the external mixing frequency bands. Two signal identification methods are available. AUTO employs the image response method for locating correct mixer responses. Place a marker on the desired signal, then activate signal_identification = 'AUTO'. The frequency of a correct response appears in the active function block. Use this mode before executing the signal_identification_to_center_frequency() command. The second method of signal identification, 'MAN', shifts responses both horizontally and vertically. A correct response is shifted horizontally by less than 80 kHz. To ensure accuracy in MAN mode, limit the frequency span to less than 20 MHz. Where True = manual mode is active and False = auto mode is active or 'signal_identification' is off.

property signal_identification_frequency

Measure the frequency of the last identified signal. After an instrument preset or an invalid signal identification, IDFREQ returns a "0".

unlock_harmonic_number()

Unlock the harmonic number, allowing you to select frequencies and spans outside the range of the locked harmonic number.

Also, when HNUNLK is executed, more than one harmonic can then be used to sweep across a desired span. For example, sweep a span from 18 GHz to 40 GHz. In this case, the analyzer will automatically sweep first using 6—, then using 8—.

Enumerations

class pymeasure.instruments.hp.hp856Xx.**AmplitudeUnits**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the amplitude units.

AUTO = 'AUTO'

Automatic Unit (Usually derives to 'DBM')

DBM = 'DBM'

DB over millit Watt

DBMV = 'DBMV'

DB over milli Volt

DBUV = 'DBUV'

DB over micro Volt

MANUAL = 'MAN'

Manual Mode

V = 'V'

Volts

W = 'W'

Watt

class pymeasure.instruments.hp.hp856Xx.**MixerMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Mixer Mode of the HP8561B.

External = 'EXT'

Mixer Mode External

Internal = 'INT'

Mixer Mode Internal

class pymeasure.instruments.hp.hp856Xx.**Trace**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent either Trace A or Trace B.

A = 'TRA'

Trace A

B = 'TRB'

Trace B

class pymeasure.instruments.hp.hp856Xx.**CouplingMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Coupling Mode.

AC = 'AC'

AC

DC = 'DC'

DC

class pymeasure.instruments.hp.hp856Xx.**DemodulationMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Demodulation Mode.

Amplitude = 'AM'

Amplitude Modulation

Frequency = 'FM'

Frequency Modulation

Off = 'OFF'

Demodulation Off

class pymeasure.instruments.hp.hp856Xx.**DetectionModes**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Detection Modes.

NegativePeak = 'NEG'

Negative Peak Detection

Normal = 'NRM'

Normal Peak Detection

PositivePeak = 'POS'

Positive Peak Detection

Sample = 'SMP'

Sampl Mode Detection

class pymeasure.instruments.hp.hp856Xx.**ErrorCode**(*code*)

Bases: object

Class to decode error codes from the spectrum analyzer.

class pymeasure.instruments.hp.hp856Xx.**FrequencyReference**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the frequency reference source.

External = 'EXT'

External Frequency Standard

Internal = 'INT'

Internal Frequency Reference

class pymeasure.instruments.hp.hp856Xx.**PeakSearchMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Marker Peak Search Mode.

High = 'HI'

Place marker to the highest value on the trace

NextHigh = 'NH'

Place marker to the next highest value on the trace

NextLeft = 'NL'

Place marker to the next peak to the left

NextRight = 'NR'

Place marker to the next peak to the right

class pymeasure.instruments.hp.hp856Xx.**SourceLevelingControlMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the Source Leveling Control Mode of the HP8560A.

External = 'EXT'

Source Leveling Control Mode External

Internal = 'INT'

Source Leveling Control Mode Internal

class pymeasure.instruments.hp.hp856Xx.**StatusRegister**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Enumeration to represent the Status Register.

COMMAND_COMPLETE = 16

Any command is completed

END_OF_SWEEP = 4

Set when any sweep is completed

ERROR_PRESENT = 32

Set when error present

MESSAGE = 2

Set when display message appears

NA = 8

Unused but sometimes set

NONE = 0

No Interrupts can interrupt the program sequence

RQS = 64

Request Service

TRIGGER = 1

Trigger is activated

class pymeasure.instruments.hp.hp856Xx.**SweepCoupleMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration.

SpectrumAnalyzer = 'SA'

Stimulus Response

StimulusResponse = 'SR'

Spectrum Analyeze

class pymeasure.instruments.hp.hp856Xx.**SweepOut**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration.

Fav = 'FAV'

DC Ramp 0.5V / GHz

Ramp = 'RAMP'

0 - 10V Ramp

class pymeasure.instruments.hp.hp856Xx.**TriggerMode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the different trigger modes

External = 'EXT'

External Mode

Free = 'FREE'

Free Running

Line = 'LINE'

Line Mode

Video = 'VID'

Video Mode

class pymeasure.instruments.hp.hp856Xx.**WindowType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: StrEnum

Enumeration to represent the different window mode for FFT functions

Flattop = 'FLATTOP'

Flattop provides optimum amplitude accuracy

Hanning = 'HANNING'

Hanning provides an amplitude accuracy/frequency resolution compromise

Uniform = 'UNIFORM'

Uniform provides equal weighting of the time record for measuring transients.

7.28.7 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*, *name='Hewlett-Packard HP8657B'*, ***kwargs*)

Bases: *Instrument*

Represents the Hewlett Packard 8657B signal generator and provides a high-level interface for interacting with the instrument.

class Modulation(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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

7.28.8 HP 8753E Vector Network Analyzer

class pymeasure.instruments.hp.**HP8753E**(*adapter=None*, *name=None*, *min_frequency=30000.0*, *max_frequency=6000000000.0*, *min_power=-70*, *max_power=10*, ***kwargs*)

Bases: *Instrument*

Represents the HP8753E Vector Network Analyzer and provides a high-level interface for taking sweeps of the scattering or measurement parameters.

Parameters

- **name** – Optional set the name of the instrument (str).
- **min_frequency** – Optional set the minimum validator for *HP8753E.start_frequency*, *HP8753E.stop_frequency*, *HP8753E.span_frequency*, *HP8753E.center_frequency* on initialization (float in Hz).
- **max_frequency** – Optional set the maximum validator for *HP8753E.start_frequency*, *HP8753E.stop_frequency*, *HP8753E.span_frequency*, *HP8753E.center_frequency* on initialization (float in Hz).
- **min_power** – Optional set the minimum validator for *HP8753E.power* on initialization (float in dBm).
- **max_power** – Optional set the maximum validator for *HP8753E.power* on initialization (float in dBm).

property IFBW

Control the IF Bandwidth of the instrument for a scan sweep. (int from the set [10, 30, 100, 300, 1000, 3000, 3700, 6000]).

property averages

Control the number of averages for a scan sweep. (int truncated from 1 to 999).

property averaging_enabled

Control whether or not averaging is enabled. (boolean)

averaging_restart()

Restart sweep averaging.

property center_frequency

Control the center frequency in Hz (float).(dynamic)

property correction_enabled

Control whether or not correction is enabled. (boolean)

property data_complex

Get the complex s-parameter measurements from the last scan. This function is blocking until it is completed.

Parameters
timeout – Optional value in seconds to wait until a timeout occurs.

Raises
TimeoutError – If the VNA fails to complete the sweep before timeout occurs.

Returns
An array of s-parameters for the measurement_parameter.

Return type
numpy.ndarray

emit_beep()

Make the VNA emit a beep

property frequencies

Get a list of frequencies from the last scan.

Returns
An array of frequencies sized by number of points in sweep)

Return type
numpy.ndarray

property fw

Get the firmware of the instrument.

property id

Get the identification of the instrument (list of strings)

property manu

Get the manufacturer of the instrument.

property measuring_parameter

Get the active Scattering or Measuring Parameter being measured. (str from ['S11', 'S21', 'S12', 'S22', 'A', 'B', 'R'])

property model

Get the model of the instrument.

property options

Get the installed options for the instrument

property output_enabled

Control RF Output State (boolean)

property power

Control the output RF power of the instrument's active port. (float).(dynamic)

reset()

Reset the instrument. May cause RF Output power to be enabled!

scan(*timeout=10*)

Initiates a scan with the number of averages specified and blocks until the operation is complete.

Parameters
timeout – Optional value in seconds to add to the sweep time before timeout occurs.

Raises
TimeoutError – If the VNA fails to complete the sweep before timeout occurs.

Returns
None.

property scan_points

Control the number of points used for a scan sweep. From the set [3, 11, 21, 26, 51, 101, 201, 401, 801, 1601]

scan_single()

Initiates a single scan or N scans averaged based on averaging This function is not blocking.

set_fixed_frequency(*frequency*)

Set the sweep to be a fixed frequency in Hz.

set_sweep_time_fastest()

Set instrument scan sweep time to select fastest possible time.

shutdown()

Shutdown - Disables RF Output.

property sn

Get the serial number for the instrument

property span_frequency

Control the span of the sweep frequency in Hz (float).(dynamic)

property start_frequency

Control the start frequency in Hz. (float).(dynamic)

property stop_frequency

Control the stop frequency in Hz. (float).(dynamic)

property sweep_time

Control the sweep time in seconds. (float truncated from 0.0 to 36_400.0)

property trigger_continuous

Control Sweep Scan Trigger State (boolean)

property trigger_hold

Control Sweep Scan Trigger State (boolean)

7.28.9 HP 11713A Attenuator/Switch Driver

class pymeasure.instruments.hp.**HP11713A**(*adapter*, *name='Hewlett-Packard HP11713A'*, ***kwargs*)

Bases: *Instrument*

Represents the HP 11713A Switch and Attenuator Driver and provides a high-level interface for interacting with the instrument.

Usually an attenuator is hooked to either X or Y or X and Y. To ease the control of the attenuator driver you have the possibility to set an attenuator type via the attribute 'ATTENUATOR_X' or 'ATTENUATOR_Y'. The hp11713a keeps different default attenuator mappings. After setting the attenuator type you are able to use the methods 'attenuation_x' and/or 'attenuation_y' to set the switch driver to the correct value for the specified attenuation. The attenuation values are rounded.

```
from pymeasure.instruments.hp import HP11713A
from pymeasure.instruments.hp.hp11713a import Attenuator_110dB

sd = HP11713A("GPIB::1")

sd.ATTENUATOR_Y = Attenuator_110dB
sd.attenuation_y(10)
sd.ch_0.enabled = True
```

channels

Channels
ch_0: *SwitchDriverChannel*, ch_1: *SwitchDriverChannel*, ch_2: *SwitchDriverChannel*, ch_3: *SwitchDriverChannel*, ch_4: *SwitchDriverChannel*, ch_5: *SwitchDriverChannel*, ch_6: *SwitchDriverChannel*, ch_7: *SwitchDriverChannel*, ch_8: *SwitchDriverChannel*

attenuation_x(*attenuation*)

Set switches according to the attenuation in dB for X

The set attenuation will be rounded to the next available step. An attenuation mapping has to be set in before e.g.

```
from pymeasure.instruments.hp.hp11713a import HP11713A, Attenuator_110dB

instr.ATTENUATOR_X = Attenuator_110dB
instr.attenuation_x(10)
```

attenuation_y(*attenuation*)

Set switches according to the attenuation in dB for Y

The set attenuation will be rounded to the next available step. An attenuation mapping has to be set in before e.g.

```
from pymeasure.instruments.hp.hp11713a import HP11713A, Attenuator_110dB

instr.ATTENUATOR_Y = Attenuator_110dB
instr.attenuation_y(10)
```

deactivate_all()

Deactivate all switches to polarity 'B'.

class pymeasure.instruments.hp.hp11713a.**SwitchDriverChannel**(*parent*, *id*)

Bases: *Channel*

property enabled

Set this channel to the polarity 'A' for True and 'B' for False.

pymeasure.instruments.hp.hp11713a.**Attenuator_11dB**

Mapping of logical values for use with 0 - 11 dB attenuators

pymeasure.instruments.hp.hp11713a.**Attenuator_110dB**

Mapping of logical values for use with 0 - 110 dB attenuators

pymeasure.instruments.hp.hp11713a.**Attenuator_70dB_3_Section**

Mapping of logical values for use with 0 - 70 dB attenuators with 3 switching sections

pymeasure.instruments.hp.hp11713a.**Attenuator_70dB_4_Section**

Mapping of logical values for use with 0 - 70 dB attenuators with 4 switching sections

7.28.10 HP 437B Power Meter

class pymeasure.instruments.hp.**HP437B**(*adapter*, *name='Hewlett-Packard HP437B'*, ***kwargs*)

Bases: *Instrument*

Represents the HP437B Power Meters.

> **Note**
>
> Most command descriptions are taken from the document: 'Operating Manual 437B Power Meter'

activate_auto_range()

The power meter divides each sensor's power range into 5 ranges of 10 dB each. Range 1 is the most sensitive (lowest power levels), and Range 5 is the least sensitive (highest power levels). Range 5 can be less than 10 dB if the sensor's power range is less than 50 dB. The range can be set either automatically or manually. 'activate_auto_range' automatically selects the correct range for the current measurement.

property automatic_range_enabled

Control the automatic range. The power meter divides each sensor's power range into 5 ranges of 10 dB each. Range 1 is the most sensitive (lowest power levels), and Range 5 is the least sensitive (highest power levels). The range can be set either automatically or manually.

calibrate(*calibration_factor*)

Calibrate a sensor to the power meter with a 'calibration_factor' in percent.

property calibration_factor

Control the calibration factor of a specific power sensor at a specific input frequency. (A chart or table of CAL FACTOR % versus Frequency is printed on each sensor and an accompanying data sheet.) Calibration factor is entered in percent. Valid entries for 'calibration_factor' range from 1.0 to 150.0%.

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

property display_all_segments_enabled

Set all segments of the display of the power meter active or resume normal state.

property display_enabled

Set the display of the power meter active or inactive.

property display_output

Get the current displayed string of values of the power meter.

```
print(instr.display_output)
-0.23  dB REL
```

property display_user_message

Set a custom user message up to 12 alpha-numerical chars. If the string is empty or None the user message gets disabled.

property duty_cycle

Control the duty cycle for calculation of a pulsed input signal. This function will cause the power meter to report the pulse power of a rectangular pulsed input signal. The allowable range of values for 'duty_cycle' is 0.00001 to 0.99999.

Pulse power, as reported by the power meter, is a mathematical representation of the pulse power rather than an actual measurement. The power meter measures the average power of the pulsed input signal and then divides the measurement by the duty cycle value to obtain a pulse power reading.

property duty_cycle_enabled

Control whether the duty cycle is active or inactive. See *duty_cycle*

property event_status

Get the status byte and Master Summary Status bit.

```
print(instr.request_service_conditions)
StatusRegister.PowerOn|CommandError
```

property filter

Control the filter number for averaging. Setting a value implicitly enables the manual filter mode. Setting a value of 1 basically disables the averaging.

property filter_automatic_enabled

Control the filter mode. By switching over from automatic to manual (true to false) the instrument implicitly keeps (holds) the filter value from the automatic selection.

property frequency

Control the frequency of the input signal. Entering a frequency causes the power meter to select a sensor-specific calibration factor. The allowed range of 'frequency' values is from 0.0001 to 999.9999 GHz with a 100 kHz resolution. The unit is Hz.

property group_trigger_mode

Control the group execute trigger mode. When in remote and addressed to listen, the power meter responds to a Trigger message (the Group Execute Trigger bus command [GET]) according to the programmed mode.

property limit_high

Control the upper limit for the builtin limit checking.

property limit_high_hit

Get if the upper limit check got triggered.

property limit_low

Control the lower limit for the builtin limit checking.

property limit_low_hit

Get if the lower limit check got triggered.

property limits_enabled

Control the limits checking function to allow the power meter to monitor the power level at the sensor and to indicate when that power is outside preset limits.

property linear_display_enabled

Control if the power meter displays or reports the power values in logarithmic or linear units. Set *linear_display_enabled* to 'True' to activate linear value readout.

```
from pymeasure.instruments.hp.hp437b import LogLin

instr.relative_mode_enabled = False
instr.linear_display_enabled = True
```

property measurement_unit

Get the measurement unit the power meter is currently reporting the power values in.

Depends on: *relative_mode_enabled* and attr:*linear_display_enabled*

```
instr.relative_mode_enabled = False
instr.linear_display_enabled = True

print(instr.measurement_unit)
MeasurementUnit.Watts
```

property offset

Control the offset applied to the measured value to compensate for signal gain or loss (for example, to compensate for the loss of a 10 dB directional coupler). Offsets are entered in dB. In case the *offset_enabled* is false this returns automatically 0.0

property offset_enabled

Control the offset being applied.

property operating_mode

Get the operating mode the power meter is currently in.

property power

Measure the power at the power sensor attached to the power meter in the corresponding unit. In case a measurement would be invalid the power meter responds with the value float('nan').

property power_reference_enabled

Control the builtin reference power source 1mW @ 50 MHz.

preset()

Sets the power meter to a known state. Preset conditions are shown in the following table.

Table 3: Preset values

Parameter	Value/Condition
Frequency	50 MHz
Resolution	0.01 dB
Duty Cylce	1.000%, Off
Relative	0 dB, Off
Power Reference	Off
Range	Auto
Unit	dBm
Low Limit	-90.000 dBm
High Limit	+90.000 dBm
Limit Checking	Off
Trigger Mode	Free Run
Group Trigger Mode	Trigger with Delay
Display Function	Display Enable

property range

Control the range to be selected manually. Valid range numbers are 1 through 5. See *automatic_range_enabled* for further information.

property relative_mode_enabled

Control the relative mode. In the relative mode the current measured power value will be used as reference and any further reported value from *power* will refer to this.

reset()

Resets the instrument.

property resolution

Control the resolution of the power meter's measured value. Three levels of resolution can be set: 0.1 dB, 0.01 dB and 0.001 dB or if the selected unit is Watts 1%, 0.1% and 0.001%.

sensor_data_clear(*sensor_id*)

Clear the Sensor Data table of 'sensor_id' previous to entering new values.

sensor_data_read_cal_factor_table(*sensor_id*)

Read the Sensor Data calibration table. See *sensor_data_write_cal_factor_table()* Returns a tuple of frequencies as list and calibration factors as list.

sensor_data_ref_cal_factor(*sensor_id*, *ref_cal_factor*)

Set the power sensor's reference calibration factor to the Sensor Data table.

sensor_data_write_cal_factor_table(*sensor_id*, *frequency_table*, *cal_fac_table*)

Write the 'calibration_table' for 'sensor_id' to the Sensor Data table. And write the reference calibration factor for the 'sensor_id'. Frequency is given in Hz. Calibration factor as percentage.

The power meter's memory contains space for 10 tables, numbered 0—9. Tables 0-7 each contain space for 40 frequency /calibration factor pairs. Tables 8 and 9 each contain space for 80 frequency/calibration factor pairs.

This function clears the sensor table before writing.

Example table:

```
calibration_table = {
    10e6: 100.0,
    1e9: 96.5,
    2e9: 97.0
}

instr.sensor_data_cal_factor_table(0, calibration_table.keys(),
calibration_table.values())
```

sensor_data_write_id_label(*sensor_id*, *label*)

Set a particular power sensor's ID label table to be modified. The sensor ID label must not exceed 7 characters. For example, to identify Sensor Data table #2 with an ID number of 1234567:

```
instr.sensor_data_id_label(2, "1234567")
```

property sensor_type

Set the sensor type connected to the power meter to select the corresponding calibration factor.

```
from pymeasure.instruments.hp.hp437b import SensorType

instr.sensor_type = SensorType.HP_8481A
```

store(*register*)

The power meter can store instrument configurations for recall at a later time. The following information can be stored in the power meter's internal registers:

- reference calibration factor value
- Measurement units (dBm or watts)
- relative value and status (on or off)
- power reference status (on or off)
- calibration factor value
- SENSOR ID (sensor data table selection)
- offset value and status (on or off)
- range (Auto or Set)
- frequency value
- resolution
- duty cycle value and status (on or off)
- Filter (number of readings averaged, auto or manual)
- Limits value and status (on or off)

Registers 1 through 10 are available for storing instrument configurations.

trigger_delay()

Trigger with delay.

Triggering with delay is identical to *trigger_immediate()* except the power meter inserts a settling-time delay before taking the requested measurement. This settling time allows the internal digital filter to be updated with new values to produce valid, accurate measurement results. The trigger with delay command allows time for settling of the internal amplifiers and filters. It does not allow time for power sensor delay. In cases of large power changes, the delay may not be sufficient for complete settling. Accurate readings can be assured by taking two successive measurements for comparison. Once the measurement results are displayed and read onto the bus, the power meter reverts to standby mode.

trigger_immediate()

Trigger immediate.

When the power meter receives the trigger immediate program code, it inputs one more data point into the digital filter, measures the reading from the filter, and then updates the display and HP-IB. (When the trigger immediate command is executed, the internal digital filter is not cleared.) The power meter then waits for the measurement results to be read by the controller. While waiting, the power meter can process most bus commands without losing the measurement results. If the power meter receives a trigger immediate command and then receives the GET (Group Execute Trigger) command, the trigger immediate command will be aborted and a new measurement cycle will be executed. Once the measurement results are read onto the bus, the power meter always reverts to standby/hold mode. Measurement results obtained via trigger immediate are normally valid only when the power meter is in a steady, settled state.

property trigger_mode

Control the trigger mode.

The power meter has two modes of triggered operation; standby mode and free run mode. Standby mode means the power meter is making measurements, but the display and HP-IB are not updated until a trigger command is received. Free run means that Meter takes measurements and updates the display and HP-IB continuously.

zero()

Adjust the power meter's internal circuitry for a zero power indication when no power is applied to the sensor.

> **Note**
>
> Ensure that no power is applied to the sensor while the power meter is zeroing. Any applied RF input power will cause an erroneous reading.

7.28.11 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: *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

Get an object representing the current status of the unit.

status_desc

alias of 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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.

- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the `ask()` method.

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.

7.28.12 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 similar manner

class pymeasure.instruments.hp.**HP6632A**(*adapter*, *name='Hewlett-Packard HP6632A'*, ***kwargs*)

Bases: *HPLegacyInstrument*

Represents the Hewlett Packard 6632A system power supply and provides a high-level interface for interacting with the instrument.

class ERRORS(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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 property 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 `Status`

property voltage

A floating point proptery that controls the output voltage of the device.

(dynamic)

class pymeasure.instruments.hp.**HP6633A**(*adapter*, *name='Hewlett Packard HP6633A'*, ***kwargs*)

Bases: *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*, *name='Hewlett Packard HP6634A'*, ***kwargs*)

Bases: *HP6632A*

Represents the Hewlett Packard 6634A system power supply and provides a high-level interface for interacting with the instrument.

7.29 Inficon

This section contains specific documentation on the Inficon instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.29.1 Inficon SQM-160 multi-film rate/thickness monitor

class pymeasure.instruments.inficon.sqm160.**SQM160**(*adapter*, *name='Inficon SQM-160 thickness monitor'*, *baud_rate=19200*, ***kwargs*)

Bases: *Instrument*

Inficon SQM-160 multi-film rate/thickness monitor.

Uses a quartz crystal sensor to measure rate and thickness in a thin film deposition process. Connection to the device is commonly made through a serial connection (RS232) or optionally via USB or Ethernet.

A command packet always consists of the following:

- 1 Byte: Sync character ('!' appears only at the start of a message).
- **1 Byte: length character obtained from the message length without CRC.**
  A value of 34 is added so that no '!' can occur.
- Command message with variable length.
- 2 Byte: Cyclic Redundancy Check (CRC) checksum.

A response packet always consists of:

- 1 Byte: Sync character ('!' appears only at the start of a message).
- **1 Byte: length character obtained from the message length without CRC.**
  A value of 35 is added.
- 1 Byte: Response status character indicating the status of the command.
- Response message with variable length.
- 2 Byte: Cyclic Redundancy Check (CRC) checksum.

Parameters

- **adapter** – pyvisa resource name of the instrument or adapter instance

- **name** (*string*) – Name of the instrument.
- **baud_rate** (*string*) – Baud rate used by the serial connection.
- **kwargs** – Any valid key-word argument for Instrument

sensor_1

Channel
SensorChannel

sensor_2

Channel
SensorChannel

sensor_3

Channel
SensorChannel

sensor_4

Channel
SensorChannel

sensor_5

Channel
SensorChannel

sensor_6

Channel
SensorChannel

property all_values

Get the current rate (Angstrom/s), Thickness (Angstrom), and frequency (Hz) for each sensor

property average_rate

Get the current average rate in Angstrom per second

property average_thickness

Get the current average thickness in Angstrom

check_set_errors()

Check the errors after setting a property.

property firmware_version

Get the firmware version.

property number_of_channels

Get the number of installed channels

read()

Reads a response message from the instrument.

This method also checks for a correct checksum.

Returns
the response packet

Return type
string

Raises
ConnectionError – if a checksum error is detected or a wrong response status is detected.

property reset_flag
Get the power-up reset flag. It is True only when read first after a power cycle.

reset_system_parameters()
Reset all film and system parameters.

reset_thickness_rate()
Reset the average thickness and rate.

This also sets all active Sensor Rates and Thicknesses to zero

reset_time()
Reset the time of the monitor to zero.

write(*command*)
Write a command to the device.

class pymeasure.instruments.inficon.sqm160.**SensorChannel**(*parent*, *id*)
Bases: *Channel*

Sensor channel for individual rate measurements.

property crystal_life
Get the crystal life value in percent

property frequency
Get the current frequency for a sensor in Hz

property rate
Get the current rate for a sensor in Angstrom per second

property thickness
Get the current thickness for a sensor in Angstrom

7.30 IPG Photonics

This section contains specific documentation on the IPG Photonics instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.30.1 YAR fiber amplifier series

class pymeasure.instruments.ipgphotonics.yar.**YAR**(*adapter*, *name='YAR fiber amplifier'*, ***kwargs*)
Bases: *Instrument*

Communication with the YAR fiber amplifier series by IPG Photonics.

This is the RS232 command set. GPIB has different commands.

class Status(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

check_set_errors()

Check for errors after having set a property.

Called if check_set_errors=True is set for that property.

clear()

Reset all errors.

property current

Measure the diode current in A.

property emission_enabled

Control emission of the amplifier (bool).

property firmware

Get firmware version

property id

Get the model number.

property maximum_case_temperature

Measure the maximum temperature for the optical module in °C.

property minimum_display_power

Measure the minimum displayable output power in W.

property power

Measure current output power in W.(dynamic)

property power_range

Get the power limits in W.

property power_setpoint

Control output power setpoint in W.(dynamic)

read()

Read an instrument answer and check whether it is an error.

property status

Get the current status.

property temperature

Measure case temperature in °C.

property temperature_seed

Measure current seed temperature in °C

property wavelength_temperature

Control temperature in °C for seed wavelength control.

7.31 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*.

7.31.1 Keithley 2000 Multimeter

class pymeasure.instruments.keithley.**Keithley2000**(*adapter*, *name='Keithley 2000 Multimeter'*, ***kwargs*)

Bases: KeithleyBuffer, SCPIUnknownMixin, *Instrument*

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)
```

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

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

Control whether the system status beeper is enabled, which can take the values: `enabled` and `disabled`.

property buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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*)

Configure 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

Get a DC or AC current measurement in Amps, based on the active *mode*.

property current_ac_bandwidth

Control (floating) the AC current detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

property current_ac_digits

Control (integer) the number of digits in the AC current readings, which can take values from 4 to 7.

property current_ac_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the AC current reference value in Amps, which can take values from -3.1 to 3.1 A.

property current_digits

Control (integer) the number of digits in the DC current readings, which can take values from 4 to 7.

property current_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the DC current reference value in Amps, which can take values from -3.1 to 3.1 A.

disable_buffer()

Disable 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

Get a frequency measurement in Hz, based on the active *mode*.

property frequency_aperature

Control (floating) 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

Control (integer) the number of digits in the frequency readings, which can take values from 4 to 7.

property frequency_reference

Control (floating) the frequency reference value in Hz, which can take values from 0 to 15 MHz.

property frequency_threshold

Control (floating) the voltage signal threshold level in Volts for the frequency measurement, which can take values from 0 to 1010 V.

property id

Get the identification of the instrument.

is_buffer_full()

Return 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 resistance, based on a maximum resistance to set the range, and the number of wires utilized.

Parameters

- **max_resistance** – A resistance in ohms to set the resistance range
- **wires** – Number of wires employed (2 or 4)

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

Control (string) the configuration mode for measurements, which can take the values: `current` (DC), `current ac`, `voltage` (DC), `voltage ac`, `resistance` (2-wire), `resistance 4W` (4-wire), `period`, `temperature`, `diode`, and `frequency`.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property period

Get a period measurement in seconds, based on the active *mode*.

property period_aperature

Control (floating) 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

Control (integer) the number of digits in the period readings, which can take values from 4 to 7.

property period_reference

Control (floating) the period reference value in seconds, which can take values from 0 to 1 s.

property period_threshold

Control (floating) the voltage signal threshold level in Volts for the period measurement, which can take values from 0 to 1010 V.

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).

remote()

Places the instrument in the remote state, which is does not need to be explicitly called in general.

remote_lock()

Disables and locks the front panel controls to prevent changes during remote operations. This is disabled by calling *local()*.

reset()

Resets the instrument state.

reset_buffer()

Reset the buffer.

property resistance

Get a resistance measurement in Ohms for both 2-wire and 4-wire configurations, based on the active *mode*.

property resistance_4W_digits

Control (integer) the number of digits in the 4-wire resistance readings, which can take values from 4 to 7.

property resistance_4W_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the 4-wire resistance reference value in Ohms, which can take values from 0 to 120 MOhms.

property resistance_digits

Control (integer) the number of digits in the 2-wire resistance readings, which can take values from 4 to 7.

property resistance_nplc

Control (floating) 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

Control (floating) 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

Control (floating) 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

Get the status byte and Master Summary Status bit.

stop_buffer()

Abort the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property temperature

Get a temperature measurement in Celsius, based on the active *mode*.

property temperature_digits

Control (integer) the number of digits in the temperature readings, which can take values from 4 to 7.

property temperature_nplc

Control (floating) 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

Control (floating) the temperature reference value in Celsius, which can take values from -200 to 1372 C.

property trigger_count

Control (integer) the trigger count, which can take values from 1 to 9,999.

property trigger_delay

Control (floating) the trigger delay in seconds, which can take values from 1 to 9,999,999.999 s.

property voltage

Get a DC or AC voltage measurement in Volts, based on the active *mode*.

property voltage_ac_bandwidth

Control (floating) the AC voltage detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

property voltage_ac_digits

Control (integer) the number of digits in the AC voltage readings, which can take values from 4 to 7.

property voltage_ac_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the AC voltage reference value in Volts, which can take values from -757.5 to 757.5 Volts.

property voltage_digits

Control (integer) the number of digits in the DC voltage readings, which can take values from 4 to 7.

property voltage_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the DC voltage reference value in Volts, which can take values from -1010 to 1010 V.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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(*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.

7.31.2 Keithley 2200 Series Power Supplies

class pymeasure.instruments.keithley.**Keithley2200**(*adapter*, *name='Keithley2200'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Keithley 2200 Power Supply.

ch_1

Channel
PSChannel

ch_2

Channel
PSChannel

ch_3

Channel
PSChannel

class BaseChannelCreator(*cls*, ***kwargs*)

Bases: object

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Bases: *BaseChannelCreator*

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.
- ****kwargs** – Keyword arguments for all children.

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

Bases: *BaseChannelCreator*

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.

- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns

Instance of the created child.

binary_values(*command*, *query_delay=None*, ***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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors()`
Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`property complete`
Get the synchronization bit.

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_enabled`
Control whether the display is enabled.

`property display_text_data`
Control text to be displayed(32 characters).

`static get_channel_pairs(cls)`
Return a list of all the Instrument's channel pairs

`static get_channels(cls)`
Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

`property id`
Get the identification of the instrument.

`property next_error`
Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

`property options`
Get 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()
Reset the instrument.

shutdown()
Brings the instrument to a safe and stable state

property status
Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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.keithley.keithley2200.**PSChannel**(*parent*, *id*)
Bases: *Channel*

Implementation of a Keithley 2200 channel.

property current
Measure the current in Amps.

property current_limit
Control output current in Amps.

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.

property output_enabled
Control the output state.

property power
Measure the power in watts.

property voltage
Measure the voltage in Volts.

property voltage_limit
Control the maximum voltage that can be set.

property voltage_limit_enabled
Control whether the maximum voltage limit is enabled.

property voltage_setpoint
Control output voltage in Volts.

7.31.3 Keithley 2260B DC Power Supply

class pymeasure.instruments.keithley.**Keithley2260B**(*adapter*, *name='Keithley 2260B DC Power Supply'*, *read_termination='\n'*, ***kwargs*)

Bases: *SCPIMixin*, *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)
print(source.power)
print(source.applied)
```

property applied

Control voltage (volts) and current (amps) simultaneously. 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.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Get the current (in Ampere) the dc power supply is putting out.

property current_limit

Control 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. (float)

property enabled

Control whether the output is enabled, see *output_enabled*.

property error

Get the next error of the instrument (list of code and message).

property id

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property output_enabled

Control whether the source is enabled, takes values True or False. (bool)

property power

Get the power (in Watt) the dc power supply is putting out.

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()

Reset the instrument.

shutdown()

Disable output, call parent function

property status

Get the status byte and Master Summary Status bit.

property voltage

Get the voltage (in Volt) the dc power supply is putting out.

property voltage_setpoint

Control 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. (float)

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.31.4 Keithley 2281S power supply and battery simulator / characterizer

class pymeasure.instruments.keithley.**Keithley2281S**(*adapter*, *name='Keithley2281S'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*, KeithleyBuffer

Represents the Keithley 2281S-20-6 power supply and battery simulator / characterizer. Common commands beside *function_mode* and power supply commands should also work for Keithley 2280S power supplies, although this is untested.

bs

Channel
BatterySimulationChannel

bt

Channel
BatteryTestChannel

ps

Channel
PowerSupplyChannel

property buffer_data

Get a pandas dataframe of values from the buffer.

property buffer_points

Control the maximum number of buffer points to store.

characterize(*lower_voltage: float*, *upper_voltage: float*, *charge_current: float*, *memory_slot: int*, *model_voltage_offset: float = 0.05*, *charge_delay: float = 0*)

Convenience function for testing a battery and saving its model to the internal memory.

The device can only discharge the battery at a fixed 1A! If this current is too high, a series resistor has to be used during discharge to limit the current! If the battery is discharged below the lower limit, it will be charged with a 10th of the set charge current till it reaches the lower limit, then the battery profile will be characterized. The function will block until the end of the measurement!

Parameters

- **lower_voltage** (*float*) – discharge end voltage, set this slightly lower (~0.05V) than in normal operation
- **upper_voltage** (*float*) – charge end voltage, set this slightly higher (~0.05V) than in normal operation
- **charge_current** (*float*) – maximum charge current. A 1/100th is used as (dis-)charge end current.
- **memory_slot** (*int*) – Internal memory slot to save the model to
- **model_voltage_offset** (*float, optional*) – Voltage offset for the generated model upper and lower limits. These have to be in the range of the actual measured values, i.e. in between the hard limits of the measurement. Defaults to 0.05
- **charge_delay** (*float, optional*) – Seconds to wait between discharging and charging. Actual time is higher since the device has some internal delay on reporting end of a test. Defaults to 0.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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_text_data

Set control text to be displayed(24 characters).

property function_mode

Control function mode to use.

Valid values are "POWER", "TEST" and "SIMULATOR".

property id

Get the identification of the instrument.

property line_frequency

Get line frequency.

property measurement_event

Get measurement event register.

property measurement_ongoing: bool

Get measurement status.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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).

property reading_available: bool

Get availability of a reading.

reset()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property status

Get the status byte and Master Summary Status bit.

property summary_event

Get summary event register.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.keithley.keithley2281S.**Keithley2281SOperationEventRegister**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum containing Keithley2281S Operation Instrument Summary Event Register definition

class pymeasure.instruments.keithley.keithley2281S.**Keithley2281SMeasurementEventRegister**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum containing Keithley2281S Measurement Instrument Summary Event Register definition

7.31.5 Keithley 2306 Dual Channel Battery/Charger Simulator

class pymeasure.instruments.keithley.**Keithley2306**(*adapter*, *name='Keithley 2306'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Keithley 2306 Dual Channel Battery/Charger Simulator.

property both_channels_enabled

Set 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
Keithley2306Channel

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Control (floating) 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

Control (integer) the display channel, takes values 1 or 2.

property display_enabled

Control (boolean) whether the display is enabled, takes values True or False.

property display_text_data

Control text to be displayed, takes strings up to 32 characters.

property display_text_enabled

Control (boolean) whether display text is enabled, takes values True or False.

property id

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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).

relay(*relay_number*)

Get a relay channel from this instrument.

Param

relay_number: int: the number of the relay to be selected

Type
Relay

reset()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property status

Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.31.6 Keithley 2400 SourceMeter

class pymeasure.instruments.keithley.**Keithley2400**(*adapter*, *name='Keithley 2400 SourceMeter'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represent the Keithley 2400 SourceMeter and provide 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
```

(continues on next page)

(continued from previous page)

```
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
```

apply_current(*current_range=None*, *compliance_voltage=0.1*)

Configure 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*)

Configure 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

Control whether auto output-off is activated. Valid values are True (output off after measurement) and False (output stays on after measurement).

auto_range_source()

Configure the source to use an automatic range.

property auto_zero

Control whether the auto zero option is enabled. Valid values are True (enabled) and False (disabled) and 'ONCE' (force immediate).

beep(*frequency*, *duration*)

Sound 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 buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control (integer) 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Control (floating) the compliance current in Amps.

property compliance_voltage

Control the compliance voltage in Volts (float, truncated from -210 to 210).

config_buffer(*points=64*, *delay=0*)

Configure 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

Get the current in Amps if configured (float).

property current_nplc

Control (floating) 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

Control the measurement current range in Amps (float, truncated from -1.05 to 1.05). Auto-range is disabled when this property is set.

disable_buffer()

Disable the connection between measurements and the buffer, but does not abort the measurement process.

disable_output_trigger()

Disable the output trigger for the Trigger layer

disable_source()

Disable the source of current or voltage depending on the configuration of the instrument.

property display_enabled

Control whether or not the display of the sourcemeter is enabled. Valid values are True and False.

enable_source()

Enable the source of current or voltage depending on the configuration of the instrument.

property error

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

property filter_count

Control the number of readings that are acquired and stored in the filter buffer (integer, truncated from 1 to 100).

property filter_state

Control if the filter is active (string, strictly 'ON' or 'OFF').

property filter_type

Control (String) the filter's type. REP : Repeating filter MOV : Moving filter

property id

Get the identification of the instrument.

is_buffer_full()

Return True if the buffer is full of measurements.

property line_frequency

Control the line frequency in Hertz (integer, strictly 50 or 60).

property line_frequency_auto

Control the auto line frequency (boolean).

property max_current

Get the maximum current from the buffer.

property max_resistance

Get the maximum resistance from the buffer.

property max_voltage

Get the maximum voltage from the buffer.

property maximums

Get the calculated maximums for voltage, current, and resistance from the buffer data as a list (list of floats).

property mean_current

Get the mean current from the buffer.

property mean_resistance

Get the mean resistance from the buffer.

property mean_voltage

Get the mean voltage from the buffer.

property means

Get the calculated means for voltage, current, and resistance from the buffer data as a list (list of floats).

property measure_concurent_functions

Control the ability to measure more than one function simultaneously (boolean).

measure_current(*nplc=1*, *current=0.000105*, *auto_range=True*)

Configure 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*)

Configure 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*)

Configure 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

Get the minimum current from the buffer.

property min_resistance

Get the minimum resistance from the buffer.

property min_voltage

Get the minimum voltage from the buffer.

property minimums

Get the calculated minimums for voltage, current, and resistance from the buffer data as a list (list of floats).

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property output_off_state

Control 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'*)

Configure 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*)

Ramp 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*)

Ramp 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(***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()

Reset the instrument and clear the queue.

reset_buffer()

Reset the buffer.

property resistance

Get the resistance in Ohms, if configured for this reading.

property resistance_nplc

Control 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

Control the resistance range in Ohms (float, truncated from 0 to 210e6). Auto-range is disabled when this property is set.

sample_continuously()

Cause the instrument to continuously read samples and turns off any buffer or output triggering

set_timed_arm(*interval*)

Set 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*)

Set 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()

Ensure that the current or voltage is turned to zero and disable the output.

property source_current

Control the source current in Amps (float, truncated from -1.05 to 1.05).

property source_current_range

Control the source current range in Amps (float, truncated from -1.05 to 1.05). Auto-range is disabled when this property is set.

property source_delay

Control (floating) 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

Control the auto delay (boolean).

property source_enabled

Control 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

Control (string) 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

Control the source voltage in Volts (float).

property source_voltage_range

Control the source voltage range in Volts (float, truncated from -210 to 210). Auto-range is disabled when this property is set.

property standard_devs

Get the calculated standard deviations for voltage, current, and resistance from the buffer data as a list (list of floats).

start_buffer()

Starts the buffer.

status()

Get the status byte and Master Summary Status bit.

property std_current

Get the current standard deviation from the buffer.

property std_resistance

Get the resistance standard deviation from the buffer.

property std_voltage

Get the voltage standard deviation from the buffer.

stop_buffer()

Abort the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

triad(*base_frequency*, *duration*)

Sound 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()

Execute a bus trigger, which can be used when *trigger_on_bus()* is configured.

property trigger_count

Control the trigger count (integer, truncated from 1 to 2500).

property trigger_delay

Control the trigger delay in seconds (float, truncated from 0 to 999.9999).

trigger_immediately()

Configure measurements to be taken with the internal trigger at the maximum sampling rate.

trigger_on_bus()

Configure the trigger to detect events based on the bus trigger, which can be activated by *trigger()*.

trigger_on_external(*line=1*)

Configure 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()

Enable the front terminals for measurement, and disable the rear terminals.

use_rear_terminals()

Enable the rear terminals for measurement, and disable the front terminals.

property voltage

Get the voltage in Volts if configured (float).

property voltage_nplc

Control 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

Control the measurement voltage range in Volts (float, truncated from -210 to 210). Auto-range is disabled when this property is set.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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

Control the number of wires in use for resistance measurements (integer, strictly 2 or 4).

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.

7.31.7 Keithley 2450 SourceMeter

class pymeasure.instruments.keithley.**Keithley2450**(*adapter*, *name='Keithley 2450 SourceMeter'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represents the Keithley 2450 SourceMeter and provides a high-level interface for interacting with the instrument.

NOTE: The default buffer handling SCPI command set of the Keithley 2450 model is currently unsupported. The instrument works if made to emulate a 2400 model by setting its command set to "SCPI 2400" through the instrument's system settings.

```
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

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
```

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

property buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control (integer) 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if check_get_errors=True is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if check_set_errors=True is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Control (floating) the compliance current in Amps.

property compliance_voltage

Control (floating) the compliance voltage in Volts.

config_buffer(*points=64*, *delay=0*)

Configure 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

Get the current in Amps, if configured for this reading.

property current_filter_count

Control (integer) the number of readings that are acquired and stored in the filter buffer for the averaging

property current_filter_state

Control (string) if the filter is active.

property current_filter_type

Control (String) the filter's type for the current. REP : Repeating filter MOV : Moving filter

property current_nplc

Control (floating) 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

Control SourceMeter output. 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

Control (floating) 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()

Disable 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

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

property id

Get the identification of the instrument.

is_buffer_full()

Return True if the buffer is full of measurements.

property max_current

Get the maximum current from the buffer

property max_resistance

Get the maximum resistance from the buffer

property max_voltage

Get the maximum voltage from the buffer

property maximums

Get the calculated maximums for voltage, current, and resistance from the buffer data as a list.

property mean_current

Get the mean current from the buffer

property mean_resistance

Get the mean resistance from the buffer

property mean_voltage

Get the mean voltage from the buffer

property means

Get 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

Get the minimum current from the buffer

property min_resistance

Get the minimum resistance from the buffer

property min_voltage

Get the minimum voltage from the buffer

property minimums

Get the calculated minimums for voltage, current, and resistance from the buffer data as a list.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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(***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 and clears the queue.

reset_buffer()

Reset the buffer.

property resistance

Get the resistance in Ohms, if configured for this reading.

property resistance_nplc

Control (floating) 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

Control (floating) 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

Control (floating) the source current in Amps.

property source_current_delay

Control (floating) 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

Control (bool) auto delay. Valid values are True and False.

property source_current_range

Control (floating) 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

Get a boolean value that is True if the source is enabled.

property source_mode

Control (string) 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

Control (floating) the source voltage in Volts.

property source_voltage_delay

Control (floating) 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

Control (bool) auto delay. Valid values are True and False.

property source_voltage_range

Control (floating) 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

Get the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

start_buffer()

Starts the buffer.

property status

Get the status byte and Master Summary Status bit.

property std_current

Get the current standard deviation from the buffer

property std_resistance

Get the resistance standard deviation from the buffer

property std_voltage

Get the voltage standard deviation from the buffer

stop_buffer()

Abort 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

Get the voltage in Volts, if configured for this reading.

property voltage_filter_count

Control (integer) the number of readings that are acquired and stored in the filter buffer for the averaging

property voltage_filter_type

Control (String) the filter's type for the current. REP : Repeating filter MOV : Moving filter

property voltage_nplc

Control (floating) 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

Control the SourceMeter output. 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

Control (floating) 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=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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

Control (integer) the number of wires in use for resistance measurements, which can take the value of 2 or 4.

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.

7.31.8 Keithley 2510 TEC SourceMeter

class pymeasure.instruments.keithley.**Keithley2510**(*adapter*, *name='Keithley 2510 TEC SourceMeter'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Keithley 2510 TEC Sourcemeter and provides a high-level interface for interacting with the instrument.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_temperature_stability`(*tolerance=0.1*, *period=10*, *points=64*)

Determine whether the temperature is stable at the temperature setpoint over a specified period.

Parameters

- **tolerance** – Maximum allowed deviation from temperature setpoint, in degrees Centigrade.
- **period** – Time period over which stability is checked, in seconds.

Returns
True if stable, False otherwise.

`clear`()

Clear the instrument status byte.

`property complete`

Get the synchronization bit.

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 DC current through the thermoelectric cooler, in Amps.

`disable_source`()

Disables the source.

`disable_temperature_protection`()

Disable temperature protection.

`enable_source`()

Enables the source.

`enable_temperature_protection`()

Enable temperature protection.

`property id`

Get the identification of the instrument.

`property next_error`

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

`property options`

Get 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()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property source_enabled

Control whether the source is enabled (bool). The convenience methods *enable_source()* and *disable_source()* can also be used.

property status

Get the status byte and Master Summary Status bit.

property temperature

Measure the temperature using the thermistor, in degrees centigrade.

property temperature_pid

Control the temperature PID loop constants through the tuple (proportional_const, integral_const, derivative_const).

property temperature_pid_d

Control the derivative constant of the temperature PID loop.

property temperature_pid_i

Control the integral constant of the temperature PID loop.

property temperature_pid_p

Control the proportional constant of the temperature PID loop.

property temperature_protection_enabled

Control whether temperature protection is enabled. Takes values True or False. The convenience methods *enable_temperature_protection()* and *disable_temperature_protection()* can also be used.

property temperature_protection_high

Control the upper temperature limit in degrees centigrade (float strictly from -50 to 225)

property temperature_protection_low

Control the lower temperature limit in degrees centigrade (float strictly from -50 to 225)

property temperature_protection_range

Control the lower and upper temperature limits in degrees centigrade through the tuple (lower_limit, upper_limit).

property temperature_setpoint

Control the temperature setpoint, in degrees centigrade (float strictly from -50 to 225)

property voltage

Measure the DC voltage through the thermoelectric cooler, in Volts.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_temperature_stable(*tolerance=0.1*, *period=10*, *should_stop=<function Keithley2510.<lambda>>*, *timeout=60*)

Block the program, waiting for the temperature to stabilize at the temperature setpoint.

Parameters

- **tolerance** – Maximum allowed deviation from temperature setpoint, in degrees Centigrade.
- **period** – Time period over which stability is checked, in seconds.
- **should_stop** – Function that returns True to stop waiting.
- **timeout** – Maximum waiting time, in seconds.

Returns
True when stable, False if stopped by should_stop.

Raises
TimeoutError – If the temperature does not stabilize within the timeout period.

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.

7.31.9 Keithley 2600 SourceMeter

class pymeasure.instruments.keithley.**Keithley2600**(*adapter*, *name='Keithley 2600 SourceMeter'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Keithley 2600 series (channel A and B) SourceMeter

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

`check_get_errors()`

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors()`

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`clear()`

Clear the instrument status byte.

`property complete`

Get the synchronization bit.

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`

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

`property id`

Get the identification of the instrument.

`property next_error`

Get a tuple of an error code and message from a single error.

`property options`

Get 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()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property status

Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.31.10 Keithley 2700 MultiMeter/Switch System

class pymeasure.instruments.keithley.**Keithley2700**(*adapter*, *name='Keithley 2700 MultiMeter/Switch System'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represents the Keithley 2700 Multimeter/Switch System and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2700("GPIB::1")
```

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 buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control 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()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

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

Control the opened and closed channels. All mentioned channels are closed, other channels will be opened.

property complete

Get the synchronization bit.

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*)

Configure 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()

Disable 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

Control (string) 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

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

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

Get the identification of the instrument.

is_buffer_full()

Return True if the buffer is full of measurements.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

open_all_channels()

Open all channels of the Keithley 2700.

property open_channels

Set 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

Get the lists of the installed cards in the Keithley 2700. Returns a dict with the integer card numbers on the position.

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 and clears the queue.

reset_buffer()

Reset the buffer.

shutdown()

Brings the instrument to a safe and stable state

start_buffer()

Starts the buffer.

property status

Get the status byte and Master Summary Status bit.

stop_buffer()

Abort the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property text_enabled

Control (boolean) 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=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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(*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.

7.31.11 Keithley 2750 Multimeter/Switch System

class pymeasure.instruments.keithley.**Keithley2750**(*adapter*, *name='Keithley 2750 Multimeter/Switch System'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Keithley2750 multimeter/switch system and provides a high-level interface for interacting with the instrument.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors()`

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`clear()`

Clear the instrument status byte.

`close(channel)`

Closes (connects) the specified channel.

Parameters
channel (*int*) – 3-digit number for the channel

Returns
None

`property closed_channels`

Get the list of closed channels.

`property complete`

Get the synchronization bit.

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`

Get the identification of the instrument.

`property next_error`

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

`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`

Get 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()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property status

Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.31.12 Keithley 6221 AC and DC Current Source

class pymeasure.instruments.keithley.**Keithley6221**(*adapter*, *name='Keithley 6221 SourceMeter'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represents the Keithley 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

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
```

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 buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if check_get_errors=True is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if check_set_errors=True is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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*)

Configure 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.

delta_abort()

Stop delta and place the Model 2182A in the local mode.

delta_arm()

Arm delta.

property delta_buffer_points

Control the size of the buffer (integer strictly from 1 to 1000000).

Buffer size should be the same value as Delta count.

property delta_cold_switch_enabled

Control if cold switching mode is enabled (boolean).

property delta_compliance_abort_enabled

Control if compliance abort is enabled (boolean).

property delta_connected

Get connection status to 2182A.

property delta_cycles

Control the number of cycles to run for the delta measurements (integer strictly from 1 to 65536, or "INF").

property delta_delay

Control the delta delay in seconds (float strictly from 0 to 9999.999, or "INF").

property delta_high_source

Control the delta high source value in A (float strictly from 0 to 0.105).

Set high source value will automatically set the low source value to minus the high source value.

property delta_low_source

Control the delta low source value in A (float strictly from -0.105 to 0).

Usually no need to manually set this. By default, the low source value is minus the high source value.

property delta_measurement_sets

Control the number of measurement sets to repeat for delta measurements (integer strictly from 1 to 65536, or "INF").

property delta_sense

Get the latest delta reading results from 2182/2182A.

delta_start()

Start delta measurements.

property delta_unit

Control the reading unit (string strictly in 'V', 'Ohms', 'W' and 'Siemens').

property delta_values

Get delta sense readings stored in 6221 buffer.

disable_buffer()

Disable 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

Control (boolean) 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

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

property id

Get the identification of the instrument.

is_buffer_full()

Return True if the buffer is full of measurements.

property measurement_event_enabled

Control 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

Get 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 next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property operation_event_enabled

Control 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

Get 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

Get the device options installed.

property output_low_grounded

Control (boolean) 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

Control 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

Get 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(***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 and clears the queue.

reset_buffer()

Reset 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

property shield_to_guard_enabled

Control if shield is connected to the guard(boolean).

If not, the shield is connected to the output-low.

shutdown()

Disables the output.

property source_auto_range

Control (boolean) the auto range of the current source. Valid values are True or False.

property source_compliance

Control (floating) the compliance of the current source in Volts. valid values are in range 0.1 [V] to 105 [V].

property source_current

Control (floating) the source current in Amps.

property source_delay

Control (floating) 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

Control (boolean) 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

Control (floating) 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

Control 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

Control 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

Get 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

Get the status byte and Master Summary Status bit.

stop_buffer()

Abort 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=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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

Control (floating) 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

Control (floating) 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

Control (floating) the duration of the waveform in seconds. Valid values are in range 100e-9 to 999999.999.

property waveform_dutycycle

Control (floating) 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

Control (floating) the frequency of the waveform in Hertz. Valid values are in range 1e-3 to 1e5.

property waveform_function

Control (string) the selected wave function. Valid values are "sine", "ramp", "square", "arbitrary1", "arbitrary2", "arbitrary3" and "arbitrary4".

property waveform_offset

Control (floating) the offset of the waveform in Amps. Valid values are in range -0.105 to 0.105.

property waveform_phasemarker_line

Control (numerical) the line of the phase marker.

property waveform_phasemarker_phase

Control (numerical) the phase of the phase marker.

property waveform_ranging

Control (string) 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

Control (boolean) 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(*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.

7.31.13 Keithley 6517B Electrometer

class pymeasure.instruments.keithley.**Keithley6517B**(*adapter*, *name='Keithley 6517B Electrometer/High Resistance Meter'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represents the Keithley 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
```

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.

property buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control (integer) 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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*)

Configure 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

Get the current in Amps, if configured for this reading.

property current_nplc

Control (floating) 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

Control (floating) 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()

Disable 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

Get the next error from the queue.

Deprecated since version 0.15: Use *next_error* instead.

static extract_value(*result*)

extracts the physical value from a result object returned by the instrument

property id

Get the identification of the instrument.

is_buffer_full()

Return 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 attribute

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 attribute

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 attribute

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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(***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 and clears the queue.

reset_buffer()

Reset the buffer.

property resistance

Get the resistance in Ohms, if configured for this reading.

property resistance_nplc

Control (floating) 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

Control (floating) 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

Control whether the current limit is enabled.

property source_enabled

Get whether the source is enabled.

property source_voltage

Control (floating) the source voltage in Volts.

property source_voltage_range

Control (floating) 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

Get the status byte and Master Summary Status bit.

stop_buffer()

Abort 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

Get the voltage in Volts, if configured for this reading.

property voltage_nplc

Control (floating) 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

Control (floating) 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=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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(*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.

7.31.14 Keithley DMM6500 Multimeter

class pymeasure.instruments.keithley.**KeithleyDMM6500**(*adapter*, *name='Keithley DMM6500 6½-Digit Multimeter'*, *read_termination='\n'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represent the Keithely DMM6500 6½-Digit Multimeter and provide a high-level interface for interacting with the instrument. This class only uses "SCPI" command set (see also *command_set*) to communicate with the instrument.

```
# Access via LAN
ip_address = "xxx.xxx.xxx.xxx"
dmm = KeithleyDMM6500(f"TCPIP::{ip_address}::inst0::INSTR")
```

User can also use PyVISA to get DMM6500's USB port name and pass it to *KeithleyDMM6500*

```
import pyvisa
rm = pyvisa.ResourceManager()
resources = rm.list_resources()

# assume there is only one USB instruments
# Ex. ('USB0::1510::25856::01234567::0::INSTR')

dmm = KeithleyDMM6500( resources[0] )

# Measure voltage
dmm.measure_voltage()
print(dmm.voltage)

# Measure AC voltage
dmm.measure_voltage(ac=True)
print(dmm.voltage)
```

channels

Channels

ch_1: *ScannerCard2000Channel*, ch_2: *ScannerCard2000Channel*, ch_3: *ScannerCard2000Channel*, ch_4: *ScannerCard2000Channel*, ch_5: *ScannerCard2000Channel*, ch_6: *ScannerCard2000Channel*, ch_7: *ScannerCard2000Channel*, ch_8: *ScannerCard2000Channel*, ch_9: *ScannerCard2000Channel*, ch_10: *ScannerCard2000Channel*

`acquire_relative`(*mode=None*)

Set the active value as the relative for the active mode, or can set another mode by its name.

Parameters
mode – A valid *mode* name, or *None* for the active mode

Returns
The relative value that was acquired

`property aperture`

Control the aperture time of currently active *mode*.

Valid values: `MIN`, `DEF`, `MAX`, or number between 8.333u and 0.25 s.

> **Note**
>
> Only `voltage`, `current`, `resistance`, `resistance 4W`, `diode`, `temperature`, `frequency`, `period`, and `voltage ratio` mode support aperture setting. If current active mode doesn't support aperture, this command will hang till adapter's timeout and cause -113 "Undefined header" error.

`auto_range`(*mode=None*)

Set the active mode to use auto-range, or can set another mode by its name.

Only `current` (DC), `current ac`, `voltage` (DC), `voltage ac`, `resistance` (2-wire), `resistance 4W` (4-wire), `capacitance`, and `voltage ratio` support autorange. If chosen mode is not in these modes, this command will do nothing.

Parameters
mode – A valid *mode* name, or *None* for the active mode

`auto_range_status`(*mode=None*)

Get the status of auto-range of active mode or another mode by its name. Only `current` (DC), `current ac`, `voltage` (DC), `voltage ac`, `resistance` (2-wire), `resistance 4W` (4-wire), `capacitance`, and `voltage ratio` support autorange. If chosen mode is not in these modes, this command will also return `False`.

Parameters
mode – A valid *mode* name, or *None* for the active mode

Returns
a bool value for auto-range enabled or disabled

Return type
bool

`property autorange_enabled`

Control the autorange state for currently active *mode*.

> **Note**
>
> If currently active mode doesn't support autorange, this command will hang till adapter's timeout and cause -113 "Undefined header" error.

`property autozero_enabled`

Control automatic updates to the internal reference measurements (autozero) of the instrument.

beep(*frequency*, *duration*)

Sound a system beep.

Parameters

- **frequency** – A frequency in Hz between 20 Hz and 8000 Hz
- **duration** – The amount of time to play the tone between 0.001 s to 100 s

Returns

None

property buffer_points

Control the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead. *0* means the largest buffer possible based on the available memory when the bufer is created.

property buffer_size

Control the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead. *0* means the largest buffer possible based on the available memory when the bufer is created.

property capacitance

Measure a capacitance in Farad, based on the active *mode*.

property capacitance_digits

Control the number of digits in the capacitance readings (integer strictly from 3 to 6). See also the *digits*.

property capacitance_range

Control the capacitance full-scale measure range in Farad. Available ranges are 1e-9, 10e-9, 100e-9, 1e-6, 10e-6, 100e-6, 1e-3. Auto-range is disabled when this property is set. See also the *range*.

property capacitance_relative

Control the capacitance relative value in Farad (float strictly from -0.001 to 0.001 F). See also the *relative*.

property capacitance_relative_status

Control a relative offset value applied to capacitance measurement. See also the *relative_enabled*.

close()

Close the connection

property command_set

Control the command set that to use with DMM6500. Reboot the instrument is needed after changing the command set. Available values are: `SCPI`, `TSP`, `SCPI2000`, and `SCPI34401`. The *KeithleyDMM6500* class was designed to use `SCPI` command set only.

> **Note**
>
> If you want to use TSP command set, you can use `write()` and `ask()` to send TSP command instead.

property current

Measure a DC or AC current in Amps, based on the active *mode*.

property current_ac_bandwidth

Control the detector bandwidth in Hz for AC current measurement (integer strictly among 3, 30, and 300).

property current_ac_digits

Control the number of digits in the AC current readings (integer strictly from 3 to 6). See also the *digits*.

property current_ac_range

Control the AC current positive full-scale measure range in Amps. Available ranges are 1e-3, 10e-3, 100e-3, 1, 3 Amps (for front terminals), and 10 Amps (for rear terminals). See also the *range*.

property current_ac_relative

Control the AC current relative value in Amps (float strictly from -3 to 3). See also the *relative*.

property current_ac_relative_enabled

Control a relative offset value applied to AC current measurement. See also the *relative_enabled*.

property current_digits

Control the number of digits in the DC current readings (integer strictly from 3 to 6). See also the *digits*.

property current_nplc

Control the number of power line cycles (NPLC) for the DC current measurement (float strictly from 0.0005 to 15). See also the *nplc*.

property current_range

Control the DC current full-scale measure range in Amps. Available ranges are 10e-6, 100e-6, 1e-3, 10e-3, 100e-3, 1, 3 Amps (for front terminals), and 10 Amps (for rear terminals). Auto-range is disabled when this property is set. See also the *range*.

property current_relative

Control the DC current relative value in Amps (float strictly from -3 to 3). See also the *relative*.

property current_relative_enabled

Control a relative offset value applied to DC current measurement. See also the *relative_enabled*.

property data_format

Control data format that is used when transferring readings over the remote interface. Available values are ASC (ASCII), REAL (double-precision), or SRE (single-precision).

property detector_bandwidth

Control the lowest frequency expected in the input signal in Hz ONLY for AC voltage and AC current measurement,

Valid values: 3, 30, 300, MIN, DEF, MAX.

property digits

Control the displaying number of digits for currently active *mode*. Available values are from 3 to 6 representing display digits from 3.5 to 6.5.

property diode

Measure a diode's forward voltage drop of general-purpose diodes and the Zener voltage of Zener diodes on the 10V range with a constant test current (bias level), based on the active *mode*.

property diode_bias

Control the amount of current in Amps the instrument sources while making measurement. Available bias levels are 1e-5, 0.0001, 0.001, 0.01.

property diode_nplc

Control the number of power line cycles (NPLC) for the diode measurement (float strictly from 0.0005 to 15). See also the *nplc*.

disable_filter(*mode=None*)

Disable 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

Returns
Filter status read from the instrument

`disable_relative`(*mode=None*)

Disable the application of a relative offset value to the measurement 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 display_screen`

Set displayed front-panel screen by the name. Available names are: `HOME` (home), `HOME_LARG` (home screen with large readings), `READ` (reading table), `HIST` (histogram), `SWIPE_FUNC` (FUNCTIONS swipe screen), `SWIPE_GRAP` (GRAPH swipe screen), `SWIPE_SEC` (SECONDARY swipe screen), `SWIPE_SETT` (SETTINGS swipe screen), `SWIPE_STAT` (STATISTICS swipe screen), `SWIPE_USER` (USER swipe screen), `SWIPE_CHAN` (CHANNEL swipe screen), `SWIPE_NONS` (NONSWITCH swipe screen), `SWIPE_SCAN` (SCAN swipe screen), `CHANNEL_CONT` (Channel control screen), `CHANNEL_SETT` (Channel settings screen), `CHANNEL_SCAN` (Channel scan screen), or `PROC` (minimal CPU resources).

`displayed_text`(*top_line=None*, *bot_line=None*)

Display text messages on the front-panel USER swipe screen. If no messages were defined, screen will be cleared.

Parameters

- **top_line** – 1st line message
- **bot_line** – 2nd line message

Returns
None

`enable_filter`(*mode=None*, *type='repeat'*, *count=1*)

Enable 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, could be `REPeat`, `MOVing`, or `HYBRid`.
- **count** – A number of averages, which can take take values from 1 to 100

Returns
Filter status read from the instrument

`enable_relative`(*mode=None*)

Enable the application of a relative offset value to the measurement 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`

Measure a frequency in Hz, based on the active *mode*.

`property frequency_aperature`

Control the aperture time in seconds for frequency measurement (float strictly from 2 ms to 273 ms). See also *aperture*.

property frequency_digits

Control the number of digits in the frequency readings (integer strictly from 3 to 6). See also the *digits*.

property frequency_relative

Control the frequency relative value in Hz (float strictly from -1 MHz to 1 MHz). See also the *relative*.

property frequency_relative_enabled

Control a relative offset value applied to frequency measurement. See also the *relative_enabled*.

property frequency_threshold

Control the expected input level in Volts for the frequency measurement (float strictly from 0.1 to 750V).

property frequency_threshold_auto_enabled

Control the auto threshold range for frequency measurement enabled or not.

property line_frequency

Get the power line frequency which automatically detected while the instrument is powered on.

measure_capacitance(*max_capacitance=0.001*)

Configure the instrument to measure capacitance.

Parameters
max_capacitance – Set *capacitance_range* after changing *mode*

Returns
None

measure_continuity()

Configure the instrument to perform continuity testing.

Returns
None

measure_current(*max_current=0.01*, *ac=False*)

Configure 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()

Configure the instrument to perform diode testing.

Returns
None

measure_frequency()

Configure the instrument to measure frequency.

measure_period()

Configure the instrument to measure period.

measure_resistance(*max_resistance=10000000.0*, *wires=2*)

Configure the instrument to measure resistance, based on a maximum resistance to set the range.

Parameters

- **max_resistance** (*float*) – A resistance in Ohms to set the resistance range

- **wires** (*int*) – 2 for normal resistance, and 4 for 4-wires resistance

Returns
None

measure_temperature()

Configure the instrument to measure temperature.

measure_voltage(*max_voltage=1*, *ac=False*)

Configure 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

Control the active measure function. Available values are: `current` (DC), `current ac`, `voltage` (DC), `voltage ac`, `resistance` (2-wire), `resistance 4W` (4-wire), `diode`, `capacitance`, `temperature`, `continuity`, `period`, `frequency`, and `voltage ratio`.

property nplc

Control the integration time in number of power line cycles (NPLC) of currently active *mode*. This value sets the amount of time that the input signal is measured. Valid values are: 0.0005 to 15 (60Hz) or 12 (50Hz or 400Hz).

> **Note**
>
> Only `voltage`, `current`, `resistance`, `resistance 4W`, `diode`, `temperature`, and `voltage ratio` mode support NPLC setting. If current active mode doesn't support NPLC, this command will hang till adapter's timeout and cause -113 "Undefined header" error.

property period

Measure a period in seconds, based on the active *mode*.

property period_aperature

Control the aperture time in seconds for period measurement (float strictly from 2 ms to 273 ms). See also *aperture*

property period_digits

Control the number of digits in the period readings (integer strictly from 3 to 6). See also the *digits*.

property period_relative

Control the period relative value in seconds (float strictly from -1 s to 1 s). See also the *relative*.

property period_relative_enabled

Control a relative offset value applied to period measurement. See also the *relative_enabled*.

property period_threshold

Control the expected input level in Volts for the period measurement (float strictly from 0.1 to 750V).

property period_threshold_auto_enabled

Control the auto threshold range for period measurement enabled or not.

property points_in_buffer

Get the number of readings stored in the buffer.

property pseudo_scanner_enabled

Set pseudo scanner card if there's no scanner card in the instrument. After setting, user can check current scanner card by *scan_id*. If a scanner card is installed, this setting won't have any effect.

property range

Control the positive full-scale measure range for currently active *mode*. Auto-range is disabled when this property is set.

For frequency and period measurements, ranging applies to the signal's input voltage, not its frequency

property relative

Control the relative offset value of currently active *mode*. When relative offset is enabled, all subsequent measured readings are offset by the value that is set for this command. If the instrument acquires the value, read this setting to return the value that was measured internally. See also the *relative_enabled*.

property relative_enabled

Control the relative offset value applied to new measurements for currently active *mode*.

property resistance

Measure a resistance in Ohms for both 2-wire and 4-wire configurations, based on the active *mode*.

property resistance_4W_digits

Control the number of digits in the 4-wire resistance readings (integer strictly from 3 to 6). See also the *digits*.

property resistance_4W_nplc

Control the number of power line cycles (NPLC) for the 4-wire resistance measurement (float strictly from 0.0005 to 15). See also the *nplc*.

property resistance_4W_range

Control the 4-wire resistance full-scale measure range in Ohms. Available ranges are: 1, 10, 100, 1e3, 10e3, 100e3, 1e6, 10e6, and 100e6. Auto-range is disabled when this property is set. See also the *range*.

property resistance_4W_relative

Control the 4-wire resistance relative value in Ohms (float strictly from -100M to 100M). See also the *relative*.

property resistance_4W_relative_enabled

Control a relative offset value applied to 4-wire resistance measurement. See also the *relative_enabled*.

property resistance_digits

Control the number of digits in the 2-wire resistance readings (integer strictly from 3 to 6). See also the *digits*.

property resistance_nplc

Control the number of power line cycles (NPLC) for the 2-wire resistance measurement (float strictly from 0.0005 to 15). See also the *nplc*.

property resistance_range

Control the 2-wire resistance full-scale measure range in Ohms. Available ranges are: 10, 100, 1e3, 10e3, 100e3, 1e6, 10e6, and 100e6. Auto-range is disabled when this property is set. See also the *range*.

property resistance_relative

Control the 2-wire resistance relative value in Ohms (float strictly from -100M to 100M). See also the *relative*.

property resistance_relative_enabled

Control a relative offset value applied to 2-wire resistance measurement. See also the *relative_enabled*.

property scan_card_vmax

Get the maximum voltage of all channels.

property scan_channels

Control the channel list of scanning. An empty string will clear the list. Use comma to separate single channel and use a colon to separate the first and last channel in the list. Examples: `1`, `1,3,5`, `1:2, 7:8`, or `1:10`.

property scan_channels_list

Get *scan_channels* string to a list of integers.

For example, when *scan_channels* is `1,3:5,7:8,10`, this attribute will return `[1,3,4,5,7,8,10]`. If `scan_channels_list=[1,2,3,4,6]`, the *scan_channels* will be `1:4,6`.

property scan_count

Control the number of times the scan is repeated. Set to `0` set the scan to repeat until aborted.

property scan_id

Get scanner card's ID.

property scan_interval

Control the interval time (0s to 100ks) between scan starts when the *scan_count* is more than one.

property scan_iscomplete

Get Event Status Register (ESR) bit 0 to determine if previous works were completed. This property is used while running time-consuming scanning operation.

property scan_modes

Get a dictionary of every channel's mode.

scan_start(*block_communication=True*, *count=None*, *interval=None*)

Start the scanner card to close each channel of *scan_channels* sequentially and to do measurements.

If *scan_count* is larger than 1, the next scanning will start again after *scan_interval* second. Running large counts or long interval scanning is a time-consuming operation. It's better to set `block_communication=False` and use *scan_iscomplete* to check if the measurement is completed.

Parameters

- **block_communication** – A bool value which controls communication state while scanning. Default is `True` and the communication waits until the commands are complete to accept new commands
- **count** – An alternative way to set *scan_count* before scanning.
- **interval** – An alternative way to set *scan_interval* in second before scanning.

Returns

None

scan_stop()

Abort the scanning measurement by stopping the measurement arming and triggering sequence.

Returns

None

property scan_vch_end

Get the last channel in the slot that supports voltage or 2-wire measurements.

property scan_vch_start

Get the first channel in the slot that supports voltage or 2-wire measurements.

scanned_data(*start_idx=None*, *end_idx=None*, *raw=False*)

Return a list of scanning values from the buffer.

Parameters

- **start_idx** – A bool value which controls communication state while scanning. Default is `True` and the communication waits until the commands are complete to accept new commands
- **end_idx** – An alternative way to set *scan_count*
- **raw** – An alternative way to set *scan_interval* in second

Returns

A list of scan channels' measured

Return type

A list of channels' list

property system_time

Control system time on the instrument. Format of set is: `year, month, day, hour, minute, second` or `hour, minute, second`. Example: Using `time` package to set instrument's clock: `dmm.system_time = time.strftime("%Y, %m, %d, %H, %M, %S")`

property temperature

Measure a temperature in Celsius, based on the active *mode*.

property temperature_digits

Control the number of digits in the temperature readings (integer strictly from 3 to 6). See also the *digits*.

property temperature_nplc

Control the number of power line cycles (NPLC) for the temperature measurement (float strictly from 0.0005 to 15). See also the *nplc*.

property temperature_relative

Control the temperature relative value in Celsius (float strictly from -3310 C to 3310 C). See also the *relative*.

property temperature_relative_enabled

Control a relative offset value applied to temperature measurement. See also the *relative_enabled*.

property terminals_used

Get which set of input and output terminals the instrument is using. Return can be `FRONT` or `REAR`.

trigger_single_autozero()

Cause the instrument to refresh the reference and zero measurements once.

Consequent autozero measurements are disabled.

property voltage

Measure a DC or AC voltage in Volts, based on the active *mode*.

property voltage_ac_bandwidth

Control the detector bandwidth in Hz for AC voltage measurement (integer strictly among 3, 30, and 300).

property voltage_ac_digits

Control the number of digits in the AC voltage readings (integer strictly from 3 to 6). See also the *digits*.

property voltage_ac_range

Control the AC voltage positive full-scale measure range in Volts. Available ranges are 0.1, 1, 10, 100, 750. Auto-range is disabled when this property is set. See also the *range*.

property voltage_ac_relative

Control the AC voltage relative value in Volts (float strictly from -750 to 750). See also the *relative*.

property voltage_ac_relative_enabled

Control a relative offset value applied to AC voltage measurement. See also the *relative_enabled*.

property voltage_digits

Control the number of digits in the DC voltage readings (integer strictly from 3 to 6). See also the *digits*.

property voltage_nplc

Control the number of power line cycles (NPLC) for the DC voltage measurement (float strictly from 0.0005 to 15). See also the *nplc*.

property voltage_range

Control the DC voltage full-scale measure range in Volts. Available ranges are 0.1, 1, 10, 100, 1000. Auto-range is disabled when this property is set. See also the *range*.

property voltage_relative

Control the DC voltage relative value in Volts (float strictly from -1000 to 1000). See also the *relative*.

property voltage_relative_enabled

Control a relative offset value applied to DC voltage measurement. See also the *relative_enabled*.

class pymeasure.instruments.keithley.keithleyDMM6500.**ScannerCard2000Channel**(*parent*, *id*)

Bases: *Channel*

property autorange_enabled

Control the autorange state for currently active mode.

> **Note**
>
> If current active mode doesn't support autorange, this command will hang till adapter's timeout and cause -113 "Undefined header" error.

disable_filter(*mode=None*)

Disable 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

Returns
Filter status read from the instrument

enable_filter(*mode=None*, *type='repeat'*, *count=1*)

Enable 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, could be `REPeat`, `MOVing`, or `HYBRid`.

- **count** – A number of averages, which can take take values from 1 to 100

Returns
Filter status read from the instrument

property mode

Control the configuration mode for measurements, which can take the values: `current` (DC), `current ac`, `voltage` (DC), `voltage ac`, `resistance` (2-wire), `resistance 4W` (4-wire), `diode`, `capacitance`, `temperature`, `continuity`, `period`, `frequency`, and `voltage ratio`.

property nplc

Control the integration time in number of power line cycles (NPLC). Valid values: 0.0005 to 15 (60Hz) or 12 (50Hz or 400Hz) This command is valid only for `voltage`, 2-wire ohms, and 4-wire ohms.

> **Note**
>
> Only `voltage`, `current`, `resistance`, `resistance 4W`, `diode`, `temperature`, and `voltage ratio` mode support NPLC setting. If current active mode doesn't support NPLC, this command will hang till adapter's timeout and cause -113 "Undefined header" error.

property range

Control measuring range for currently active mode. For `frequency` and `period` measurements, *range* applies to the signal's input voltage, not its frequency

write(*command*)

Write a command to the instrument.

7.31.15 Keithley 2182 Nanovoltmeter

class pymeasure.instruments.keithley.**Keithley2182**(*adapter*, *name='Keithley 2182 Nanovoltmeter'*, *read_termination='\r'*, ***kwargs*)

Bases: *SCPIMixin*, KeithleyBuffer, *Instrument*

Represents the Keithley 2182 Nanovoltmeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2182("GPIB::1")

keithley.reset()                        # Return instrument settings to default
                                          values
keithley.thermocouple = 'S'             # Sets thermocouple type to S
keithley.active_channel = 1             # Sets channel 1 for active measurement
keithley.channel_function = 'voltage'   # Configures active channel for voltage
                                          measurement
print(keithley.voltage)                 # Prints the voltage in volts

keithley.ch_1.setup_voltage()           # Set channel 1 active and prepare
                                          voltage measurement
keithley.ch_2.setup_temperature()       # Set channel 2 active and prepare
                                          temperature measurement
```

ch_1

Channel
Keithley2182Channel

ch_2

Channel
Keithley2182Channel

property active_channel

Control which channel is active for measurement. Valid values are 0 (internal temperature sensor), 1, and 2.

auto_line_frequency()

Set appropriate limits for NPLC voltage and temperature readings.

property auto_zero_enabled

Control the auto zero option (bool).

property buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

property channel_function

Control the measurement mode of the active channel. Valid options are *voltage* and *temperature*.

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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*)

Configure 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.

disable_buffer()

Disable the connection between measurements and the buffer, but does not abort the measurement process.

property display_enabled

Control whether the front display of the voltmeter is enabled. Valid values are True and False.

property id

Get the identification of the instrument.

property internal_temperature

Measure the internal temperature in Celsius.

is_buffer_full()

Return True if the buffer is full of measurements.

property line_frequency

Get the line frequency in Hertz. Values are 50 or 60. Cannot be set on 2182.

property maximum

Get the calculated maximum from the buffer data.

property mean

Get the calculated mean (average) from the buffer data.

property minimum

Get the calculated minimum from the buffer data.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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()
Reset the instrument and clear the queue.

reset_buffer()
Reset the buffer.

sample_continuously()
Configure the instrument to continuously read samples and turn off any buffer or output triggering.

shutdown()
Brings the instrument to a safe and stable state

property standard_dev
Get the calculated standard deviation from the buffer data.

start_buffer()
Starts the buffer.

property status
Get the status byte and Master Summary Status bit.

stop_buffer()
Abort the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property temperature
Measure the temperature in Celsius, if active channel is configured for this reading.

property temperature_nplc
Control the number of power line cycles (NPLC) for temperature measurements, which sets the integration period and measurement speed. Valid values are from 0.01 to 50 or 60, depending on the line frequency. Default is 5.(dynamic)

property temperature_reference_junction
Control whether the thermocouple reference junction is internal (INT) or simulated (SIM). Default is INT.

property temperature_simulated_reference
Control the value of the simulated thermocouple reference junction in Celsius. Default is 23 C.

property thermocouple
Control the thermocouple type for temperature measurements. Valid options are B, E, J, K, N, R, S, and T.

trigger()
Execute a bus trigger, which can be used when *trigger_on_bus()* is configured.

property trigger_count
Control the trigger count which can take values from 1 to 9,999. Default is 1.

property trigger_delay
Control the trigger delay in seconds, which can take values from 0 to 999999.999 s. Default is 0.

trigger_immediately()

Configure measurements to be taken with the internal trigger at the maximum sampling rate.

trigger_on_bus()

Configure the trigger to detect events based on the bus trigger, which can be activated by *trigger()*.

property voltage

Measure the voltage in Volts, if active channel is configured for this reading.

property voltage_nplc

Control the number of power line cycles (NPLC) for voltage measurements, which sets the integration period and measurement speed. Valid values are from 0.01 to 50 or 60, depending on the line frequency. Default is 5.(dynamic)

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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(*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.keithley.keithley2182.**Keithley2182Channel**(*parent*, *id*)

Bases: *Channel*

Implementation of a Keithley 2182 channel.

Channel 1 is the fundamental measurement channel, while channel 2 provides sense measurements. Channel 2 inputs are referenced to Channel 1 LO. Possible configurations are

Voltage (Channel 1) Temperature (Channel 1) Voltage (Channel 1) and Voltage (Channel 2) Voltage (Channel 1) and Temperature (Channel 2)

`acquire_temperature_reference()`

Acquire a temperature measurement and store it as the relative offset value.

Only acquires reference if temperature offset is enabled.

`acquire_voltage_reference()`

Acquire a voltage measurement and store it as the relative offset value.

Only acquires reference if voltage offset is enabled.

`setup_temperature(nplc=5)`

Change active channel and configure channel for temperature measurement.

Parameters
nplc – Number of power line cycles (NPLC) from 0.01 to 50/60

`setup_voltage(auto_range=True, nplc=5)`

Set active channel and configure channel for voltage measurement.

Parameters

- **auto_range** – Enables auto_range if True, else uses set voltage range
- **nplc** – Number of power line cycles (NPLC) from 0.01 to 50/60

`property temperature_offset`

Control the relative offset for measuring temperature. Displayed value = actual value - offset value. Valid values are -273 C to 1800 C.

`property temperature_offset_enabled`

Control whether temperature is measured as a relative or absolute value (bool). Disabled by default.

`property voltage_offset`

Control the relative offset for measuring voltage. Displayed value = actual value - offset value. Valid ranges are -120 V to +120 V for Ch. 1, and -12 V to +12 V for Ch. 2.(dynamic)

`property voltage_offset_enabled`

Control whether voltage is measured as a relative or absolute value (bool). Enabled by default for Ch. 2 voltage, which is measured relative to Ch. 1 voltage.

`property voltage_range`

Control the positive full-scale measurement voltage range in Volts. The Keithley 2182 selects a measurement range based on the expected voltage. DCV1 has five ranges: 10 mV, 100 mV, 1 V, 10 V, and 100 V. DCV2 has three ranges: 100 mV, 1 V, and 10 V. Valid limits are from 0 to 120 V for Ch. 1, and 0 to 12 V for Ch. 2. Auto-range is automatically disabled when this property is set.(dynamic)

`property voltage_range_auto_enabled`

Control the auto voltage ranging option (bool).

7.31.16 Keithley DAQ6510 Data Acquisition Logging Multimeter System

class pymeasure.instruments.keithley.**KeithleyDAQ6510**(*adapter*, *name='Keithley DAQ6510'*, ***kwargs*)

Bases: KeithleyBuffer, *SCPIMixin*, *Instrument*

Represents the Keithley DAQ6510 Data Acquisition Logging Multimeter System and provides a high-level interface for interacting with the instrument.

```
keithley = KeithleyDAQ6510("GPIB::1")
keithley = KeithleyDAQ6510("TCPIP::192.168.1.1::INSTR")

print(keithley.current)                 # Prints the current in Amps
keithley.current_range = 10E-6          # Select the 10 uA range

print(keithley.voltage)                 # Prints the voltage in Volts
keithley.voltage_range = 100e-3         # Select the 100 mV range

print(keithley.resistance)              # Prints the resistance in Ohms
keithley.offset_compensated = "ON"      # Turns offset-compensated ohms on

keithley.open_channels([134, 135])      # Open channels 134 and 135 on the MUX card
keithley.close_channel(133)             # Close channel 133 on the MUX card
```

beep(*frequency*, *duration*)

Sound a system beep.

Parameters

- **frequency** – A frequency in Hz from 20 and 8000 Hz
- **duration** – The amount of time to play the tone, between 0.001 s to 100 s

Returns

None

property buffer_data

Get a numpy array of values from the buffer.

property buffer_points

Control 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns

List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if check_set_errors=True is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

close_channel(*channel*)

Set a single channel to closed.

Parameters
channel – Channel to be set to closed.

close_channels(*channel_list*)

Configure multiple channels to be closed.

Parameters
channel_list – List of channels to be set to closed.

property complete

Get the synchronization bit.

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*)

Configure 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

Measure the current in Amps, if configured for this reading.

property current_nplc

Control the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz). The smallest value is the shortest time, and results in the fastest reading rate, but increases the reading noise and decreases the number of usable digits. The largest value is the longest time, and results in the lowest reading rate, but increases the number of usable digits.

property current_range

Control the measurement current range in Amps. If the measurement function is DC, available ranges are 10E-6 A to 3A. If the measurement function is AC, available ranges are 100E-6 to 3A. Auto-range is disabled when this property is set.

disable_buffer()

Disable the connection between measurements and the buffer, but does not abort the measurement process.

property id

Get the identification of the instrument.

is_buffer_full()

Return True if the buffer is full of measurements.

measure_current(*nplc=1*, *current=3*, *auto_range=True*)

Configure the measurement of current.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz).
- **current** – Upper limit of current in Amps, from 10E-6 to 3 A (DC) or 100E-6 to 3A (AC).
- **auto_range** – A boolean value to enable auto_range if `True`, else uses the set current.

measure_resistance(*nplc=1*, *resistance=100000000.0*, *auto_range=True*)

Configure the measurement of resistance.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz).
- **resistance** – Upper limit of resistance in Ohms, from 10 to 100E6 (2-wire), 1 to 100E6 (4-wire with OCOM off), or 1 to 10E3 (4-wire with OCOM on).
- **auto_range** – A boolean value to enable auto_range if `True`, else uses the set resistance.

measure_voltage(*nplc=1*, *voltage=1000*, *auto_range=True*)

Configure the measurement of voltage.

Parameters

- **nplc** – Number of power line cycles (NPLC) from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz).
- **voltage** – Upper limit of voltage in Volts, from 100E-3 to 1000 V (DC) or 100E-3 to 750 V (AC).
- **auto_range** – A boolean value to enable auto_range if `True`, else uses the set voltage.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property offset_compensated

Control if offset compensation is used or not. Valid values are OFF, ON, and AUTO.

open_channel(*channel*)

Set a single channel to open.

Parameters
channel – Channel to be set to open.

open_channels(*channel_list*)

Configure multiple channels to be open.

Parameters
channel_list – List of channels to be set to open.

property options

Get 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()

Reset the instrument.

reset_buffer()

Reset the buffer.

property resistance

Measure the resistance in Ohms, if configured for this reading.

property resistance_nplc

Control the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz). The smallest value is the shortest time, and results in the fastest reading rate, but increases the reading noise and decreases the number of usable digits. The largest value is the longest time, and results in the lowest reading rate, but increases the number of usable digits.

property resistance_range

Control the resistance range in Ohms. If the measurement function is 2-wire, the available ranges are 10 to 100E6 Ohms. If the measurement function is 4-wire resistance with offset compensation off, the available ranges are 1 to 100E6 Ohms. If the measurement function is 4-wire resistance with offset compensation on, the available ranges are 1 to 10E3 Ohms. Auto-range is disabled when this property is set.

property sense_mode

Control the reading mode, which can take the values 'current' or 'voltage'. The convenience methods `sense_current()` and `sense_voltage()` can also be used.

shutdown()

Brings the instrument to a safe and stable state

start_buffer()

Starts the buffer.

property status

Get the status byte and Master Summary Status bit.

stop_buffer()

Abort the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

property voltage

Measure the voltage in Volts, if configured for this reading.

property voltage_nplc

Control the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 5E-4 to 15 (60 Hz) or 12 (50 Hz or 400 Hz). The smallest value is the shortest time, and results in the fastest reading rate, but increases the reading noise and decreases the number of usable digits. The largest value is the longest time, and results in the lowest reading rate, but increases the number of usable digits.

property voltage_range

Control the measurement voltage range in Volts. If the measurement function is DC, available ranges are 100E-3 V to 1000 V. If the measurement function is AC, available ranges are 100E-3 to 750 V. Auto-range is disabled when this property is set.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*should_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Block 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(*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.

7.31.17 Keithley 4200A-SCS Parameter Analyzer

class pymeasure.instruments.keithley.**Keithley4200**(*adapter*, *name='Keithley 4200A-SCS'*, ***kwargs*)

Bases: *Instrument*

A class representing the Keithley 4200A-SCS Parameter Analyzer.

This driver only uses the user mode commands for controlling the SMUs.

Start the 'KXCI' program on the 4200A-SCS to activate remote control. The remote interface is configured with the Keithley Configuration Utility 'KCon'.

Currently, the driver is only working with the ethernet interface.

add_smu(*id*)

Add a SMU channel to the device.

check_set_errors()

Check for errors after sending a command.

Raise
ValueError if response is not 'ACK'

clear()

Clear all data from the buffer.

It also clears bit B0 (DATA_READY) of the status byte.

property id

Get the identification of the instrument (str).

property options

Get the installed options (list of str).

property status

Get the status byte (IntFlag).

class pymeasure.instruments.keithley.keithley4200.**SMU**(*parent*, *id*)

Bases: *Channel*

A class representing the SMU (source/measure unit) channel.

property current

Measure the current in Amps.

property current_setpoint

Set range, output current and voltage compliance (int, float, float).

Current is in Amps and voltage in Volts.

```
inst = Keithley4200("TCPIP::192.168.1.1::1225::SOCKET")
inst.smu1.current_setpoint = (9, 55e-3, 10)  # (range, value, compliance)
# Set SMU1 to output 55 mA in 100 mA range with 10 V voltage compliance
```

Current source range:

- 0: autorange
- 1: 1 nA range, only with a preamplifier
- 2: 10 nA range, only with a preamplifier

- 3: 100 nA range
- 4: 1 µA range
- 5: 10 µA range
- 6: 100 µA range
- 7: 1 mA range
- 8: 10 mA range
- 9: 100 mA range
- 10: 1 A range, only with a 4210-SMU or 4211-SMU
- 11: 1 pA range, only with a preamplifier
- 12: 10 pA range, only with a preamplifier
- 13: 100 pA range, only with a preamplifier

disable()

Disable the SMU.

property voltage

Measure the voltage in Volts (float).

property voltage_setpoint

Set range, output voltage and current compliance (int, float, float).

Voltage is in Volts and current in Amps.

```
inst = Keithley4200("TCPIP::192.168.1.1::1225::SOCKET")
inst.smu1.voltage_setpoint = (0, 3.3, 1e-2)  # (range, value, compliance)
# Set SMU1 to output 3.3 V in autorange with 10 mA current compliance
```

Voltage source range:

- 0: autorange
- 1: 20 V range
- 2: 200 V range
- 3: 200 V range
- 4: 200 mV range, only with a preamplifier
- 5: 2 V range, only with a preamplifier

class pymeasure.instruments.keithley.keithley4200.**StatusCode**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

A class representing the status codes ofd the Keithley 4200.

Status codes:

- 0: NONE
- 1: DATA_READY

- 2: SYNTAX_ERROR
- 16: BUSY
- 64: SERVICE_REQUEST

7.32 KEPCO INC.

This section contains specific documentation on the Kepco Inc. power supplies instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.32.1 BOP Bipolar Power Supply with BIT 4886 Digital Interface Card

class pymeasure.instruments.kepco.**KepcoBOP3612**(*adapter*, *name='Kepco BOP 36-12 Bipolar Power Supply'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Kepco BOP 36-12 (M or D) 400 W bipolar power supply fitted with BIT 4886 digital interface card (minimal implementation) and provides a high-level interface for interacting with the instrument.

beep()

Cause the unit to emit a brief audible tone.

property bop_test

Get error code after performing full power supply self-test.

Returns 0 if all tests passed, otherwise corresponding error code as detailed in manual. Caution: Output will switch on and swing to maximum values. Disconnect any load before testing.

property confidence_test

Get error code after performing interface self-test procedure.

Returns 0 if all tests passed, otherwise corresponding error code as detailed in manual.

property current

Measure current through the output terminals in Amps.

property current_setpoint

Control the output current setpoint.

Functionality depends on the operating mode. If power supply in current mode, this sets the output current setpoint. The current achieved depends on the voltage compliance and load conditions (see: *current*). If power supply in voltage mode, this sets the compliance current for the corresponding voltage set point. Query returns programmed value, meaning of which is dependent on power supply operating context (see: *operating_mode*).

Output must be enabled separately (see: *output_enabled*)

property operating_mode

Control the operating mode of the BOP.

As a command, a string, VOLT or CURR, is sent. As a query, a 0 or 1 is returned, corresponding to VOLT or CURR respectively. This is mapped to corresponding string.

property output_enabled

Control whether the source is enabled, takes values True or False (bool)

property voltage

Measure voltage present across the output terminals in Volts.

property voltage_setpoint

Control the output voltage setpoint.

Functionality depends on the operating mode. If power supply in voltage mode, this sets the output voltage setpoint. The voltage achieved depends on the current compliance and load conditions (see: *voltage*). If power supply in current mode, this sets the compliance voltage for the corresponding current set point. Query returns programmed value, meaning of which is dependent on power supply operating context (see: *operating_mode*).

Output must be enabled separately (see: *output_enabled*)

wait_to_continue()

Cause the power supply to wait until all previously issued commands and queries are complete before executing subsequent commands or queries.

7.33 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*.

If the instrument you are looking for is not here, also check *Agilent* or *HP* for older instruments.

7.33.1 Keysight DSOX1102G Oscilloscope

class pymeasure.instruments.keysight.**KeysightDSOX1102G**(*adapter*, *name='Keysight DSOX1102G Oscilloscope'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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) occurring when sending commands immediately after digitize. Current fix: if deemed necessary, add delay between digitize and follow-up command to scope.

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".

autoscale()

Autoscale displayed channels.

clear_status()

Clear device status.

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.

run()

Starts repetitive acquisitions.

This is the same as pressing the Run key on the front panel.

single()

Causes the instrument to acquire a single trigger of data. This is the same as pressing the Single key on the front panel.

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.

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”.

7.33.2 Keysight N5767A Power Supply

class pymeasure.instruments.keysight.**KeysightN5767A**(*adapter*, *name='Keysight N5767A power supply'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Keysight N5767A Power supply interface for interacting with the instrument.

property current

Get current in Amps.

property current_range

Control the DC current range in Amps, which can take values from 0 to 25 A. Auto-range is disabled when this property is set. (float)

disable()

Disables the flow of current.

enable()

Enables the flow of current.

is_enabled()

Returns True if the current supply is enabled.

property voltage

Get a DC voltage measurement in Volts.

property voltage_range

Control the DC voltage range in Volts, which can take values from 0 to 60 V. Auto-range is disabled when this property is set.

7.33.3 Keysight N5776C Power Supply

class pymeasure.instruments.keysight.**KeysightN7776C**(*adapter*, *name='N7776C Tunable Laser Source'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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
```

close()

Fully closes the connection to the instrument through the adapter connection.

get_wl_data()

Function returning the wavelength data logged in the internal memory of the laser

property locked

Control the lock state (True/False) of the laser source. (bool)

next_step()

Performs the next sweep step in stepped sweep if it is paused or in manual mode.

property output_enabled

Control the state (on/off) of the laser source (bool)

property output_power_dBm

Control the output power in dBm.

property output_power_mW

Control the output power in mW

previous_step()

Performs one sweep step backwards in stepped sweep if its paused or in manual mode.

property sweep_mode

Control sweep mode of the swept laser source

property sweep_points

Get the number of datapoints that the :READout:DATA? command will return.

property sweep_speed

Control speed of the sweep (in nanometers per second).

property sweep_state

Control 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

Control step width of the sweep (in nanometers).

property sweep_twoway

Control the repeat mode. Applies in stepped,continuous and manual sweep mode.

property sweep_wl_start

Control Start Wavelength (in nanometers) for a sweep.

property sweep_wl_stop

Control End Wavelength (in nanometers) for a sweep.

property trigger_in

Control the incoming trigger response and arm the module.

property trigger_out

Control if and at which point in a sweep cycle an output trigger is generated and arms the module.

property wavelength

Control absolute wavelength of the output light (in nanometers)

property wl_logging

Control State (on/off) of the lambda logging feature of the laser source.

7.33.4 Keysight E36312A Triple Output Power Supply

class pymeasure.instruments.keysight.**KeysightE36312A**(*adapter*, *name='Keysight E36312A'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Keysight E36312A Power supply interface for interacting with the instrument.

```
supply = KeysightE36312A(resource)
supply.ch_1.voltage_setpoint=10
supply.ch_1.current_setpoint=0.1
supply.ch_1.output_enabled=True
print(supply.ch_1.voltage)
```

ch_1

Channel
VoltageChannel

ch_2

Channel
VoltageChannel

ch_3

Channel
VoltageChannel

class BaseChannelCreator(*cls*, ***kwargs*)

Bases: `object`

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Bases: *BaseChannelCreator*

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
```

(continues on next page)

(continued from previous page)

```
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.
- ****kwargs** – Keyword arguments for all children.

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

Bases: *BaseChannelCreator*

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns

Instance of the created child.

ask(*command*, *query_delay=None*)

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=None*, ***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.

`check_get_errors`()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors`()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`clear`()

Clear the instrument status byte.

`property complete`

Get the synchronization bit.

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>>*, *get_process_list=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=None*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=', '*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.

- **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

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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 get_channel_pairs(*cls*)

Return a list of all the Instrument's channel pairs

static get_channels(*cls*)

Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

property id

Get the identification of the instrument.

static measurement(*get_command*, *docs*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *command_process=None*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=','*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **command_process** – A function that take a command and allows processing before executing the command, for getting

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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()

Reset the instrument.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*)

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

Get the status byte and Master Summary Status bit.

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*, ***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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the *ask()* method.

Returns

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

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.keysight.keysightE36312A.**VoltageChannel**(*parent*, *id*)

Bases: *Channel*

property current

Measure the actual current of this channel.

property current_limit

Control the current limit of this channel, range depends on channel.(dynamic)

property output_enabled

Control whether the channel output is enabled (boolean).

property voltage

Measure actual voltage of this channel.

property voltage_setpoint

Control the output voltage of this channel, range depends on channel.(dynamic)

7.33.5 Keysight E3631A Triple Output Power Supply

class pymeasure.instruments.keysight.**KeysightE3631A**(*adapter*, *name='Keysight E3631A'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Keysight E3631A Triple Output DC Power Supply interface for interacting with the instrument.

```
supply = KeysightE3631A(resource)
supply.ch_1.voltage_setpoint=10
supply.ch_1.current_setpoint=0.1
supply.ch_1.output_enabled=True
print(supply.ch_1.voltage)
```

ch_1

Channel
VoltageChannel

ch_2

Channel
VoltageChannel

ch_3

Channel
VoltageChannel

class BaseChannelCreator(*cls*, ***kwargs*)

Bases: object

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Bases: *BaseChannelCreator*

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with CommonBase.add_child(). The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.
- ****kwargs** – Keyword arguments for all children.

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

Bases: *BaseChannelCreator*

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns

Instance of the created child.

ask(*command*, *query_delay=None*)

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=None*, ***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.

`check_get_errors`()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors`()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`clear`()

Clear the instrument status byte.

`property complete`

Get the synchronization bit.

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>>*, *get_process_list=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=None*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=', '*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.

- **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

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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 get_channel_pairs(*cls*)

Return a list of all the Instrument's channel pairs

static get_channels(*cls*)

Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

property id

Get the identification of the instrument.

static measurement(*get_command*, *docs*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *command_process=None*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=','*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **command_process** – A function that take a command and allows processing before executing the command, for getting

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property output_enabled

Control whether the output of the last used channel is enabled (boolean).(dynamic)

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()

Reset the instrument.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*)

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

Get the status byte and Master Summary Status bit.

property tracking_enabled

Control whether the power supply operates in the track mode (boolean)

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*, ***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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the *ask()* method.

Returns

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

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.keysight.keysightE3631A.**VoltageChannel**(*parent*, *id*)

Bases: *Channel*

Implementation of a power supply base class channel

property current

Measure the actual current of this channel.

property current_limit

Control the current limit of this channel, range depends on channel.(dynamic)

property output_enabled

Control whether the channel output is enabled (boolean).

property voltage

Measure actual voltage of this channel.

property voltage_setpoint

Control the output voltage of this channel, range depends on channel.(dynamic)

7.33.6 Keysight 81160A Pulse Function Arbitrary Noise Generator

class pymeasure.instruments.keysight.**Keysight81160A**(*adapter*, *name='Agilent 33500 Function/Arbitrary Waveform generator family'*, ***kwargs*)

Bases: *Agilent33500*

Represent the Keysight 81160A and provide a high-level interface for interacting with the instrument.

```
generator = Keysight81160A("GPIB::1")          # Replace with your device address
generator.reset()                              # Reset the generator to default␣
↪settings

generator.shape = "SIN"                        # Set default channel output shape to␣
↪sine
generator.channels[1].shape = "SIN"            # Set channel 1 output signal shape to␣
↪sine
generator.frequency = 1e3                      # Set default channel output frequency␣
↪to 1 kHz
generator.channels[1].frequency = 1e3          # Set channel 1 output frequency to 1␣
↪kHz
generator.channels[2].amplitude = 5            # Set channel 2 output amplitude to 5␣
↪Vpp
generator.channels[2].offset = 0.5             # Set channel 2 output offset to 0.5 V
generator.channels[2].output = True            # Enable channel 2 output

# Output a square wave at 1 kHz with 5 Vpp and 0.5 V offset
generator.channels[2].apply_square(1e3, 5, 0)

ch1 = generator.channels[1]                    # Short form for channel 1
waveform = [-2.0, -1.5, 0.0, 1.5, 2.0]         # Define a user-defined waveform in␣
↪Volts
ch1.waveform_volatile = waveform               # Set user-defined waveform to␣
↪volatile memory
print(f"{ch1.free_memory_slots} slots free") # Get number of slots in non-volatile␣
↪memory
ch1.save_waveform(waveform, "test")            # Save the waveform to non-volatile␣
↪memory
```

(continues on next page)

(continued from previous page)

```
ch1.shape = "USER"                              # Set channel 1 shape to user-defined
ch1.trigger_mode = "MAN"                        # Set trigger mode to manual
ch1.trigger_count = 2                           # Set number of cycles to be generated␣
↪to 2
ch1.output = True                               # Enable channel 1 output
generator.trigger()                             # Trigger the generator to output the␣
↪waveform
ch1.delete_waveform("test")                     # Delete the waveform from non-
↪volatile memory
```

ch_1

Channel
Keysight81160AChannel

ch_2

Channel
Keysight81160AChannel

class BaseChannelCreator(*cls*, ***kwargs*)

Bases: `object`

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Bases: *BaseChannelCreator*

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.
- ****kwargs** – Keyword arguments for all children.

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

Bases: *BaseChannelCreator*

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.
- ****kwargs** – Keyword arguments for all children.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.

- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns
Instance of the created child.

property amplitude

Control the voltage amplitude in Volts (float).(dynamic)

property amplitude_unit

Control the amplitude units (string, strictly 'VPP', 'VRMS', or 'DBM').(dynamic)

property arb_advance

Control how the device advances from data point to data point (str). Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

property arb_file

Control the arbitrary signal to use from the volatile memory of the device.(dynamic)

property arb_filter

Control the filter setting for arbitrary signals (str). Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

property arb_srate

Control the sample rate of the currently selected arbitrary signal in Sa/s (float). Valid values range from 1 µSa/s to 250 MSa/s (maximum range, device-dependent).

ask(*command*, *query_delay=None*)

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.

beep()

Causes a system beep.

binary_values(*command*, *query_delay=None*, ***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 burst_mode

Control the burst mode type (str, strictly 'TRIG<GERED>' or 'GAT<ED>').(dynamic)

property burst_ncycles

Control the number of cycles to be output when a burst is triggered (int).(dynamic)

property burst_period

Control the period of subsequent bursts in seconds (float).(dynamic)

property burst_state

Control the burst mode state (bool).(dynamic)

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

clear_display()

Removes a text message from the display.

property complete

Get the synchronization bit.

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>>*, *get_process_list=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=None*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=', '*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **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

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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
```

(continues on next page)

(continued from previous page)

```
    # 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.

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

Set text to be displayed on the front panel of the device (string).

property ext_trig_out

Control whether the trigger out signal is active (bool).

property frequency

Control the waveform frequency in Hz (float). Depends on the specified shape.(dynamic)

static get_channel_pairs(*cls*)

Return a list of all the Instrument's channel pairs

static get_channels(*cls*)

Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

property id

Get the identification of the instrument.

static measurement(*get_command*, *docs*, *values=()*, *map_values=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *command_process=None*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=','*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **command_process** – A function that take a command and allows processing before executing the command, for getting

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property offset

Control the voltage offset in Volts (float).(dynamic)

property options

Get the device options installed.

property output

Control the output state (bool).(dynamic)

property output_load

Control the expected load resistance in Ohms (str or float). The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm.(dynamic)

property phase

Control the waveform phase in degrees (float, from -360 to 360).

phase_sync()

Synchronize the phase of all channels.

property pulse_dutycycle

Control the pulse duty cycle in percent (float, from 0 to 100).(dynamic)

property pulse_hold

Control whether pulse width or duty cycle is maintained when the period or frequency of the waveform is changed (str).(dynamic)

property pulse_period

Control the pulse period in seconds (float). Overwrites frequency. If the period is shorter than the pulse width + the edge time, the edge time and pulse width are adjusted.(dynamic)

property pulse_transition

Control the pulse edge time (both rising and falling) in seconds (float).(dynamic)

property pulse_width

Control the pulse width in seconds (float).(dynamic)

property ramp_symmetry

Control the ramp waveform symmetry in percent (float, from 0 to 100).(dynamic)

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()

Reset the instrument.

static setting(*set_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map_values=False*, *set_process=<function CommonBase.<lambda>>*, *check_set_errors=False*, *dynamic=False*)

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.

property shape

Control the output waveform shape (str).(dynamic)

shutdown()

Brings the instrument to a safe and stable state

property square_dutycycle

Control the square wave duty cycle in percent (float).(dynamic)

property status

Get the status byte and Master Summary Status bit.

trigger()

Send a trigger signal to the function generator.

property trigger_source

Control the trigger source (str).

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*, ***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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.

- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the *ask()* method.

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

property voltage_high
Control the upper voltage level in Volts (float).(dynamic)

property voltage_low
Control the lower voltage level in Volts (float).(dynamic)

wait_for(*query_delay=None*)
Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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.

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.keysight.keysight81160A.**Keysight81160AChannel**(**args*, ***kwargs*)
Bases: *Agilent33500Channel*

Represent the Keysight 81160A channel and provide a high-level interface for interacting with the instrument channels.

apply_dc(*voltage*)
Apply a DC voltage.

Parameters
voltage – Voltage to be applied (float).

apply_noise(*amplitude*, *offset*)

Apply noise to the output.

Parameters

- **amplitude** – The amplitude of the noise (float).
- **offset** – The offset voltage (float).

apply_pulse(*frequency*, *amplitude*, *offset*)

Apply a pulse waveform.

Parameters

- **frequency** – Frequency of the pulse (float).
- **amplitude** – Amplitude of the pulse (float).
- **offset** – Offset voltage (float).

apply_sin(*frequency*, *amplitude*, *offset*)

Apply a sine waveform.

Parameters

- **frequency** – Frequency of the sine wave (float).
- **amplitude** – Amplitude of the sine wave (float).
- **offset** – Offset voltage (float).

apply_square(*frequency*, *amplitude*, *offset*)

Apply a square waveform.

Parameters

- **frequency** – Frequency of the square wave (float).
- **amplitude** – Amplitude of the square wave (float).
- **offset** – Offset voltage (float).

apply_user_waveform(*frequency*, *amplitude*, *offset*)

Apply a user-defined waveform.

Parameters

- **frequency** – Frequency of the user waveform (float).
- **amplitude** – Amplitude of the user waveform (float).
- **offset** – Offset voltage (float).

property coupling_enabled

Control whether the channel coupling is enabled (bool). 'True' to copy values from this channel to another one.

delete_waveform(*name*)

Delete a waveform from the generator's nonvolatile memory.

Parameters

name – The name of the user-defined waveform to be deleted.

property free_memory_slots

Get the number of free non-volatile memory slots to store user waveforms (int).

property limit_high

Control the high-level voltage limit in Volts (float).(dynamic)

property limit_low

Control the low-level voltage limit in Volts (float).(dynamic)

property limit_state_enabled

Control whether the state limit is enabled (bool).

save_waveform(*waveform*, *name*)

Save a waveform to the generator's nonvolatile memory.

Parameters

- **waveform** – The waveform data.
- **name** – The name that will be used to identify the waveform in memory, up to 12 characters. The first character must be a letter (A-Z), but the remaining characters can be numbers (0-9) or the underscore character ('_'). Blank spaces are not allowed.

property trigger_count

Control the number of cycles to be output when a burst is triggered (int). Enable burst state if number > 1.

Short form of burst_ncycles and burst_state = True(dynamic)

property trigger_mode

Control the triggering mode (string, strictly 'IMM', 'INT2', 'EXT' or 'MAN').(dynamic)

property waveform_volatile

Get volatile waveform data.

Warning: This property may not reflect the current device volatile waveform. It only returns the waveform previously set via this driver. If the device has been power-cycled, modified externally, or if no waveform was set, the returned value may be invalid or None.

Returns

waveform data (array-like) or None if not set.

property waveforms

Get the available user waveforms in memory (list[str]).

7.34 Kuhne Electronic

This section contains specific documentation on the Kuhne Electronic instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.34.1 KU SG 2.45 250 A - 2.45 GHz ISM-Band Microwave Generator

class pymeasure.instruments.kuhneelectronic.**Kusg245_250A**(*adapter*, *name='KU SG 2.45 250 A'*, *power_limit=250*, ***kwargs*)

Bases: *Instrument*

Represents KU SG 2.45 250 A the 2.45 GHz ISM-Band Microwave Generator and provides a high-level interface for interacting with the instrument.

Parameters
power_limit – power set-point limit in Watts (integer from 0 to 250). See *power_setpoint* and *tune()*.

Usage example:

```
from pymeasure.instruments.kuhneelectronic import Kusg245_250A

generator = Kusg245_250A("ASRL3::INSTR", power_limit=100) # limits the output
                                                          # power set-point to 100 W

generator.external_enabled = False      # biasing and RF output controlled by␣
↪serial comm
generator.power = 20                    # Sets the output power to 20 Watts
generator.bias_enabled = True           # Enables amplifier biasing
generator.rf_enabled = True             # Enables the RF output

p_fwd = generator.power_forward         # Reads forward power in Watts
p_rev = generator.power_reverse         # Reads reflected power in Watts
```

property bias_enabled

Control whether transistor biasing is enabled (boolean).

Biasing must be enabled before switching RF on (see *rf_enabled*).

clear_VSWR_error()

Clear the VSWR error.

See: *reflection_limit*.

property external_enabled

Control whether amplifier enabling is done via external inputs on 8-pin connector or via serial interface (boolean).

property freq_steps_fine_enabled

Control whether fine frequency steps are enabled (boolean).

property frequency_coarse

Control coarse frequency in MHz (integer from 2400 to 2500).

Fine frequency mode must be disabled (see *freq_steps_fine_enabled*). Resolution: 1 MHz. Invalid values are truncated.

property frequency_fine

Control fine frequency in kHz (integer from 2400000 to 2500000).

Fine frequency mode must be enabled (see *freq_steps_fine_enabled*). Resolution: 10 kHz. Invalid values are truncated. Values are rounded to tens.

property off_time

Control off time for the pulse mode in ms (integer from 10 to 1000).

Resolution: 5 ms. Invalid values are truncated. Values are rounded to multipliers of 5.

property phase_shift

Control phase shift in degrees (float from 0 to 358.6).

Resolution: 8-bits. Values out of range are truncated.

property power_forward

Measure forward power in Watts.

property power_reverse

Measure reverse power in Watts.

property power_setpoint

Control output power set-point in Watts (integer from 0 to `power_limit` parameter - see constructor).

Resolution: 1 W. Invalid values are truncated. (dynamic)

property pulse_mode_enabled

Control whether pulse mode is enabled (boolean).

> **Note**
>
> Biasing must be enabled before the pulse mode is enabled (see *bias_enabled*)

property pulse_width

Control pulse width in ms (integer from 10 to 1000).

Resolution: 5 ms. Invalid values are truncated. Values are rounded to multipliers of 5.

property reflection_limit

Control limit of reflection in Watts (integer in 0 - no limit, 100, 150, 180, 200, 230).

> **Note**
>
> If the limit for the reflected power is reached, the forward power is reduced to the specified value and the power control mechanism is locked until the alarm has been cleared by the user via *clear_VSWR_error()*.

property rf_enabled

Control whether RF output is enabled (boolean).

> **Note**
>
> Biasing must be enabled before RF is enabled (see *bias_enabled*)

store_settings()

Save actual settings to EEPROM.

The following parameters are stored: frequency mode (see *freq_steps_fine_enabled*), frequency (see *frequency_coarse* or *frequency_fine*), output power set-point (see *power_setpoint*), ON/OFF control setting (see *external_enabled*), reflection limit (see *reflection_limit*), on time for pulse mode (see *pulse_width*) and off time for pulse mode (see *off_time*).

property temperature

Measure temperature near final transistor in °C.

tune(*power*)

Find and set frequency with lowest reflection at a given power.

Parameters
power – A power set-point for tuning (in Watts). (integer from 0 to `power_limit` parameter - see constructor).

`turn_off()`
Safe turn-off the generator.

1. Disable RF output.
2. Deactivate biasing.

`turn_on()`
Safe turn-on the generator.

1. Activate biasing.
2. Enable RF output.

`property version`
Get firmware version.

`property voltage_32v`
Measure 32V supply voltage in Volts.

`property voltage_5v`
Measure internal 5V supply voltage in Volts.

`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.

7.35 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*. lakeshore3xx is a general class for Lake Shore temperature controllers of the series 3xx (like 325, 331, 334, 336, 340, 350).

7.35.1 Lake Shore 211 Temperature Monitor

`class pymeasure.instruments.lakeshore.LakeShore211(adapter, name='Lake Shore 211 Temperature Monitor', **kwargs)`

Bases: `SCPIUnknownMixin`, *Instrument*

Represents the Lake Shore 211 Temperature Monitor and provides a high-level interface for interacting with the instrument.

Untested properties and methods will be noted in their docstrings.

```
controller = LakeShore211("GPIB::1")

print(controller.temperature_celsius)     # Print the sensor temperature in celsius
```

class AnalogMode(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class AnalogRange(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class RelayMode(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

class RelayNumber(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

property analog_configuration

Control the analog mode and analog range. Values need to be supplied as a tuple of (analog mode, analog range) Analog mode can be 0 or 1

setting	mode
0	voltage
1	current

Analog range can be 0 through 5

setting	range
0	0 – 20 K
1	0 – 100 K
2	0 – 200 K
3	0 – 325 K
4	0 – 475 K
5	0 – 1000 K

property analog_out

Measure the percentage of output of the analog output.

configure_alarm(*on=True*, *high_value=270.0*, *low_value=0.0*, *deadband=0*, *latch=False*)

Configures the alarm parameters for the input.

Parameters

- **on** – Boolean setting of alarm, default True
- **high_value** – High value the temperature is checked against to activate the alarm
- **low_value** – Low value the temperature is checked against to activate the alarm
- **deadband** – Value that the temperature must change outside of an alarm condition
- **latch** – Specifies if the alarm should latch or not

configure_relay(*relay*, *mode*)

Configure the relay mode of a relay

Property is UNTESTED

Parameters

- **relay** (RelayNumber) – Specify which relay to configure
- **mode** (RelayMode) – Specify which mode to assign

property display_units

Control the input data to display. Valid entries:

setting	units
'kelvin'	Kelvin
'celsius'	Celsius
'sensor'	Sensor Units
'fahrenheit'	Fahrenheit

get_alarm_status()

Query the current alarm status

Returns

Dictionary of current status [on, high_value, low_value, deadband, latch]

get_relay_mode(*relay*)

Get the status of a relay

Property is UNTESTED

Parameters

relay (RelayNumber) – Specify which relay to query

Returns

Current RelayMode of queried relay

reset_alarm()

Resets the alarm of the Lakeshore 211

property temperature_celsius

Measure the temperature of the sensor in celsius

property temperature_fahrenheit

Measure the temperature of the sensor in fahrenheit

property temperature_kelvin

Measure the temperature of the sensor in kelvin

property temperature_sensor

Measure the temperature of the sensor in sensor units

7.35.2 Lake Shore 224 Temperature Monitor

class pymeasure.instruments.lakeshore.**LakeShore224**(*adapter*, *name='Lakeshore Model 224 Temperature Controller'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Lakeshore 224 Temperature monitor and provides a high-level interface for interacting with the instrument. Note that the 224 provides 12 temperature input channels (A, B, C1-5, D1-5). This driver makes use of the *LakeShore Channel Classes*

```
monitor = LakeShore224('GPIB::1')

print(monitor.input_A.kelvin)           # Print the temperature in kelvin on sensor␣
↪A
monitor.input_A.wait_for_temperature()  # Wait for the temperature on sensor A to␣
↪stabilize.
```

input_0

Channel
LakeShoreTemperatureChannel

input_A

Channel
LakeShoreTemperatureChannel

input_B

Channel
LakeShoreTemperatureChannel

input_C1

Channel
LakeShoreTemperatureChannel

input_C2

Channel
LakeShoreTemperatureChannel

input_C3

Channel
LakeShoreTemperatureChannel

input_C4

Channel
LakeShoreTemperatureChannel

input_C5

Channel
LakeShoreTemperatureChannel

input_D1

Channel
LakeShoreTemperatureChannel

input_D2

Channel
LakeShoreTemperatureChannel

input_D3

Channel
LakeShoreTemperatureChannel

input_D4

Channel
LakeShoreTemperatureChannel

input_D5

Channel
LakeShoreTemperatureChannel

7.35.3 Lake Shore 331 Temperature Controller

class pymeasure.instruments.lakeshore.**LakeShore331**(*adapter*, *name='Lakeshore Model 331 Temperature Controller'*, *read_termination='\r\n'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Lake Shore 331 Temperature Controller and provides a high-level interface for interacting with the instrument. Note that the 331 provides two input channels (A and B) and two output channels (1 and 2). This driver makes use of the *LakeShore Channel Classes*.

```
controller = LakeShore331("GPIB::1")

print(controller.output_1.setpoint)          # Print the current setpoint for loop 1
controller.output_1.setpoint = 50            # Change the loop 1 setpoint to 50 K
controller.output_1.heater_range = 'low'     # Change the heater range to low.
controller.input_A.wait_for_temperature()    # Wait for the temperature to stabilize.
print(controller.input_A.kelvin              # Print the temperature at sensor A.
```

Deprecated since version 0.16.0: Use LakeShore3xx instead.

input_A

Channel
LakeShoreTemperatureChannel

input_B

Channel
LakeShoreTemperatureChannel

output_1

Channel
LakeShoreHeaterChannel

output_2

Channel
LakeShoreHeaterChannel

7.35.4 Lake Shore 3xx Temperature Controller

class pymeasure.instruments.lakeshore.**LakeShore3xx**(*adapter*, *name='Lakeshore Model 3xx Temperature Controller'*, *read_termination='\r\n'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents a Lake Shore Temperature Controller of the 3xx-Series and provides a high-level interface for interacting with the instrument. Multiple input-and output options are implemented. Not all channels exist in all device versions! Some devices can host input option cards to add more input channels.

The LS 331 and 335 has two input channels (A and B) and two output channels (1 and 2). The LS 336 has four input channels (A, B, C and D) and four output channels (1, 2, 3 and 4).

The LS 340 has two input channels (A and B) and one output channel (1). Following extension cards can be used: * 3465 Single Capacitance Input Option Card adds one channel (C). * 3462 Dual Standard Input Option Card adds two channels (C and D). * 3464 Dual Thermocouple Input Option Card adds two channels (C and D). * 3468 Eight Channel Input Option Card adds eight channels (C1-C4, D1-D4).

The LS 350 has four input channels (A, B, C and D) and four output channels (1, 2, 3 and 4). Following extension card can be used: * 3062 4-Channel Scanner option adds 4 additional channels (D2, D3, D4, and D5).

This driver makes use of the *LakeShore Channel Classes*.

```
controller = LakeShore3xx("GPIB::1")

print(controller.output_1.setpoint)          # Print the current setpoint for loop 1
controller.output_1.setpoint = 50            # Change the loop 1 setpoint to 50 K
controller.output_1.heater_range = 'low'     # Change the heater range to low.
controller.input_A.wait_for_temperature()    # Wait for the temperature to stabilize.
print(controller.input_A.kelvin)             # Print the temperature at sensor A.
```

input_A

Channel
LakeShoreTemperatureChannel

input_B

Channel
LakeShoreTemperatureChannel

input_C

Channel
LakeShoreTemperatureChannel

input_C1

Channel
LakeShoreTemperatureChannel

input_C2

Channel
LakeShoreTemperatureChannel

input_C3

Channel
LakeShoreTemperatureChannel

input_C4

Channel
LakeShoreTemperatureChannel

input_D

Channel
LakeShoreTemperatureChannel

input_D1

Channel
LakeShoreTemperatureChannel

input_D2

Channel
LakeShoreTemperatureChannel

input_D3

Channel
LakeShoreTemperatureChannel

input_D4

Channel
LakeShoreTemperatureChannel

input_D5

Channel
LakeShoreTemperatureChannel

output_1

Channel
LakeShoreHeaterChannel

output_2

Channel
LakeShoreHeaterChannel

output_3

Channel
LakeShoreHeaterChannel

output_4

Channel
LakeShoreHeaterChannel

7.35.5 Lake Shore 421 Gaussmeter

class pymeasure.instruments.lakeshore.**LakeShore421**(*adapter*, *name='Lake Shore 421 Gaussmeter'*, *baud_rate=9600*, ***kwargs*)

Bases: *Instrument*

Represents the Lake Shore 421 Gaussmeter and provides a high-level interface for interacting with the instrument.

```
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

Get whether the alarm is triggered.

property alarm_audible

Control the audible alarm beeper.

property alarm_high

Control the upper setpoint for the alarm mode in the current units. This takes into account the field multiplier.

property alarm_high_multiplier

Get the multiplier for the upper alarm setpoint field.

property alarm_high_raw

Control the upper setpoint for the alarm mode in the current unit and multiplier.

property alarm_in_out

Control 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

Control the lower setpoint for the alarm mode in the current units. This takes into account the field multiplier.

property alarm_low_multiplier

Get the multiplier for the lower alarm setpoint field.

property alarm_low_raw

Control the lower setpoint for the alarm mode in the current units and multiplier.

property alarm_mode_enabled

Control the alarm mode.

property alarm_sort_enabled

Control the alarm Sort Pass/Fail function.

property auto_range

Control 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

Control 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

Control 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

Get the field in the current units. This property takes into account the field multiplier. Get np.nan if field is out of range.

property field_mode

Control whether the gaussmeter measures AC or DC magnetic fields. Valid values are "AC" and "DC".

property field_multiplier

Get the field multiplier for the returned magnetic field.

property field_range

Control (floating) 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

Control (integer) the field range of the meter. Valid values are 0 (highest) to 3 (lowest).

property field_raw

Get the field in the current units and multiplier

property front_panel_brightness

Control (integer) the brightness of the from panel display. Valid values are 0 (dimmest) to 7 (brightest).

property front_panel_locked

Control the lock state of all front panel entries except pressing the Alarm key to silence alarms.

property max_hold_enabled

Control the Max Hold function to store the largest field since the last reset (with max_hold_reset).

property max_hold_field

Get 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

Get the largest field since the last reset in the current units and multiplier.

property max_hold_multiplier

Get the multiplier for the returned max hold field.

max_hold_reset()

Clears the stored Max Hold value.

property probe_type

Get type of field-probe used with the gaussmeter. Possible values are High Sensitivity, High Stability, or Ultra-High Sensitivity.

property relative_field

Get 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

Get the relative field in the current units and the current multiplier.

property relative_mode_enabled

Control the relative mode to see small variations with respect to a given setpoint.

property relative_multiplier

Get the relative field multiplier for the returned magnetic field.

property relative_setpoint

Control the setpoint for the relative field mode in the current units. This takes into account the field multiplier.

property relative_setpoint_multiplier

Get the multiplier for the setpoint field.

property relative_setpoint_raw

Control the setpoint for the relative field mode in the current units and multiplier.

property serial_number

Get the serial number of the probe.

shutdown()

Close the serial connection to the system.

property unit

Control (string) 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.

7.35.6 Lake Shore 425 Gaussmeter

class pymeasure.instruments.lakeshore.**LakeShore425**(*adapter*, *name='LakeShore 425 Gaussmeter'*, ***kwargs*)

Bases: *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

Get 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 mode

Control the mode, filter, and bandwidth settings.

property range

Control the field range in units of Gauss, which can take the values 35, 350, 3500, and 35,000 G. (float)

property unit

Control the units of the instrument, which can take the values of G, T, Oe, or A/m. (str)

zero_probe()

Initiates the zero field sequence to calibrate the probe

7.35.7 LakeShore Channel Classes

Several Lakeshore instruments are channel based and make use of the *Channel Interface*. For temperature monitoring and controller instruments the following common *Channel Classes* are utilized:

class pymeasure.instruments.lakeshore.lakeshore_base.**LakeShoreTemperatureChannel**(*parent*, *id*)

Bases: *Channel*

Temperature input channel on a lakeshore temperature monitor. Reads the temperature in kelvin, celsius, or sensor units. Also provides a method to block the program until a given stable temperature is reached.

property celcius

Get celsius attribute with celcius (sic) property.

Deprecated since version 0.14.0: Use celsius instead.

property celsius

Get the temperature in celsius from a channel.

property kelvin

Get the temperature in kelvin from a channel.

property sensor

Get the temperature in sensor units from a channel.

wait_for_temperature(*target*, *unit='kelvin'*, *accuracy=0.1*, *interval=1*, *timeout=360*, *should_stop=<function LakeShoreTemperatureChannel.<lambda>>*)

Blocks the program, waiting for the temperature to reach the target within the accuracy (%), checking this each interval time in seconds.

Parameters

- **target** – Target temperature in kelvin, celsius, or sensor units.
- **unit** – 'kelvin', 'celsius', or 'sensor' specifying the unit for queried temperature values.
- **accuracy** – An acceptable percentage deviation between the target and temperature.
- **interval** – Interval time in seconds between queries.
- **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

class pymeasure.instruments.lakeshore.lakeshore_base.**LakeShoreHeaterChannel**(*parent*, *id*)

Bases: *Channel*

Heater output channel on a lakeshore temperature controller. Provides properties to query the output power in percent of the max, set the manual output power, heater range, and PID temperature setpoint.

property mout

Control manual heater output in percent.

property output

Get the heater output in percent of the max.

property range

Control heater range, which can take the values: off, low, medium, and high.

property setpoint

Control the setpoint temperature in the preferred units of the control loop sensor.

7.36 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*.

If the instrument you are looking for is not here, also check *Teledyne* for newer instruments.

7.36.1 LeCroy T3DSO1204 Oscilloscope

class pymeasure.instruments.lecroy.**LeCroyT3DSO1204**(*adapter*, *name='LeCroy T3DSO1204 Oscilloscope'*, ***kwargs*)

Bases: *TeledyneOscilloscope*

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.

This implementation is based on the shared base class `TeledyneOscilloscope`.

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-negligible 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()
```

ch_1

> **Channel**
> *LeCroyT3DSO1204Channel*

ch_2

> **Channel**
> *LeCroyT3DSO1204Channel*

ch_3

> **Channel**
> *LeCroyT3DSO1204Channel*

ch_4

> **Channel**
> *LeCroyT3DSO1204Channel*

class BaseChannelCreator(*cls*, ***kwargs*)

Bases: `object`

Base class for ChannelCreator and MultiChannelCreator.

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- ****kwargs** – Keyword arguments for all children.

class ChannelCreator(*cls*, *id=None*, ***kwargs*)

Bases: *BaseChannelCreator*

Add a single channel to the parent class.

The child will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute name that ChannelCreator was assigned to in the *Instrument* class will be the name of the channel interface.

```
class Extreme5000(Instrument):
    # Two output channels, accessible by their property names
    # and both are accessible through the 'channels' collection
    output_A = Instrument.ChannelCreator(Extreme5000Channel, "A")
    output_B = Instrument.ChannelCreator(Extreme5000Channel, "B")
    # A channel without a channel accessible through the 'motor' collection
    motor = Instrument.ChannelCreator(MotorControl)

inst = SomeInstrument()
# Set the extreme_temp for channel A of Extreme5000 instrument
inst.output_A.extreme_temp = 42
```

Parameters

- **cls** – Channel class for channel interface
- **id** – The id of the channel on the instrument, integer or string.
- ****kwargs** – Keyword arguments for all children.

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

Bases: *BaseChannelCreator*

Add channels to the parent class.

The children will be added to the parent instance at instantiation with `CommonBase.add_child()`. The attribute 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 the attribute name and leave the prefix at the default `"ch_"`.

```
class Extreme5000(Instrument):
    # Three channels of the same type: 'ch_A', 'ch_B', 'ch_C'
    # and add them to the 'channels' collection
    channels = Instrument.MultiChannelCreator(Extreme5000Channel, ["A", "B", "C
↪"])
    # Two channel interfaces of different types: 'fn_power', 'fn_voltage'
    # and add them to the 'functions' collection
    functions = Instrument.MultiChannelCreator((PowerChannel, VoltageChannel),
                                    ["power", "voltage"], prefix="fn_")
```

Parameters

- **cls** – Class for all children or tuple/list of classes, one for each child.
- **id** – tuple/list of ids of the channels on the instrument.
- **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. Required if id is tuple/list.

- ****kwargs** – Keyword arguments for all children.

property acquisition_average

Control the averaging 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

Get 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

Get 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

Get 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

Get 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

Get the sample rate of the scope.

property acquisition_status

Get the acquisition status of the scope.

property acquisition_type

Control the type of data acquisition.

Can be 'normal', 'peak', 'average', 'highres'.

add_child(*cls*, *id=None*, *collection='channels'*, *prefix='ch_'*, *attr_name=''*, ***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, e.g. 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 dictionary access to the channel interfaces.
- **prefix** – For creating multiple channel interfaces, the prefix e.g. *"ch_"* is prepended to the attribute name of the channel interface *self.ch_A*. If prefix evaluates False, the child will be added directly under the collection name.
- **attr_name** – For creating a single channel interface, the attr_name argument is used when setting the attribute name of the channel interface.
- ****kwargs** – Keyword arguments for the channel creator.

Returns
Instance of the created child.

ask(*command*, *query_delay=None*)

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=None*, ***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 bwlimit

Set the internal low-pass filter for all channels.(dynamic)

center_trigger()

Set 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.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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>>*, *get_process_list=<function CommonBase.<lambda>>*, *set_process=<function CommonBase.<lambda>>*, *command_process=None*, *check_set_errors=False*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=', '*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **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

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

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()

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.

Parameters

- **source** – measurement source. It can be "C1", "C2", "C3", "C4", "MATH".
- **requested_points** – number of points to acquire. If None the number of points requested in the previous call will be assumed, i.e. the value of the number of points stored in the oscilloscope memory. If 0 the maximum number of points will be returned.
- **sparsing** – interval between data points. For example if sparsing = 4, only one point every 4 points is read. If 0 or None the sparsing of the previous call is assumed, i.e. the value of the sparsing stored in the oscilloscope memory.

Returns

data_ndarray, time_ndarray, waveform_preamble_dict: see waveform_preamble property for dict format.

static get_channel_pairs(*cls*)

Return a list of all the Instrument's channel pairs

static get_channels(*cls*)

Return a list of all the Instrument's ChannelCreator and MultiChannelCreator instances

property grid_display

Control the type of the grid which is used to display (FULL, HALF, OFF).

property id

Get the identification of the instrument.

property intensity

Set the intensity level of the grid or the trace in percent

property math_define

Control 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

Control 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

Control 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

Control measurement 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=False*, *get_process=<function CommonBase.<lambda>>*, *get_process_list=<function CommonBase.<lambda>>*, *command_process=None*, *check_get_errors=False*, *dynamic=False*, *preprocess_reply=None*, *separator=', '*, *maxsplit=-1*, *cast=<class 'float'>*, *values_kwargs=None*, ***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 takes a value and allows processing before value mapping, returning the processed value
- **get_process_list** – A function that takes the value list and processes it.
- **command_process** – A function that take a command and allows processing before executing the command, for getting

  Deprecated since version 0.12: Use a dynamic property instead.
- **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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- **values_kwargs** (*dict*) – Further keyword arguments for *values()*.
- ****kwargs** – Keyword arguments for *values()*.

  Deprecated since version 0.12: Use *values_kwargs* dictionary parameter instead.

property memory_size

Control 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 next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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()

Reset 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*)

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

Get 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

Get 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

Control the zoomed (delayed) window horizontal scale (seconds/div).

The main sweep scale determines the range for this command.

property timebase_hor_position

Control 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

Control the time interval in seconds between the trigger event and the reference position (at center of screen by default).

property timebase_scale

Control the horizontal scale (units per division) in seconds for the main window (float).

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

Get 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

Control the trigger sweep mode (string).

<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

Control the condition that will trigger the acquisition of waveforms (string).

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.
   - {ti} for drop trigger.
   - {ps, pl, p2, p1} for glit/runt trigger.

- {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.

Parameters

- **mode** – trigger sweep mode [auto, normal, single, stop]
- **source** – trigger source [c1, c2, c3, c4, line]
- **trigger_type** – condition that will trigger the acquisition of waveforms [edge,slew,glit,intv,runt,drop]
- **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

values(*command*, *separator=','*, *cast=<class 'float'>*, *preprocess_reply=None*, *maxsplit=-1*, ***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.
- **preprocess_reply** – Optional callable used to preprocess the string received from the instrument, before splitting it. The callable returns the processed string.
- **separator** – A separator character to split the string returned by the device into a list.
- **maxsplit** – The string returned by the device is splitted at most *maxsplit* times. -1 (default) indicates no limit.
- **cast** – A type to cast each element of the splitted string.
- ****kwargs** – Keyword arguments to be passed to the *ask()* method.

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

`wait_for`(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

`property waveform_first_point`

Control the address of the first data point to be sent (int). 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`

Control the number of waveform points to be transferred with the digitize method (int). 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`

Control the interval between data points (integer). For example:

SP = 0 sends all data points. SP = 4 sends 1 point every 4 data points.

write(*command*, ***kwargs*)

Write 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.

class pymeasure.instruments.lecroy.lecroyT3DSO1204.**LeCroyT3DSO1204Channel**(*parent*, *id*)

Bases: *TeledyneOscilloscopeChannel*

Implementation of a LeCroy T3DSO1204 Oscilloscope channel.

Implementation modeled on Channel object of Keysight DSOX1102G instrument.

property bwlimit

Control the 20 MHz internal low-pass filter (strict bool).

This oscilloscope only has one frequency available for this filter.

property invert

Control the inversion of the input signal (strict bool).

property skew_factor

Control the channel-to-channel skew factor for the specified channel. Each analog channel can be adjusted + or -100 ns for a total of 200 ns difference between channels. You can use the oscilloscope's skew control to remove cable-delay errors between channels.

property trigger_level2

Control the lower trigger level voltage for the specified source (float). Higher and lower trigger levels are used with runt/slope triggers. When setting the trigger level it must be divided by the probe attenuation. This is not documented in the datasheet and it is probably a bug of the scope firmware. An out-of-range value will be adjusted to the closest legal value.

property unit

Control the unit of the specified trace. Measurement results, channel sensitivity, and trigger level will reflect the measurement units you select. ("A" for Amperes, "V" for Volts).

7.37 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*.

7.37.1 MKS Instruments Abstract Instrument

class pymeasure.instruments.mksinst.mksinst.**MKSInstrument**(*adapter*, *name='MKS Instrument'*, *address=253*, ***kwargs*)

Bases: *Instrument*

Abstract MKS Instrument

Connection to the device is made through an RS232/RS485 serial connection. The communication protocol of these devices 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

check_set_errors()

Check reply string for acknowledgement string.

read()

Reads from the instrument including the correct termination characters

write(*command*)

Write to the instrument including the device address.

Parameters

command – command string to be sent to the instrument

class pymeasure.instruments.mksinst.mksinst.**RelayChannel**(*parent*, *id*)

Bases: *Channel*

Settings of the optionally included setpoint relay.

The relay is energized either below or above the setpoint depending on the 'direction' property. The relay is de-energized when the reset value is crossed in the opposite direction.

Note that device by default uses an auto hysteresis setting of 10% of the setpoint value that overwrites the current reset value whenever the setpoint value or direction is changed. If other hysteresis value than 10% is required, first set the setpoint value and direction before setting the reset value.

property direction

Control the switching direction

property resetpoint

Control the relay switch off value

property setpoint

Control the relay switch setpoint

property status

Get the setpoint relay status

7.37.2 MKS Instruments 937B Vacuum Gauge Controller

class pymeasure.instruments.mksinst.mks937b.**MKS937B**(*adapter*, *name='MKS 937B vacuum gauge controller'*, *address=253*, ***kwargs*)

Bases: *MKSInstrument*

MKS 937B vacuum gauge controller

Connection to the device is made through an RS232/RS485 serial connection.

The 937B gauge controller can connect up to 6 pressure measurement channels for gauges of various types which includes ionization gauges, Pirani and piezo gauges. Based on the pressure values of the gauges twelve setpoint relays can be energized when certain pressure values are reached. The assignment of the relays to measurement channels is fixed to have two relays for each pressure channel.

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

ch_1

Channel
IonGaugeAndPressureChannel

ch_2

Channel
PressureChannel

ch_3

Channel
IonGaugeAndPressureChannel

ch_4

Channel
PressureChannel

ch_5

Channel
IonGaugeAndPressureChannel

ch_6

Channel
PressureChannel

relay_1

Channel
Relay

relay_10

Channel
Relay

relay_11

Channel
Relay

relay_12

Channel
Relay

relay_2

Channel
Relay

relay_3

Channel
Relay

relay_4

Channel
Relay

relay_5

Channel
Relay

relay_6

Channel
Relay

relay_7

Channel
Relay

relay_8

Channel
Relay

relay_9

Channel
Relay

property all_pressures

Get pressures on all channels in selected units

property combined_pressure1

Get pressure on channel 1 and its combination sensor

property combined_pressure2

Get pressure on channel 2 and its combination sensor

property serial

Get the serial number of the instrument

property unit

Control pressure unit used for all pressure readings from the instrument.

Allowed units are Unit.Torr, Unit.mbar, Unit.Pa, Unit.uHg.

class pymeasure.instruments.mksinst.mks937b.**IonGaugeAndPressureChannel**(*parent*, *id*)

Bases: *PressureChannel*

Channel having both a pressure and an ion gauge sensor

property ion_gauge_status

Get ion gauge status of the channel.

class pymeasure.instruments.mksinst.mks937b.**PressureChannel**(*parent*, *id*)

Bases: *Channel*

property power_enabled

Control power status of the channel. (bool)

property pressure

Get the pressure on the channel in units selected on the device

class pymeasure.instruments.mksinst.mks937b.**Relay**(*parent*, *id*)

Bases: *RelayChannel*

property enabled

Control the relay function or disable the setpoint relay.

Possible values are True/False to enable/disable pressure based activation of the relay and 'SET' to permanently energize the relay.

7.37.3 MKS Instruments 974B Vacuum Pressure Transducer

class pymeasure.instruments.mksinst.mks974b.**MKS974B**(*adapter*, *name='MKS 974B vacuum pressure transducer'*, *address=253*, ***kwargs*)

Bases: *MKSInstrument*

MKS 974B vacuum pressure transducer

Connection to the device is made through an RS232/RS485 serial connection.

The 974B pressure transducer is a pressure gauge combining a cold cathode ionization current measurement, a Pirani sensor, and a Piezo sensor to cover a large pressure range. It optionally includes up to three mechanical relays which can be energized based on the measured pressure value.

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`

relay_1

Channel
Relay

relay_2

Channel
Relay

relay_3

Channel
Relay

property coldcathode_pressure

Get Cold Cathode sensor pressure

property device_type

Get the device type

property firmware_version

Get the firmware version of the instrument

property hardware_version

Get the hardware version of the instrument

id()

Get the identification of the instrument.

property manufacturer

Get the manufacturer name

property model

Get the transducer model number

property operation_hours

Get the operation hours of the instrument

property part_number

Get the transducer part number

property piezo_pressure

Get Piezo differential sensor pressure

property pirani_pressure

Get MicroPirani sensor pressure

property pressure

Get combined pressure reading

property serial_number

Get the serial number of the instrument

property status

Get transducer status

property switch_enabled

Control the user switch to prevent accidental execution of adjustments

property temperature

Get the MicroPirani sensor temperature

property unit

Control pressure unit used for all pressure readings from the instrument.

Allowed units are Unit.Torr, Unit.mbar, Unit.Pa.

property user_tag

Control the user programmable tag

class pymeasure.instruments.mksinst.mks974b.**Relay**(*parent*, *id*)

Bases: *RelayChannel*

property enabled

Control the assigned input channel or disable the setpoint relay.

7.38 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*.

7.38.1 ESP 300 Motion Controller

class pymeasure.instruments.newport.**ESP300**(*adapter*, *name='Newport ESP 300 Motion Controller'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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 available axes. Axes are controlled through an *Axis* class.

property axes

Get a list of the *Axis* objects that are present.

clear_errors()

Clears the error messages by checking until a 0 code is received.

disable()

Disables all of the axes associated with this controller.

enable()

Enables all of the axes associated with this controller.

property error

Get an error code from the motion controller.

property errors

Get 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: encoder 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.

7.39 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*.

7.39.1 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)

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: 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: `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: 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: `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: 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`

Represents National Instruments Virtual Bench main frame. This class provides direct access to the armstrap/pyvirtualbench Python wrapper.

7.40 Novanta Photonics

This section contains specific documentation on the Novanta photonics instruments that are implemented. Novanta contains also Lasers developed by Laserquantum. If you are interested in an instrument not included, please consider *adding the instrument*.

7.40.1 Novanta FPU60 laser power supply unit

class pymeasure.instruments.novanta.**Fpu60**(*adapter*, *name='Laserquantum fpu60 power supply unit'*, ***kwargs*)

Bases: *Instrument*

Represents a fpu60 power supply unit for the finesse laser series by Laserquantum, a Novanta company.

The instrument responds to every command sent.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

Returns
List of error entries.

property current

Measure the diode current in percent (float).

disable_emission()

Disable emission and unlock the button afterwards.

You have to press the physical button to enable emission again.

property emission_enabled

Measure the emission status (bool).

get_operation_times()

Get the operation times in minutes as a dictionary.

property head_temperature

Measure the laser head temperature in °C (float).

property interlock_enabled

Get the interlock enabled status (bool).

property power

Measure current output power in Watts (float).

property power_setpoint

Control the output power setpoint in Watts (float).

property psu_temperature

Measure the power supply unit temperature in °C (float).

property serial_number

Get the serial number (str).

property shutter_open

Control whether the shutter is open (bool).

property software_version

Get the software version (str).

7.41 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*.

7.41.1 Oxford Instruments Base Instrument

class pymeasure.instruments.oxfordinstruments.base.**OxfordInstrumentsBase**(*adapter*, *name='OxfordInstruments Base'*, *max_attempts=5*, ***kwargs*)

Bases: *Instrument*

Base instrument for devices from Oxford Instruments.

Checks the replies from instruments for validity.

Parameters

- **adapter** – A string, integer, or *Adapter* subclass object
- **name** (*string*) – The name of the instrument. Often the model designation by default.
- **max_attempts** – Integer that sets how many attempts at getting a valid response to a query can be made
- ****kwargs** – In case `adapter` is a string or integer, additional arguments passed on to *VISAAdapter* (check there for details). Discarded otherwise.

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

write(*command*)

Write command to instrument and check whether the reply indicates that the given command was not understood. The devices from Oxford Instruments reply with '?xxx' to a command 'xxx' if this command is not known, and replies with 'x' if the command is understood. If the command starts with an "$" the instrument will not reply at all; hence in that case there will be done no checking for a reply.

Raises

OxfordVISAError if the instrument does not recognise the supplied command or if the response of the instrument is not understood

class pymeasure.instruments.oxfordinstruments.base.**OxfordVISAError**

Bases: Exception

7.41.2 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: *OxfordInstrumentsBase*

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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: 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.

7.41.3 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: *OxfordInstrumentsBase*

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
```

(continues on next page)

(continued from previous page)

```
                            # 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.

7.41.4 Oxford Instruments Mercury Intelligent Temperature Controller

class pymeasure.instruments.oxfordinstruments.**MercuryiTC**(*adapter*, *name='Oxford MercuryiTC'*, ***kwargs*)

Represents the Oxford Instruments Mercury iTC (Intelligent Temperature Controller) and provides a high-level interface for interacting with the instrument. Note thath the GPIB implementation on this instrument is not fully SCPI-compliant, with only the `*IDN?` command supported. The Mercury iTC can be configured to respond to a so-called SCPI instruction set (that is, again, not actually an SCPI implementation) or to a legacy instruction set. The implementation used here is the so-called SCPI instruction set.

The iTC is expandable and supports a mixture temperature sensors, heaters, pressure sensors, cryogen level sensors and auxiliary I/O through the installation of additional, function-specific daughter boards. Currently, only temperature sensors and heaters are implemented.

By default, there is a temperature sensor channel installed with Unique Identifier (`UID`) `'MB1.T1'` and a heater channel with UID `'MB0.H1'`. These channels are created automatically at instantiation of the controller. These

can be accessed as either `.TS_MB` and `.HTR_MB`, or through the collections (dictionaries) `.TS['MB1.T1']` and `.HTR['MB0.H1']`.

Other sensors and heaters can be installed on additional daughter boards (with specific `UID` e.g. `'DBX.T1'`, `'DBX.H1'` etc), and can be added manually to an already instantiated MercuryiTC using `.add_temperature_sensor(UID)` and `.add_heater(UID)`. They must be given their correct UID as listed by the Mercury iTC itself as this is how they are addressed during communication. Additionally, they are accessed **only** through the instrument collections `.TS[UID]` and `.HTR[UID]`.

```
controller = MercuryiTC("GPIB::1")

# Print the current measured temperature of the motherboard temperature sensor aka
↪'MB1.T1'
print(controller.TS_MB.temperature)

# Print the motherboard temperature sensor control-loop setpoint
print(controller.TS['MB1.T1'].loop_T_setpoint)

# Add new temperature sensor with UID 'DB6.T1'
controller.add_temperature_sensor('DB6.T1')

# Print the new sensor's temperature
print(controller.TS['DBS.T1'].temperature)
```

HTR_MB

Channel
Heater

TS_MB

Channel
TemperatureSensor

add_heater(*id*, *prefix='HTR'*, ***kwargs*)

Add a heater with unique identifier `id` to collection of addressable heaters: `.HTR[id]`

add_temperature_sensor(*id*, *prefix='TS'*, ***kwargs*)

Add a temperature sensor with unique identifier `id` to collection of addressable temperature sensors: `.TS[id]`

property identity

Get identity of unit.

class pymeasure.instruments.oxfordinstruments.mercuryitc.**TemperatureSensor**(*parent*, *id*)

Bases: *Channel*

A temperature sensor channel on the Oxford Instruments Mercury iTC.

Depending on sensor type (e.g. diode, NTC/PTC resistor), some properties will not be valid. Therefore, sensor configuration must be known and set a priori. Only diode sensor types are currently implemented.

Control loops are accessed via temeperature sensors, rather than heaters. Therefore loop control is kept with other temperature sensor properties.

check_set_errors()

The MercuryiTC responds to every communication (get and set). This simply reads back on setting a property.

property control_loop_D

Control the derivative term, D, of the control loop.

property control_loop_I

Control the integral term, I, of the control loop.

property control_loop_P

Control the proportional term, P, of the control loop.

property control_loop_PID_enabled

Control whether the control-loop heater is controlled by the PID loop (`True`) or the manual heater percentage (`False`).

property control_loop_heater_percent

Control the heater percentage when output configured to manual mode (in the range 0 to 100).

property control_loop_ramp_enabled

Control whether the temperature ramping function is enabled (`True`) or disabled (`False`).

property control_loop_ramp_rate

Control the control loop ramp rate, in K/min, if enabled (in the range 0 to 100000).

property control_loop_temperature_setpoint

Control the control loop temperature setpoint, in Kelvin (in the range 0 to 2000).

property temperature

Get the measured temperature, in Kelvin.

property voltage

Get the sensor voltage, in Volts.

class pymeasure.instruments.oxfordinstruments.mercuryitc.**Heater**(*parent*, *id*)

Bases: *Channel*

A heater channel on the Oxford Instruments Mercury iTC.

Various parameters of the heater output can be accessed, and some properties can be configured. PID control is accessed through an associated temperature sensor. It is assumed that the heater is already associated with a temperature sensor control loop, or otherwise set to operate in open-loop configuration.

check_set_errors()

The MercuryiTC responds to every communication (get and set). This simply reads back on setting a property.

property current

Get the heater excitation current, in Amps.

property max_power

Get the provisional maximum power of the heater, in Watts.

property power

Get the heater power dissipation, in Watts.

property resistance

Control the programmed heater resistance, in Ohms (float strictly in the range of 10 to 2000).

property voltage

Get the heater excitation voltage, in Volts.

property voltage_limit

Control the voltage limit of the heater output, in Volts (float strictly from 0 to 40).

7.41.5 Oxford Instruments Power Supply 120-10 for superconducting magnets

class pymeasure.instruments.oxfordinstruments.**PS120_10**(*adapter*, *name='Oxford PS'*, ***kwargs*)

Bases: *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.

7.42 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*.

7.42.1 Parker GV6 Servo Motor Controller

class pymeasure.instruments.parker.**ParkerGV6**(*adapter*, *name='Parker GV6 Motor Controller'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Parker Gemini GV6 Servo Motor Controller and provides a high-level interface for interacting with the instrument

property acceleration

Set the acceleration setpoint in revolutions per second squared.

property angle

Control the angle in degrees based on the position and whether relative or absolute positioning is enabled,

property angle_error

Get the angle error in degrees based on the position error, or returns None on error

property average_acceleration

Set the average acceleration setpoint in revolutions per second squared.

disable()

Disables the motor from moving

property echo

Set whether the echoing of all commands sent to the instrument is enabled (bool).

enable()

Enables the motor to move

is_moving()

Return True if the motor is currently moving

kill()

Stop the motor

move()

Initiate the motor to move to the setpoint

property position

Control the angular position in counts, 4000 per revolution (int).

property position_error

Get the error in the number of counts that corresponds to the error in the angular position where 1 revolution equals 4000 counts

read()

Overwrite the Instrument.read command to provide the correct functionality

reset()

Reset the motor controller while blocking and (CAUTION) resets the absolute position value of the motor

set_defaults()

Set up the default values for the motor, which is run upon construction

set_hardware_limits(*positive=True*, *negative=True*)

Enable (True) or disables (False) the hardware limits for the motor

set_software_limits(*positive*, *negative*)

Set the software limits for motion based on the count unit where 4000 counts is 1 revolution

property status

Get a list of the motor status in readable format

stop()

Stop the motor during movement

use_absolute_position()

Set the motor to accept setpoints from an absolute zero position

use_relative_position()

Set the motor to accept setpoints that are relative to the last position

property velocity

Set the velocity setpoint in revolutions per second.

7.43 Philips

This section contains specific documentation on the Philips instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.`.

7.43.1 Philips PM 6669 Universal Frequency Counter

class pymeasure.instruments.philips.**PM6669**(*adapter*, *name='Philips PM 6669'*, ***kwargs*)

Bases: *Instrument*

Represents the Philips PM 6669 instrument.

property SRQMask

Control the SRQ mask

property freerun_enabled

Control the freerun settings

property gate_enabled

Control the gate

In the totalize function, set this to True to open the gate and to False to close the gate. The count can then be read with read_measurement(). For most purposes you want to be in freerun mode.

property id

Get the instrument identification

property measurement_settings

Get the measurement settings from the device

property measurement_time

Control the measurement time in seconds

property measurement_timeout

Control the measurement timeout

this timeout only has meaning when freerun is off.

property measuring_function

Control the measuring function on the device (str).

read()

Read function with instrument bug work around.

If a request is made to the device while a measurement is ready, both the reply and the measurement are returned. The measurement is filtered out and put in a backlog Queue.

read_measurement(*wait_for_srq=False*)

Wait for an SRQ from the device and then reads the result.

If *wait_for_srq* is set, MSR to be set to MSRFlag.MEASUREMENT_READY

reset_to_defaults()

Reset the instruments to default settings

spoll()

Read the status of the device.

trigger()

Trigger the device when not in freerun mode.

7.44 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*.

7.44.1 Pendulum CNT91 frequency counter

class pymeasure.instruments.pendulum.cnt91.**CNT91**(*adapter*, *name='Pendulum CNT-91'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents a Pendulum CNT-91 frequency counter.

property batch_size

Get maximum number of buffer entries that can be transmitted at once.

buffer_frequency_time_series(*channel*, *n_samples*, *sample_rate=None*, *gate_time=None*, *trigger_source=None*, *back_to_back=True*)

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
- **gate_time** – Gate time in s
- **trigger_source** – Optionally specify a trigger source to start the measurement
- **back_to_back** – If True, the buffer measurement is performed back-to-back.
- **sample_rate** – Sample rate in Hz

  Deprecated since version 0.14: Use parameter *gate_time* instead.

configure_frequency_array_measurement(*n_samples*, *channel*, *back_to_back=True*)

Configure the counter for an array of measurements.

Parameters

- **n_samples** – The number of samples
- **channel** – Measurement channel (A, B, C, E, INTREF)
- **back_to_back** – If True, the buffer measurement is performed back-to-back.

property continuous

Control whether to perform continuous measurements.

property external_arming_start_slope

Control slope for the start arming condition (str 'POS' or 'NEG').

property external_start_arming_source

Control external arming source ('A', 'B', 'E' (rear) or 'IMM' for immediately arming).

property format

Control response format ('ASCII' or 'REAL').

property gate_time

Control gate time of one measurement in s (float strictly from 2e-8 to 1000).

property interpolator_autocalibrated

Control if interpolators should be calibrated automatically (bool).

property measurement_time

Control gate time of one measurement in s (float strictly from 2e-8 to 1000).

Deprecated since version 0.14: Use *gate_time* instead.

read_buffer(*n=10000*)

Read out *n* samples from the buffer.

Parameters

n – Number of samples that should be read from the buffer. The maximum number of 10000 samples is read out by default.

Returns

Frequency values from the buffer.

7.45 Proterial

This section contains specific documentation on the Proterial (formerly Hitachi Metals) instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.45.1 ROD-4 MFC Controller

class pymeasure.instruments.proterial.**ROD4**(*adapter*, *name='ROD-4 MFC Controller'*, ***kwargs*)

Bases: *Instrument*

Represents the Proterial ROD-4(A) operator for mass flow controllers and provides a high-level interface for interacting with the instrument. User must specify which channel to control (1-4).

```
rod4 = ROD4("ASRL1::INSTR")

print(rod4.version)              # Print version and series number
rod4.ch_1.mfc_range = 500        # Sets Channel 1 MFC range to 500 sccm
rod4.ch_2.valve_mode = 'flow'    # Sets Channel 2 MFC to flow control
rod4.ch_3.setpoint = 50          # Sets Channel 3 MFC to flow at 50% of full range
print(rod4.ch_4.actual_flow)     # Prints Channel 4 actual MFC flow in %
```

ch_1

Channel
ROD4Channel

ch_2

Channel
ROD4Channel

ch_3

Channel
ROD4Channel

ch_4

Channel
ROD4Channel

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Read 'OK' from ROD-4 after setting.

clear()

Clears the instrument status byte

property complete

Get the synchronization bit.

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

Get the identification of the instrument.

property keyboard_locked

Set the front keyboard lock status.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get 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

Get the status byte and Master Summary Status bit.

property version

Get the version and series number. Returns x.xx<TAB>S/N

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.proterial.rod4.**ROD4Channel**(*parent*, *id*)

Bases: *Channel*

Implementation of a ROD-4 MFC channel.

property actual_flow

Measure the actual flow in %.

property flow_unit_display

Set the flow units on the front display. Valid options are %, sccm, or slm. Display in absolute units is in sccm for control range < 10 slm.

property mfc_range

Control the MFC range in sccm. Upper limit is 200 slm.

property ramp_time

Control the MFC setpoint ramping time in seconds.

property setpoint

Control the setpoint in % of MFC range.

property valve_mode

Control the MFC valve mode. Valid options are *flow*, *close*, and *open*.

7.46 PTW Freiburg

This section contains specific documentation on the PTW Freiburg GmbH instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.46.1 PTW DIAMENTOR DAP dosemeters

class pymeasure.instruments.ptw.**ptwDIAMENTOR**(*adapter*, *name='PTW DIAMENTOR DAP dosemeter'*, *baud_rate=9600*, ***kwargs*)

Bases: *Instrument*

A class representing the PTW DIAMENTOR DAP dosemeters.

property baudrate

Control the baudrate.

Type
int, strictly in `9600`, `19200`, `38400`, `57600`, `115200`

The baudrate is changed after sending the respone.

property calibration_factor

Control the calibration factor of the measurement chamber in µGy*m^2/s.

Type
float, strictly from `1E8` to `9.999E12`, default: `1.0E9`

The unit of the calibration factor is always µGy*m^2/s. It is independent from the selected *dap_unit*.

> **Warning**
>
> Changing the calibration factor can lead to wrong measurements!
>
> The calibration factor of the last calibration can be found on a label on the case or in the calibration certificate of the DIAMENTOR chamber.

check_set_errors()

Check for errors after sending a command.

property constancy_check_passed

Get the DIAMENTOR electrical constancy check result (bool).

property correction_factor

Control the correction factor of the chamber.

Type
float, strictly from `0` to `9.999`, default: `1.0`

property dap_unit

Control the dose-area-product (DAP) unit.

Type
str, strictly in `cGycm2`, `Gycm2`, `uGym2`, `Rcm2`

- `cGycm2` selects cGy*cm^2
- `Gycm2` selects Gy*cm^2
- `uGym2` selects µGy*m^2
- `Rcm2` selects R*cm^2

`execute_selftest()`

Execute the DIAMENTOR selftest.

Raises
ValueError if selftest fails

`property id`

Get the firmware version (str) ("CRS x.xx")

`property is_calibrated`

Get the calibration status (bool).

`property is_eeprom_ok`

Get the EEPROM CRC ok status (bool).

`property measurement`

Get the measurement result.

Returns
dict

Dict keys
`dap`, `dap_rate`, `time`

The result consists of the dose-area-product (DAP) and the dose-area-product rate. The units of `dap` and `dap_rate` depend on the *dap_unit* property. Time is in seconds.

`property pressure`

Control the atmospheric pressure in hPa.

Type
int, strictly from `500` to `1500`, default: `1013`

It is used for the air density correction.

`read()`

Read the device response and check for errors.

Returns
str

Raises
ValueError for error response or *ConnectionError* for an unknown error

`reset()`

Reset the dose and charge measurement values.

`property serial_number`

Get the serial number (int).

`property temperature`

Control the chamber temperature in degree Celsius.

Type
int, strictly from `0` to `70`, default: `20`

It is used for the air density correction.

7.46.2 PTW UNIDOS Tango/Romeo dosemeters

class pymeasure.instruments.ptw.**ptwUNIDOS**(*adapter*, *name='PTW UNIDOS dosemeter'*, ***kwargs*)

Bases: *Instrument*

A class representing the PTW UNIDOS Tango/Romeo dosemeters.

property autoreset_enabled

Control the measurement autoreset function (bool).

property autostart_enabled

Control the measurement autostart function (bool).

property autostart_level

Control the threshold level of autostart measurements (str strictly in “LOW”, “MEDIUM”, “HIGH”).

Type
str, strictly in `LOW`, `MEDIUM`, `HIGH`

check_set_errors()

Check for errors after sending a command.

clear()

Clear the complete measurement history.

> **Note**
>
> Write permission is required.

property electrical_units_enabled

Control whether electrical units are used (bool).

errorflags_to_text(*flags*)

Convert the error flags to the error message(s).

Parameters
flags (*str*)

hold()

Set the measurment to HOLD state.

> **Note**
>
> Write permission is required.

property id

Get the dosemeter ID.

Returns
list of str

property integration_time

Control the time of the interval measurement in seconds (strictly from 1 to 3599999).

Type
int, strictly from 1 to `3599999`

intervall(*intervall=None*)

Execute an intervall measurement.

Parameters
intervall (*int*) – optional, measurement intervall in seconds

If *intervall* is not specified a measurement with the current setting of the *integration_time* is executed.

> **Note**
> Write permission is required.

property lan_config

Get the ethernet configuration.

Returns
dict

Dict keys
`dns`, `ipv4`, `ipv6`

property mac_address

Get the dosemeter MAC address.

Returns
str

measure()

Start the dose or charge measurement.

> **Note**
> Write permission is required.

property measurement_history

Get the measurement history.

Returns
list of dict

Dict keys
`date`, `detector`, `dose`, `flags`, `id`, `kTotal`

property measurement_parameters

Get the measurement parameters.

Returns
dict

Dict keys
`correction`, `detectorGuid`, `highResolution`, `hv`, `integrationTime`, `isElectrical`, `limitDose`, `limitDoseEnabled`, `limitDoseRate`, `limitDoseRateEnabled`, `range`, `triggerAuto`, `triggerReset`, `triggerSensitivity`

property measurement_result

Get the measurement results.

Returns
dict

Dict keys
status, charge, dose, current, doserate, timebase, time, voltage, error

property range

Control the measurement range (str strictly in "VERY_LOW", "LOW", "MEDIUM", "HIGH").

Type
str, strictly in VERY_LOW, LOW, MEDIUM, HIGH

property range_max

Get the max value of the current measurement range.

Returns
dict

Dict keys
range, current, doserate, timebase

property range_res

Get the resolution of the measurement range.

Returns
dict

Dict keys
range, charge, dose, current, doserate, timebase

read()

Read the device response and check for errors.

Returns
Read string with semicolons replaced by comma

Raises
ValueError for error response or *ConnectionError* for an unknown error

read_detector(*guid='ALL'*)

Read the properties of the requested detector.

Parameters
guid (*str*) – optional, ID of the detector. A list of all detectors is returned if *guid* is not passed.

Type
dict or list of dict

Dict keys
calibrationFactor, calibrationFactorUnit, calibrationLab, calibrationProcedure, calibrationTemp, calibrationType, calibrationUser, comment, guid, manufacturer, maxHv, measuredQuantity, minHv, name, nominalHv, polarity, radiationQuality, range, serialNumber, testDate, testLimits, testResult, testTarget, timebase, type, typeNumber

reset()

Reset the dose and charge measurement values.

> **Note**
>
> Write permission is required.

selftest()

Execute the electrometer selftest.

The function returns before the end of the selftest. End and result of the self test have to be requested by the *selftest_result* property.

> **Note**
>
> Write permission is required.

property selftest_result

Get the dosemeter selftest status and result.

Returns
dict

Dict keys
status, time_remaining, time_total, low, medium, high

property serial_number

Get the dosemeter serial number.

Returns
int

property status

Get the measurement status.

Returns
str

The status string has of one of the following values: RES, MEAS, HOLD, INT, INTHLD, ZERO, AUTO, AUTO_MEAS, AUTO_HOLD, EOM, WAIT, INIT, ERROR, SELF_TEST or TST

property system_info

Get the system information.

Returns
dict

Dict keys
calibrationDate, debugBuild, features, hardwareVersion, hostname, lastTest, macAddress, nextCalibration, serialNumber, softwareVersion, testTemperature, typeNumber

property system_settings

Get the system settings.

Returns
dict

Dict keys
`brightness`, `date`, `keyboardSound`, `locale`, `mark`, `startAsMaster`, `time`, `timezone`, `volume`

property tfi

Get the telegram failure information.

Returns
str

The property provides information about the last failed command with HTTP request.

property voltage

Control the detector voltage in Volts.

Type
int, strictly from `-400` to `400` and within detector limits

The Limits of the detector are applied.

property wlan_config

Get the WLAN access point configuration.

Returns
dict

Dict keys
`enabled`, `ssid`

property write_enabled

Control the write permission (bool).

The result of the request/release has to be checked afterwards.

Important

The UNIDOS has to be in viewer mode (closed lock symbol on the display).

zero()

Execute a zero correction measurement.

The function returns before the end of the zero correction measurement. The end of the zero correction measurement has to be requested by the *zero_status* property.

Note

Write permission is required.

property zero_status

Get the status of the zero correction measurement.

Returns
dict

Dict keys
`status`, `time_remaining`, `time_total`

7.47 Racal-Dana

This section contains specific documentation on the Raca-Dana instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.47.1 Racal-Dana 1992 Universal Counter

class pymeasure.instruments.racal.**Racal1992**(*adapter*, *name='Racal-Dana 1992'*, ***kwargs*)

Bases: *Instrument*

Represents the Racal-Dana 1992 Universal counter

```
from pymeasure.instruments.racal import Racal1992
counter = Racal1992("GPIB0::10")
```

This class should also work for Racal-Dana 1991, it has the same product manual, as long as you don't use functionality that requires channel B.

channel_settings(*channel_name*, ***settings*)

Set channel configuration paramters.

Parameters

- **channel_name** – 'A' or 'B'
- **settings** – one or multiple of the following:

  'coupling' : 'AC' or 'DC' 'attenuation' : 'X1' or 'X10' 'trigger' : 'auto' or 'manual' 'impedance' : '50' or '1M' 'slope' : 'pos' or 'neg' 'filtering' : True or False (only allowed for channel A) 'input_select' : 'separate' or 'common' (only allowed for channel B) 'trigger_level' : <floating point number>

static decode(*v*, *allowed_types=None*)

Decode received message.

All values returned follow the same format: 2 letters to indicate the type of the value returned, followed by a floating point number (which could be an integer, of course.) This here, for example, is math constant Z: MZ+001.00000000E+00

property delay_enable

Control delay. True=enable, False=disable

property delay_time

Control delay time.

property device_type

Get unit device type. Should return 1992 for a Racal-Dana 1992 or 1991 for a Racal-Dana 1991.

property gpib_software_version

Get GPIB software version

property math_mode

Set math mode. True=enable, False=disable

property math_x

Control math constant X.

property math_z

Control math constant Z.

property measured_value

Get measured value.

A Racal-Dana 1992 doesn't return measurement data after a request for measurement data. Instead, it fills a FIFO with data whenever it completes a measurement. When the FIFO is full, the oldest measurement is removed.

The FIFO buffer gets cleared when a command is received that requires an immediate reply, such reading a setting. It also gets cleared when an operating mode is cleared.

When there is no measurement data, this property will stall until data is available. It will also timeout after a time that can be set with the standard pyvisa API.

One can make sure that measurement data is available by first calling *wait_for_measurement()*.

property operating_mode

Set operating mode.

Permitted modes are:

'self_check', 'frequency_a', 'period_a', 'phase_a_rel_b', 'ratio_a_to_b', 'ratio_c_to_b', 'interval_a_to_b', 'total_a_by_b', 'frequency_c'

preset()

Configure instrument with default presets.

read()

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

reset_measurement()

Reset ongoing measurement.

property resolution

Control the resolution of the counter with an integer from 3 to 10 that specifies the number of significant digits.

property software_version

Get instrument software version

property special_function_enable

Control special function. True=enable, False=disable

property special_function_number

Control special function.

property total_so_far

Get total number of events so far.

property trigger_level_a

Control trigger level for channel A

property trigger_level_b

Control trigger level for channel B

wait_for_measurement(*timeout=None*, *progressDots=False*)

Wait until a new measurement is available.

Parameters

- **timeout** – number of seconds to wait before timeout exception.
- **progressDots** – when true, print '.' after each ready-check

write(*s*)

Add a space in front of all commands that are sent to the instrument to work around weird model issue.

It shouldn't be needed on almost all devices, but it also doesn't hurt. And it fixes a real issue that's seen on a few devices.

7.48 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*.

7.48.1 Razorbill RP100 custrom power supply for Razorbill Instrums stress & strain cells

class pymeasure.instruments.razorbill.**razorbillRP100**(*adapter*, *name='Razorbill RP100 Piezo Stack Powersupply'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

Get the current in amps present at the front panel output of channel 1

property contact_current_2

Get the current in amps present at the front panel output of channel 2

property contact_voltage_1

Get the Voltage in volts present at the front panel output of channel 1

property contact_voltage_2

Get the Voltage in volts present at the front panel output of channel 2

property instant_voltage_1

Get the instantaneous output of source one in volts

property instant_voltage_2

Get the instantaneous output of source two in volts

property output_1

Control output of channel 1 on or off

property output_2

Control output of channel 2 on or off

property slew_rate_1

Control the source slew rate in volts/sec of channel 1

property slew_rate_2

Control the source slew rate in volts/sec of channel 2

property voltage_1

Control the output voltage of channel 1

property voltage_2

Control the output voltage of channel 2

7.49 Redpitaya

This section contains specific documentation on the Redpitaya instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.49.1 Redpitaya board for analog signal acquisition and generation as well as digital input/output

class pymeasure.instruments.redpitaya.redpitaya_scpi.**RedPitayaScpi**(*adapter=None*, *ip_address: str = '169.254.134.87'*, *port: int = 5000*, *name='Redpitaya SCPI'*, *read_termination='\r\n'*, *write_termination='\r\n'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

This is the class for the Redpitaya reconfigurable board

The instrument is accessed using a TCP/IP Socket communication, that is an adapter in the form: "TCPIP::x.y.z.k::port::SOCKET" where x.y.z.k is the IP address of the SCPI server (that should be activated on the board) and port is the TCP/IP port number, usually 5000

To activate the SCPI server, you have to connect first the redpitaya to your computer/network and enter the url address written on the network plug (on the redpitaya). It should be something like "RP-F06432.LOCAL/" then browse the menu, open the Development application and activate the SCPI server. When activating the server, you'll be notified with the IP/port address to use with this Instrument.

Parameters

- **ip_address** – IP address to use, if *adapter* is None.
- **port** – Port number to use, if *adapter* is None.

analog_in

Channels
ain1: AnalogInputFastChannel, ain2: AnalogInputFastChannel

analog_in_slow

Channels
ainslow0: AnalogInputSlowChannel, ainslow1: AnalogInputSlowChannel, ainslow2: AnalogInputSlowChannel, ainslow3: AnalogInputSlowChannel

analog_out_slow

Channels
aoutslow0: AnalogOutputSlowChannel, aoutslow1: AnalogOutputSlowChannel, aoutslow2: AnalogOutputSlowChannel, aoutslow3: AnalogOutputSlowChannel

dioN

Channels
dioN0: DigitalChannelN, dioN1: DigitalChannelN, dioN2: DigitalChannelN, dioN3: DigitalChannelN, dioN4: DigitalChannelN, dioN5: DigitalChannelN, dioN6: DigitalChannelN

dioP

Channels
dioP0: DigitalChannelP, dioP1: DigitalChannelP, dioP2: DigitalChannelP, dioP3: DigitalChannelP, dioP4: DigitalChannelP, dioP5: DigitalChannelP, dioP6: DigitalChannelP

led

Channels
led0: DigitalChannelLed, led1: DigitalChannelLed, led2: DigitalChannelLed, led3: DigitalChannelLed, led4: DigitalChannelLed, led5: DigitalChannelLed, led6: DigitalChannelLed, led7: DigitalChannelLed

property acq_buffer_filled

Get the status of the buffer(bool), if True the buffer is full

property acq_format

Set the format of the retrieved buffer data (str), either 'BIN', or 'ASCII' (default)

property acq_trigger_delay_ns

Control the trigger delay in nanoseconds (int) in the range [-8192, 8192] / CLOCK

property acq_trigger_delay_samples

Control the trigger delay in number of samples (int) in the range [-8192, 8192]

property acq_trigger_level

Control the level of the trigger in volts The allowed range should be dynamically set depending on the gain settings either +-LV_MAX or +- HV_MAX (dynamic)

property acq_trigger_position

Get the position within the buffer where the trigger event happened

property acq_trigger_source

Set the trigger source (str), one of RedPitayaScpi.TRIGGER_SOURCES. PE and NE means respectively Positive and Negative edge

property acq_trigger_status

Get the trigger status (bool), if True the trigger as been fired (or is disabled)

property acq_units

Control the output data units (str), either 'RAW', or 'VOLTS' (default)

analog_reset()

Reset the voltage of all analog channels

property average_skipped_samples

Control the use of skipped samples (if decimation > 1) to average the returned acquisition array (bool)

property board_name

Get the RedPitaya board name

property buffer_length

Measure the size of the buffer, that is the number of points of the acquisition

property date

Control the date on board date should be given as a datetime.date object

property decimation

Control the decimation (int) as 2**n with n in range [0, 16] The sampling rate is given as 125MS/s / decimation

digital_reset()

Reset the state of all digital lines

property time

Control the time on board time should be given as a datetime.time object

7.50 Rigol

This section contains specific documentation on the Rigol instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.50.1 Rigol DG800 waveform generator

class pymeasure.instruments.rigol.**DG800**(*adapter*, *name='DG800'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents a Rigol DG800-series waveform generator and provides a high-level for interacting with the instrument

channel_1

Channel
VoltageChannel

channel_2

Channel
VoltageChannel

class pymeasure.instruments.rigol.rigol_dg800.**VoltageChannel**(*parent*, *id*)

Bases: *Channel*

Represents a channel of the signal generator.

property dc

Set the waveform generator to output a DC voltage in V.

property duty_cycle

Control the duty cycle of pulses out of the waveform generator. (float)

property frequency

Control the output frequency (Hz)

property high_impedance

Control the output impedance to be HighZ. This means displayed voltages will be internally converted to correspond that seen by an infinite load, not 50 Ohms. See p. 82 of the manual for more information.

property load

Control the output impedance of a given channel. Returns 9.9e+37 for infinite impedance.

property noise

Set the waveform generator to output noise of specified parameters :param ampl: (float) The peak to peak amplitude of the noise in V :param offset: (float) The DC offset of the noise in V

property output_enabled

Control the status of the channel. True if the channel is on and False if not.

property pulse

Set the waveform generator to output a pulse wave of specified parameters Note that the duty cycle is specified using the duty_cycle method. :param int freq: The frequency of the wave in Hz :param float ampl: The peak to peak amplitude of the wave in V :param float offset: The DC offset of the wave in V :param float phase: The phase offset of the wave in degrees

property pulse_width

Control the pulse width of pulses (in seconds) out of the waveform generator.

property shape

Control the waveform type of the specified channel.

property sine

Set the waveform generator to output a sine of specified parameters :param int freq: The frequency of the sine in Hz :param float ampl: The peak to peak amplitude of the sine in V :param float offset: The DC offset of the sine in V :param float phase: The phase offset of the sine in degrees

property square

Set the waveform generator to output a square wave of specified parameters :param int freq: The frequency of the wave in Hz :param float ampl: The peak to peak amplitude of the wave in V :param float offset: The DC offset of the wave in V :param float phase: The phase offset of the wave in degrees

property sync_enabled

Control the synchronization flag. True if the channel triggers the synchronization connector and False if not

property triangle

Set the waveform generator to output a triangle wave of specified parameters :param int freq: The frequency of the wave in Hz :param float ampl: The peak to peak amplitude of the wave in V :param float offset: The DC offset of the wave in V :param float phase: The phase offset of the wave in degrees

property waveform

Get a descriptor of the waveform applied.

7.51 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*.

7.51.1 R&S SFM TV test transmitter

class pymeasure.instruments.rohdeschwarz.sfm.**SFM**(*adapter*, *name='Rohde&Schwarz SFM'*, ***kwargs*)

Bases: *SCPIMixin*, *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 implemented.

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 information 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	Continuous 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	continuous 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 artificial bit error rate to be applied

property nicam_bit_error_rate

A float property that controls the artificial 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	Continuous 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 information 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 information 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

refer 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.

7.51.2 R&S FS Series spectrum analyzer

Connecting to an R&S FSL 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()
```

(continues on next page)

(continued from previous page)

```
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.fsseries.**FSSeries**(*adapter*, *name='Rohde&Schwarz FS series instrument'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents a Rohde&Schwarz FS Series of spectrum analyzers like FSL and FSW.

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

Control attenuation in dB.

continue_single_sweep()

Continue with single sweep with synchronization.

property continuous_sweep_enabled

Control 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

Control center frequency in Hz.

property freq_span

Control frequency span in Hz.

property freq_start

Control start frequency in Hz.

property freq_stop

Control stop frequency in Hz.

property res_bandwidth

Control resolution bandwidth in Hz. Can be set to 'AUTO'

single_sweep()

Perform a single sweep with synchronization.

property sweep_time

Control sweep time in s. Can be set to 'AUTO'.

property trace_mode

Control trace mode (Write: 'WRIT', Max Hold: 'MAXH', Min Hold: 'MINH', Average: 'AVER' or View: 'VIEW')

property video_bandwidth

Control video bandwidth in Hz. Can be set to 'AUTO'

class pymeasure.instruments.rohdeschwarz.fsseries.**FSL**(*adapter*, *name='Rohde&Schwarz FSL'*, ***kwargs*)

Bases: *FSSeries*

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.).

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]].

class pymeasure.instruments.rohdeschwarz.fsseries.**FSW**(*adapter*, *name='Rohde&Schwarz FSW'*, ***kwargs*)

Bases: *FSSeries*

Represents a Rohde&Schwarz FSW 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 activate_channel

Control the name of the active channel. Note: The channel needs to be open on the device!

Returns
active channel name

Return type
string

property active_channel

Control the name of the active channel. Note: The channel needs to be open on the device!

Returns
active channel name

Return type
string

auto_all()

Adjust all determinable settings automatically.

auto_freq()

Adjust center frequency automatically.

auto_level()

Adjust reference level automatically.

property available_channels

Measure open channel names and corresponding types

create_channel(*channel_type*, *channel_name*)

Create a new channel.

Parameters

- **channel_type** – Type of channel to be created.For example "PNOISE" or "SANALYZER"
- **channel_name** – Name of the channel to be added.

delete_channel(*channel_name*)

Deletes an active channel.

property nominal_level

Control the nominal level of the instrument

read_trace(*n_trace=1*)

Read trace data of the active trace.

Parameters
n_trace – The trace number (1-6). Default is 1.

Returns
2d numpy array of the trace data, [[frequency], [amplitude]].

rename_channel(*current_name*, *new_name*)

Rename current_name of a channel to a new_name.

Parameters

- **current_name** – Channel to be renamed
- **new_name** – New name of the channel

select_channel(*channel_name*)

Select an open channel

Parameters
channel_name – Channel to be selected.

property split_view

Control the viewmode of the device: True for split view or False for single channel view

7.51.3 R&S HMP4040 Power Supply

class pymeasure.instruments.rohdeschwarz.hmp.**HMP4040**(*adapter*, ***kwargs*)

Bases: *SCPIMixin*, *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

Control manual front panel ('LOC'), remote ('REM') or manual/remote control('MIX') control or locks the front panel control ('RWL').

property current

Control output current in A. Range depends on instrument type.

property current_step

Control 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

Get maximum current in A.

property max_voltage

Get maximum voltage in V.

property measured_current

Get current in A.

property measured_voltage

Get voltage in V.

property min_current

Get minimum current in A.

property min_voltage

Get minimum voltage in V.

property output_enabled

Control the output on or off or check the output status.

property repetitions

Control umber 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

Control the selected channel.

property selected_channel_active

Control the selected channel to active or inactive or check its status.

property sequence

Set 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

Get the SCPI version the instrument's command set complies with.

property voltage

Control output voltage in V. Increment 0.001 V.

property voltage_and_current

Control output voltage (V) and current (A).

property voltage_step

Control 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.

7.52 Santec

This section contains specific documentation on the Santec instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.52.1 Santec TSL-570 Tunable Laser

pymeasure.instruments.santec.**tsl570**

alias of <module 'pymeasure.instruments.santec.tsl570' from '/home/docs/checkouts/readthedocs.org/user_builds/pymeasure/chec

7.53 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*.

7.53.1 Siglent Technologies Base Class

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SPDBase**(*adapter*, *name='Siglent SPDxxxxX instrument Base Class'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

The base class for Siglent SPDxxxxX instruments.

Uses *SPDChannel* for measurement channels.

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

Get the error code and information of the instrument.

Type
string

property fw_version

Get 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

Get the system status register.

Type
SystemStatusCode

class pymeasure.instruments.siglenttechnologies.siglent_spdbase.**SPDSingleChannelBase**(*adapter*, *name='Siglent SPDxxxxX instrument Base Class'*, ***kwargs*)

Bases: *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: *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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

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

7.53.2 Siglent SPD1168X Power Supply

class pymeasure.instruments.siglenttechnologies.**SPD1168X**(*adapter*, *name='Siglent Technologies SPD1168X Power Supply'*, ***kwargs*)

Bases: *SPDSingleChannelBase*

Represent the Siglent SPD1168X Power Supply.

ch_1

Channel
SPDChannel

7.53.3 Siglent SPD1305X Power Supply

class pymeasure.instruments.siglenttechnologies.**SPD1305X**(*adapter*, *name='Siglent Technologies SPD1305X Power Supply'*, ***kwargs*)

Bases: *SPDSingleChannelBase*

Represent the Siglent SPD1305X Power Supply.

ch_1

Channel
SPDChannel

7.53.4 Siglent SDS1072CML Oscilloscope

class pymeasure.instruments.siglenttechnologies.**SDS1072CML**(*adapter*, *name='Siglent SDS1072CML Oscilloscope'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the SIGLENT SDS1072CML Oscilloscope

channel_1

Channel
VoltageChannel

channel_2

Channel
VoltageChannel

trigger

Channel
TriggerChannel

arm()
Change the acquisition mode from 'STOPPED' to 'SINGLE'. Returns True if the scope is ready and armed.

property internal_state
Get the scope's Internal state change register and clears it.

property is_ready

Get whether the scope is ready for the next acquisition (bool)

property status

Get the sampling status of the scope (Stop, Ready, Trig'd, Armed)

property template

Get a copy of the template that describes the various logical entities making up a complete waveform. In particular, the template describes in full detail the variables contained in the descriptor part of a waveform.

property time_division

Control the time division to the closest possible value, rounding downwards, in seconds.

wait(*time*)

Stop the scope from doing anything until it has completed the current acquisition (p.146)

Parameters
time – time in seconds to wait for

property waveform_setup

Control the amount of data in a waveform to be transmitted to the controller with a dict containing the keys: - sparsing: The interval between data points (0 and 1 are all points, 4 are 1 out of 4 points, etc…) - number: the maximal number of data points to return. 0 is all points - first: the index of the data point from which to begin data transfer (0 is first)

class pymeasure.instruments.siglenttechnologies.siglent_sds1072cml.**VoltageChannel**(*parent*, *id*)

Bases: *Channel*

Implementation of a SIGLENT SDS1072CML Oscilloscope channel

property coupling

Control the channel coupling mode.(DC or AC)

get_descriptor(*descriptor=None*)

Get the descriptor of data being sent when querying device for waveform Will decode a descriptor if provided one (Raw byte stream), otherwise it will query it :return: dict: A dictionary with the keys: - numDataPoints: the number of points in the waveform - verticalGain: the voltage increment per code value (in V) - verticalOffset: the voltage offset to add to the decoded voltage values - horizInterval: the time interval between points in s - horizOffset:the offset to add to the time steps - descriptorOffset: Length of the C1:WF ALL,#9000000346 message

get_waveform()

Get the waveforms displayed in the channel as a tuple with elements: - time: (1d array) the time in seconds since the trigger epoch for every voltage value in the waveforms - voltages: (1d array) the waveform in V

property vertical_division

Control the vertical sensitivity of a channel in V/divisions.

class pymeasure.instruments.siglenttechnologies.siglent_sds1072cml.**TriggerChannel**(*parent*, *id*)

Bases: *Channel*

Implementation of trigger control channel

get_trigger_config()

Get the current trigger configuration as a dict with keys: - "type": condition that will trigger the acquisition of waveforms [EDGE, slew,GLIT,intv,runt,drop] - "source": trigger source (str, {EX,EX/5,C1,C2}) - "hold_type": hold type (refer to page 131 of programing guide) - "hold_value1": hold value1 (refer to page 131 of programing guide) - "level": Level at which the trigger will be set (float) - "slope": (str,{POS,NEG,WINDOW}) Triggers on rising, falling or Window. - "mode": behavior of the trigger following a triggering event (str, {NORM, AUTO, SINGLE,STOP}) - "coupling": (str,{AC,DC}) Coupling to the trigger channel

and updates the internal configuration status

set_trigger_config(***kwargs*)

Set the current trigger configuration with keys: - "type": condition that will trigger the acquisition of waveforms [EDGE, slew,GLIT,intv,runt,drop] - "source": trigger source (str, {EX,EX/5,C1,C2}) - "hold_type": hold type (refer to page 131 of programing guide) - "hold_value1": hold value1 (refer to page 131 of programing guide) - "level": Level at which the trigger will be set (float) - "slope": (str,{POS,NEG,WINDOW}) Triggers on rising, falling or Window. - "mode": behavior of the trigger following a triggering event (str, {NORM, AUTO, SINGLE,STOP}) - "coupling": (str,{AC,DC}) Coupling to the trigger channel

Returns a flag indicating if all specified entries were correctly set on the oscilloscope and updates the interal trigger configuration

property trigger_coupling

Get the current trigger coupling as a dict with keys: - "source": trigger source whose coupling will be changed (str, {EX,EX/5,C1,C2}) - "coupling": (str,{AC,DC}) Coupling to the trigger channel

property trigger_level

Get the current trigger level as a dict with keys: - "source": trigger source whose level will be changed (str, {EX,EX/5,C1,C2}) - "level": Level at which the trigger will be set (float)

property trigger_mode

Get the current trigger mode as a dict with keys: - "mode": behavior of the trigger following a triggering event (str, {NORM, AUTO, SINGLE,STOP})

property trigger_setup

Get the current trigger setup as a dict with keys: - "type": condition that will trigger the acquisition of waveforms [EDGE, slew,GLIT,intv,runt,drop] - "source": trigger source (str, {EX,EX/5,C1,C2}) - "hold_type": hold type (refer to page 131 of programing guide) - "hold_value1": hold value1 (refer to page 131 of programing guide)

property trigger_slope

Get the current trigger slope as a dict with keys: - "source": trigger source whose level will be changed (str, {EX,EX/5,C1,C2}) - "slope": (str,{POS,NEG,WINDOW}) Triggers on rising, falling or Window.

7.54 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*.

7.54.1 DSP 7225 Lock-in Amplifier

class pymeasure.instruments.signalrecovery.**DSP7225**(*adapter*, *name='Signal Recovery DSP 7225'*, ***kwargs*)

Bases: DSPBase

Represents the Signal Recovery DSP 7225 lock-in amplifier.

Class inherits commands from the DSPBase parent class and utilizes dynamic properties for various properties.

```
lockin7225 = DSP7225("GPIB0::12::INSTR")
lockin7225.imode = "voltage mode"       # Set to measure voltages
lockin7225.reference = "internal"       # Use internal oscillator
lockin7225.fet = 1                      # Use FET pre-amp
lockin7225.shield = 0                   # Ground shields
lockin7225.coupling = 0                 # AC input coupling
lockin7225.time_constant = 0.10         # Filter time set to 100 ms
lockin7225.sensitivity = 2E-3           # Sensitivity set to 2 mV
lockin7225.frequency = 100              # Set oscillator frequency to 100 Hz
lockin7225.voltage = 1                  # Set oscillator amplitude to 1 V
lockin7225.gain = 20                    # Set AC gain to 20 dB
print(lockin7225.x)                     # Measure X channel voltage
lockin7225.shutdown()                    # Instrument shutdown
```

property adc1

Measure the voltage of the ADC1 input on the rear panel.

Returned value is a floating point number in volts.

property adc2

Measure the voltage of the ADC2 input on the rear panel.

Returned value is a floating point number in volts.

property auto_gain

Control lock-in amplifier for automatic AC gain.

auto_phase()

Adjusts the reference absolute phase to maximize the X channel output and minimize the Y channel output signals.

auto_sensitivity()

Adjusts the full-scale sensitivity so signal's magnitude lies between 30 - 90 % of full-scale.

buffer_to_float(*buffer_data*, *sensitivity=None*, *sensitivity2=None*, *raise_error=True*)

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

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clears the instrument status byte

property complete

Get the synchronization bit.

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 coupling

Control the input coupling mode.

Valid values are 0 for AC coupling mode or 1 for DC coupling mode.

property curve_buffer_bits

Control which data outputs are stored in the curve buffer.

Valid values are values are integers between 1 and 65,535 (or 2,097,151 in dual reference mode). (dynamic)

property curve_buffer_interval

Control the time interval between the collection of successive points in the curve buffer.

Valid values to the time interval are integers 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

Control the length of the curve buffer.

Valid values are integers 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

Measure the status of the curve buffer acquisition.

Command returns four values: **First value - Curve Acquisition Status:** Number with 5 possibilities: 0: no activity 1: acquisition via TD command running 2: acquisition by a TDC command running 5: acquisition via TD command halted 6: acquisition bia TDC command halted **Second value - Number of Sweeps Acquired**: Number of sweeps already acquired. **Third value - Status Byte:** Decimal representation of the status byte (the same response as the ST command **Fourth value - Number of Points Acquired:** Number of points acquired in the curve buffer.

property dac1

Control the voltage of the DAC1 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property dac2

Control the voltage of the DAC2 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property fet

Control the voltage preamplifier transistor type.

Valid values are 0 for bipolar or 1 for FET.

property frequency

Control the oscillator frequency.

Valid values are floating point numbers representing the frequency in Hz. (dynamic)

property gain

Control the AC gain of signal channel amplifier.

get_buffer(*quantity=None*, *convert_to_float=True*, *wait_for_buffer=True*)

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

Control the reference harmonic mode.

Valid values are integers. (dynamic)

property id

Measure the model number of the instrument.

Returned value is an integer.

property imode

Control the lock-in amplifier to detect a voltage or current signal.

Valid values are `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

Measure the log (base 10) of the ratio between the X channel and ADC1.

Returned value is a unitless floating point number equivalent to the mathematical expression log(X/ADC1).

property mag

Measure the magnitude of the signal.

Returned value is a floating point number in volts.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get the device options installed.

property phase

Measure the signal's absolute phase angle.

Returned value is a floating point number in degrees.

property ratio

Measure the ratio between the X channel and ADC1.

Returned value is a unitless floating point number equivalent to the mathematical expression X/ADC1.

read(***kwargs*)

Read the response and remove extra unicode character from instrument readings.

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).

property reference

Control the oscillator reference input mode.

Valid values are `internal`, `external rear` or `external front`.

property reference_phase

Control the reference absolute phase angle.

Valid values are floating point numbers between 0 - 360 degrees.

reset()

Resets the instrument.

property sensitivity

Control the signal's measurement sensitivity range.

When in voltage measurement mode, valid values are discrete values from 2 nV to 1 V. When in current measurement mode, valid values are discrete values from 2 fA to 1 µA (for normal current mode) or up to 10 nA (for low noise current mode).

setChannelAMode()

Sets lock-in amplifier to measure a voltage signal only from the A input connector.

setDifferentialMode(*lineFiltering=True*)

Sets lock-in amplifier to differential mode, measuring A-B.

set_buffer(*points*, *quantities=None*, *interval=0.01*)

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.

set_voltage_mode()

Sets lock-in amplifier to measure a voltage signal.

property shield

Control the input connector shield state.

Valid values are 0 to have shields grounded or 1 to have the shields floating (i.e., connected to ground via a 1 kOhm resistor).

shutdown()

Safely shutdown the lock-in amplifier.

Sets oscillator amplitude to 0 V and AC gain to 0 dB.

property slope

Control the low-pass filter roll-off.

Valid values are the integers 6, 12, 18, or 24, which represents the slope of the low-pass filter in dB/octave.

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 status

Get the status byte and Master Summary Status bit.

property time_constant

Control the filter time constant.

Valid values are a strict set of time constants from 10 us to 50,000 s. Returned values are floating point numbers in seconds.

property voltage

Control the oscillator amplitude.

Valid values are floating point numbers between 0 to 5 V.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*timeout=None*, *delay=0.1*)

Method that waits until the curve buffer is filled

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.

property x

Measure the output signal's X channel.

Returned value is a floating point number in volts.

property xy

Measure both the X and Y channels.

Returned values are floating point numbers in volts.

property y

Measure the output signal's Y channel.

Returned value is a floating point number in volts.

7.54.2 DSP 7265 Lock-in Amplifier

class pymeasure.instruments.signalrecovery.**DSP7265**(*adapter*, *name='Signal Recovery DSP 7265'*, ***kwargs*)

Bases: DSPBase

Represents the Signal Recovery DSP 7265 lock-in amplifier.

Class inherits commands from the DSPBase parent class and utilizes dynamic properties for various properties and includes additional functionality.

```
lockin7265 = DSP7265("GPIB0::12::INSTR")
lockin7265.imode = "voltage mode"       # Set to measure voltages
lockin7265.reference = "internal"       # Use internal oscillator
lockin7265.fet = 1                      # Use FET pre-amp
lockin7265.shield = 0                   # Ground shields
lockin7265.coupling = 0                 # AC input coupling
lockin7265.time_constant = 0.10         # Filter time set to 100 ms
lockin7265.sensitivity = 2E-3           # Sensitivity set to 2 mV
lockin7265.frequency = 100              # Set oscillator frequency to 100 Hz
lockin7265.voltage = 1                  # Set oscillator amplitude to 1 V
lockin7265.gain = 20                    # Set AC gain to 20 dB
print(lockin7265.x)                     # Measure X channel voltage
lockin7265.shutdown()                   # Instrument shutdown
```

property adc1

Measure the voltage of the ADC1 input on the rear panel.

Returned value is a floating point number in volts.

property adc2

Measure the voltage of the ADC2 input on the rear panel.

Returned value is a floating point number in volts.

property adc3

Measure the ADC3 input voltage.

property adc3_time

Control the ADC3 sample time in seconds.

property auto_gain

Control lock-in amplifier for automatic AC gain.

auto_phase()

Adjusts the reference absolute phase to maximize the X channel output and minimize the Y channel output signals.

auto_sensitivity()

Adjusts the full-scale sensitivity so signal's magnitude lies between 30 - 90 % of full-scale.

buffer_to_float(*buffer_data*, *sensitivity=None*, *sensitivity2=None*, *raise_error=True*)

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

`check_errors()`

Read all errors from the instrument and log them.

Returns
List of error entries.

`check_get_errors()`

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`check_set_errors()`

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

`clear()`

Clears the instrument status byte

`property complete`

Get the synchronization bit.

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 coupling`

Control the input coupling mode.

Valid values are 0 for AC coupling mode or 1 for DC coupling mode.

`property curve_buffer_bits`

Control which data outputs are stored in the curve buffer.

Valid values are values are integers between 1 and 65,535 (or 2,097,151 in dual reference mode). (dynamic)

`property curve_buffer_interval`

Control the time interval between the collection of successive points in the curve buffer.

Valid values to the time interval are integers 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

Control the length of the curve buffer.

Valid values are integers 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

Measure the status of the curve buffer acquisition.

Command returns four values: **First value - Curve Acquisition Status:** Number with 5 possibilities: 0: no activity 1: acquisition via TD command running 2: acquisition by a TDC command running 5: acquisition via TD command halted 6: acquisition bia TDC command halted **Second value - Number of Sweeps Acquired**: Number of sweeps already acquired. **Third value - Status Byte:** Decimal representation of the status byte (the same response as the ST command **Fourth value - Number of Points Acquired:** Number of points acquired in the curve buffer.

property dac1

Control the voltage of the DAC1 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property dac2

Control the voltage of the DAC2 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property dac3

Control the voltage of the DAC3 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property dac4

Control the voltage of the DAC4 output on the rear panel.

Valid values are floating point numbers between -12 to 12 V.

property fet

Control the voltage preamplifier transistor type.

Valid values are 0 for bipolar or 1 for FET.

property frequency

Control the oscillator frequency.

Valid values are floating point numbers representing the frequency in Hz. (dynamic)

property gain

Control the AC gain of signal channel amplifier.

get_buffer(*quantity=None*, *convert_to_float=True*, *wait_for_buffer=True*)

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

Control the reference harmonic mode.

Valid values are integers. (dynamic)

property id

Measure the model number of the instrument.

Returned value is an integer.

property imode

Control the lock-in amplifier to detect a voltage or current signal.

Valid values are `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

Measure the log (base 10) of the ratio between the X channel and ADC1.

Returned value is a unitless floating point number equivalent to the mathematical expression log(X/ADC1).

property mag

Measure the magnitude of the signal.

Returned value is a floating point number in volts.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get the device options installed.

property phase

Measure the signal's absolute phase angle.

Returned value is a floating point number in degrees.

property ratio

Measure the ratio between the X channel and ADC1.

Returned value is a unitless floating point number equivalent to the mathematical expression X/ADC1.

read(***kwargs*)

Read the response and remove extra unicode character from instrument readings.

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).

property reference

Control the oscillator reference input mode.

Valid values are `internal`, `external rear` or `external front`.

property reference_phase

Control the reference absolute phase angle.

Valid values are floating point numbers between 0 - 360 degrees.

reset()

Resets the instrument.

property sensitivity

Control the signal's measurement sensitivity range.

When in voltage measurement mode, valid values are discrete values from 2 nV to 1 V. When in current measurement mode, valid values are discrete values from 2 fA to 1 μA (for normal current mode) or up to 10 nA (for low noise current mode).

setChannelAMode()

Sets lock-in amplifier to measure a voltage signal only from the A input connector.

setDifferentialMode(*lineFiltering=True*)

Sets lock-in amplifier to differential mode, measuring A-B.

set_buffer(*points*, *quantities=None*, *interval=0.01*)

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.

set_voltage_mode()

Sets lock-in amplifier to measure a voltage signal.

property shield

Control the input connector shield state.

Valid values are 0 to have shields grounded or 1 to have the shields floating (i.e., connected to ground via a 1 kOhm resistor).

shutdown()

Safely shutdown the lock-in amplifier.

Sets oscillator amplitude to 0 V and AC gain to 0 dB.

property slope

Control the low-pass filter roll-off.

Valid values are the integers 6, 12, 18, or 24, which represents the slope of the low-pass filter in dB/octave.

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 status

Get the status byte and Master Summary Status bit.

property time_constant

Control the filter time constant.

Valid values are a strict set of time constants from 10 us to 50,000 s. Returned values are floating point numbers in seconds.

property voltage

Control the oscillator amplitude.

Valid values are floating point numbers between 0 to 5 V.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

wait_for_buffer(*timeout=None*, *delay=0.1*)

Method that waits until the curve buffer is filled

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.

property x

Measure the output signal's X channel.

Returned value is a floating point number in volts.

property xy

Measure both the X and Y channels.

Returned values are floating point numbers in volts.

property y

Measure the output signal's Y channel.

Returned value is a floating point number in volts.

7.55 Spellman HV

This section contains specific documentation on the Spellman HV instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.55.1 Spellman XRV High Voltage Power Supply

class pymeasure.instruments.spellmanhv.**SpellmanXRV**(*adapter*, *name='Spellman XRV HV Power Supply'*, *query_delay=0.15*, *baud_rate=9600*, ***kwargs*)

Bases: *Instrument*

A class representing the Spellman XRV series high voltage power supplies.

filament

Channel
Filament

unscaled

Channel
UnscaledData

property analog_monitor

Measure the analog monitor read backs.

Returns
dict

Dict keys
`voltage`, `current`, `filament`, `voltage_setpoint`, `current_setpoint`, `limit`, `preheat`, `anode_current`

(dynamic)

property baudrate

Set the baud rate (int, strictly in `9600`, `19200`, `38400`, `57600`, `115200`).

check_set_errors()

Check for errors after sending a command.

Raise
ValueError if response is not $

static checksum(*string_to_check*)

Calculate the checksum.

Parameters
string_to_check – string to calculate the checksum from

The checksum is computed as follows:

- Add all the bytes before <CSUM>, except <STX>, into a 16 bit (or larger) word. The bytes are added as unsigned integers.
- Take the two's complement.
- Truncate the result down to the eight least significant bits.
- Clear the most significant bit (bit 7) of the resultant byte, (bitwise AND with 0x7F).
- Set the next most significant bit (bit 6) of the resultant byte (bitwise OR with 0x40).

Using this method, the checksum is always a number between 0x40 and 0x7F. The checksum can never be confused with the <STX> or <ETX> control characters, since these have non overlapping ASCII values.

property configuration

Get the power supply configuration.

Returns
dict

Dict keys
reserved1, over_voltage_percentage, voltage_ramp_rate, current_ramp_rate, pre_warning_time, arc_count, reserved2, quench_time, max_kV, max_mA, watchdog_timer

property current_setpoint

Control the current setpoint in Amps (float).(dynamic)

property dsp

Get the DSP part number and version (list).

property errors

Get the power supply errors (enum).

property fpga

Get the FPGA part number and version (list).

property hv_on_timer

Get the HV On time in hours (float).

property output_enabled

Control the high voltage output (bool).

property power_limits

Control the power limits in Watts (list of int).

index = 0: power limit for large filament index = 1: power limit for small filament

Raise
ValueError if limit is out of range

`read`()
Read from the device and check for errors.

Raise
ConnectionError if response doesn't start with <STX> or checksum is incorrect. The checksum check is omitted for TCPIP connections.

`property scaling`
Get scaling factors and polarity.

Returns
dict

Dict keys
`voltage`, `current`, `polarity`

`voltage` is in V, `current` is in A, `polarity` 0: uni-polar, `polarity` 1: bipolar

`set_scaling`()
Set the scaling factors.

Used to set the scaling factor for *analog_monitor*, *current_setpoint*, *voltage*, *voltage_setpoint* and *system_voltages*.

`property status`
Get the power supply status (`StatusCode` enum).

`property system_voltages`
Measure the system voltages in Volts.

Returns
dict

Dict keys
`temperature`, `reserved`, `anode`, `cathode`, `ac_line_cathode`, `dc_rail_cathode`, `ac_line_anode`, `dc_rail_anode`, `lvps_pos`, `lvps_neg`

(dynamic)

`property temperature`
Measure the system temperature in °C (float).

`property voltage`
Measure the output voltage in Volts (float).(dynamic)

`property voltage_setpoint`
Control the voltage setpoint in Volts (float).(dynamic)

`wait_for`(*query_delay=0*)
Wait for some time.

Parameters
query_delay – override the global query_delay.

`write`(*command*)
Write to the instrument.

Adds <STX> (0x02) in front and checksum + <ETX> (0x03) at end of every command before sending it. The checksum is omitted for TCPIP connections.

class pymeasure.instruments.spellmanhv.spellmanXRV.**Filament**(*parent*, *id*)

Bases: *Channel*

A class representing the functions for the filament of the x-ray tube.

property enabled

Set the filament status (bool).

property large_size_enabled

Set the large filament state (bool)

property limit

Control the filament limit setpoint (int, strictly from 0 to 4095).

property preheat

Control the filament preheat setpoint (int, strictly from 0 to 4095).

class pymeasure.instruments.spellmanhv.spellmanXRV.**UnscaledData**(*parent*, *id*)

Bases: *Channel*

A class to handle the unscaled raw data of the Spellman XRV power supplies.

property analog_monitor

Measure the analog monitor read backs.

Returns
dict of int from `0` to `4095`

Dict keys
`voltage`, `current`, `filament`, `voltage_setpoint`, `current_setpoint`, `limit`, `preheat`, `anode_current`

`voltage_setpoint`, `current_setpoint`, `limit` and `preheat` are for local mode operation only.

`anode_current` is only valid for bipolar units.

property current_setpoint

Control the current setpoint (int, strictly from `0` to `4095`).

property lvps_monitor

Measure the –15 V low voltage power supply (int from `0` to `4095`)

property system_voltages

Measure the system voltages.

Returns
dict of int from `0` to `4095`

Dict keys
`temperature`, `reserved`, `anode`, `cathode`, `ac_line_cathode`, `dc_rail_cathode`, `ac_line_anode`, `dc_rail_anode`, `lvps_pos`, `lvps_neg`

property voltage

Measure the output voltage (int from `0` to `4095`).

property voltage_setpoint

Control the voltage setpoint (int, strictly from `0` to `4095`).

7.56 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*.

7.56.1 LDC500 Series Laser-diode Controllers

class pymeasure.instruments.srs.ldc500series.**LDC500Series**(*adapter*, *name='LDC500Series'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents an SRS LDC500Series laser diode controller and provides a high-level interface for interacting with the instrument.

ld

Channel
LDC500SeriesLD

pd

Channel
LDC500SeriesPD

tec

Channel
LDC500SeriesTEC

check_errors()

Read all errors from the instrument.

Returns
List of [last_execution_error, last_command_error]

check_set_errors()

Check for errors after having set a property, and raise an error if any are present.

property last_command_error

Get the last command error code. This also resets the execution error code to 0.

property last_execution_error

Get the last execution error code. This also resets the execution error code to 0.

property next_error

Get next error not implemented, raises: NotImplementedError

property options

Get options not implemented, raises NotImplementedError

class pymeasure.instruments.srs.ldc500series.**LDC500SeriesLD**(*parent*, *id*)

Bases: *Channel*

LDC500Series channel for control of the laser-diode (LD).

property current

Measure the laser diode current, in mA (float).

property current_limit

Control the laser diode current limit, in mA (float).

The ouput current is clamped to never exceed `current_limit` under any conditions. If `current_limit` is reduced below `current_setpoint`, `current_setpoint` is "dragged" down with `current_limit`.

property current_range

Control the laser diode current range (str "HIGH" or "LOW").

The currents corresponding to current range depend on the model:

	LDC500	LDC501	LDC502
HIGH	100mA	500mA	2000mA
LOW	50mA	250mA	1000mA

property current_setpoint

Control the laser diode current setpoint, in mA, when `mode == "CC"` (float).

property enabled

Control whether the laser diode current source is enabled (bool).

property interlock_closed

Get the status of the interlock, True if closed, False if open. Laser will only operate with a closed interlock.

property mode

Control the laser diode control mode, (str "CC" or "CP").

If the laser is on when `mode` is changed, the controller performs a *bumpless transfer* to switch control modes on-the-fly.

If `mode` is changed from "CC" to "CP", the present value of the photodiode current (or equivalently optical power) is measured and the CP setpoint is set to this measurement. If the measured photodiode current or power exceed the setpoint limit, an error is thrown.

If the `mode` is changed from "CP" to "CC", the present value of the laser current is measured and the CC setpoint set to this measurement. Note that, by hardware design, the measured laser current can never exceed the setpoint limit.

property modulation_bandwidth

Control the laser diode modulation bandwidth (str "HIGH" or "LOW").

The analog modulation input is DC coupled, with the –3 dB roll-off frequency dependent on `mode` and `modulation_bandwidth`:

	CC	CP
LOW	10kHz	100Hz
HIGH	1.2mhz	5kHz

property modulation_enabled

Control whether analog modulation of the laser diode is enabled (bool)

property voltage

Measure the laser diode voltage, in V (float).

property voltage_limit

Control the laser diode voltage limit, in V (float strictly in range 0.1 to 10). The current source turns off when the voltage limit is exceeded.

class pymeasure.instruments.srs.ldc500series.**LDC500SeriesPD**(*parent*, *id*)

Bases: *Channel*

LDC500Series channel for control of the photodiode (PD).

property bias

Control the photodiode bias (float strictly in range 0 to 5).

calibrate(*power*)

Set the photodiode responsivity, `responsivity`, via a real time power measurement.

Parmeters:
power (float): Real time power measurement, in mW.

property current

Measure the photodiode current, in uA (float).

property current_limit

Control the photodiode current limit, in uA, when `mode == "CP"` and `photodiode_units == "uA"` (float).

property current_setpoint

Control the photodiode current setpoint, in uA, when `mode == "CP"` and `photodiode_units == "uA"` (float).

property power

Measure the photodiode power, in mW (float).

property power_limit

Control the photodiode power limit, in mW, when `mode == "CP"` and `photodiode_units == "mW"` (float).

If `power_limit` is reduced below `power_setpoint`, `power_setpoint` is "dragged" down with `power_limit`.

property power_setpoint

Control the photodiode optical power setpoint, in mW, when `mode == "CP"` and `photodiode_units == "mW"` (float).

property responsivity

Control the photodiode responsivity, in uA/mW (float).

Changing `responsivity`, either directly or via `calibrate`, indirectly changes:

- `current_setpoint` and `current_limit` if `units == "mW"`
- `power_setpoint` and `power_limit` if `units == "uA"`

This is to maintain the relationships:

- `current_setpoint = power_setpoint x responsivity`
- `current_limit = power_limit x responsivity`

property units

Control the CP photodiode units (str "mW" or "uA").

class pymeasure.instruments.srs.ldc500series.**LDC500SeriesTEC**(*parent*, *id*)

Bases: *Channel*

LDC500Series channel for control of the thermo-electric-controller (TEC).

check_temperature_stability(*tolerance=0.1*, *period=10*, *points=64*)

Determine whether the temperature is stable at the temperature setpoint over a specified period.

Parameters

- **tolerance** – Maximum allowed deviation from temperature setpoint, in degrees Centigrade.
- **period** – Time period over which stability is checked, in seconds.

Returns

True if stable, False otherwise.

property current

Measure the TEC current, in A.

property current_limit

Control the TEC current limit, in A (float strictly in range -4.5 to 4.5).

If `current_limit` is reduced below `current_setpoint`, `current_setpoint` is *dragged* down with `current_limit`.

property current_setpoint

Control the TEC current setpoint, in A, when `mode == "CC"` (float).

property enabled

Control whether the TEC current source is enabled (bool).

property mode

Control the TEC control mode (str “CC” or “CT”).

If the TEC is on when `mode` is changed, the controller performs a *bumpless transfer* to swich control modes on-the-fly.

If `mode` is changed from “CT” to “CC”, the present value of the TEC current is measured, and the CC setpoint is set to this measurement.

If `mode` is changed from “CC” to “CT”, the present value of the temperature sensor is measured, and the CT setpoint is set to this measurement.

property resistance_high_limit

Control the TEC high resistance limit, in k (float).

property resistance_limits

Control the TEC resistance limits, in k (two-tuple of floats).

property resistance_low_limit

Control the TEC low resistance limit, in k (float).

property resistance_setpoint

Control the TEC resistance setpoint, in k (float).

property temperature

Measure the TEC temperature, in °C (float).

property temperature_high_limit

Control the TEC high temperature limit, in °C (float).

property temperature_limits

Control the TEC temperature limits, in °C (two-tuple of floats).

property temperature_low_limit

Control the TEC low temperature limit, in °C (float).

property temperature_setpoint

Control the TEC temperature setpoint, in °C (float).

property thermometer_raw

Measure the raw thermometer reading, k, V, or A depending on the sensor type (float).

property thermometer_type

Control the temperature sensor type (ThermometerType enum).

property voltage

Measure the TEC voltage, in V (float).

property voltage_limit

Control the TEC voltage limit, in V (float strictly in range -8.5 to 8.5).

wait_for_temperature_stable(*tolerance=0.1*, *period=10*, *should_stop=<function LDC500SeriesTEC.<lambda>>*, *timeout=60*)

Block the program, waiting for the temperature to stabilize at the temperature setpoint.

Parameters

- **tolerance** – Maximum allowed deviation from temperature setpoint, in degrees Centigrade.
- **period** – Time period over which stability is checked, in seconds.
- **should_stop** – Function that returns True to stop waiting.
- **timeout** – Maximum waiting time, in seconds.

Returns

True when stable, False if stopped by should_stop.

Raises

TimeoutError – If the temperature does not stabilize within the timeout period.

class pymeasure.instruments.srs.ldc500series.**ThermometerType**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Enumerator of thermometer types supported by the SRS LDC500Series. (NTC - negative temperature coefficient; RTD - Resistance temperature detector)

Members:

- **NTC10UA** : 10 µA excitation for NTC thermistor.
- **NTC100UA** : 100 µA excitation for NTC thermistor.
- **NTC1MA** : 1 mA excitation for NTC thermistor.

- **NTCAUTO** : Automatically detects optimal excitation for NTC.
- **RTD** : Typically Pt-100 or other PTC sensors
- **LM335** : LM335 and compatible temperature-to-voltage transducers.
- **AD590** : AD590 and compatible temperature-to-current transducers.

7.56.2 SR510 Lock-in Amplifier

class pymeasure.instruments.srs.**SR510**(*adapter*, *name='Stanford Research Systems SR510 Lock-in amplifier'*, ***kwargs*)

Bases: *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.

7.56.3 SR570 Lock-in Amplifier

class pymeasure.instruments.srs.**SR570**(*adapter*, *name='Stanford Research Systems SR570 Lock-in amplifier'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

7.56.4 SR830 Lock-in Amplifier

class pymeasure.instruments.srs.**SR830**(*adapter*, *name='Stanford Research Systems SR830 Lock-in amplifier'*, ***kwargs*)

Bases: *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.

buffer_measure(*count*, *stopRequest=None*, *delay=0.001*)

Start a fast measurement mode and transfers data from buffer to extract mean and std measurements

Return the mean and std from both channels

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.

clear()

Clear the instrument status byte.

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	Status
0	unused
1	backup error
2	RAM error
3	unused
4	ROM error
5	GPIB error
6	DSP error
7	DSP error

fill_buffer(*count: int*, *has_aborted=<function SR830.<lambda>>*, *delay=0.001*)

Fill two numpy arrays with the content of the instrument buffer

Eventually waiting until the specified number of recording is done

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*)

Acquires the 32 bit floating point data through binary transfer

get_scaling(*channel*)

Returns the offset percent and the expansion 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 id

Get the identification of the instrument.

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	Status
0	Input/Amplifier overload
1	Time constant filter overload
2	Output overload
3	Reference unlock
4	Detection frequency range switched
5	Time constant changed indirectly
6	Data storage triggered
7	unused

load_setup(*setup_number: int*)

Load a previously saved instrument configuration from the memory referred to by an integer

Parameters

setup_number – the integer referring to the memory (between 1 and 9 (included))

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)

pause_scan()

Pause the data recording

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’]

reset()

Reset the instrument.

property sample_frequency

Gets the sample frequency in Hz

save_setup(*setup_number: int*)

Save the current instrument configuration (all parameters) in a memory referred to by an integer

Parameters
setup_number – the integer referring to the memory (between 1 and 9 (included))

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 percent (-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)

start_scan()

Start the data recording into the buffer

property status

Get the status byte and Master Summary Status bit.

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.

7.56.5 SR860 Lock-in Amplifier

class pymeasure.instruments.srs.**SR860**(*adapter*, *name='Stanford Research Systems SR860 Lock-in amplifier'*, ***kwargs*)

Bases: *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

Access filter_synchronous attribute with filer_synchronous (sic) property.

Deprecated since version 0.16.0: Use filter_synchronous instead.

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 filter_synchronous

A string property that represents the synchronous filter. This property can be set. Allowed values are:[‘Off’, ‘On’]

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_config

A string property that represents the voltage input mode. This property can be set. Allowed values are:[‘A’, ‘A-B’]

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_grounding

A string property that represents the input shield grounding. This property can be set. Allowed values are:[‘Float’, ‘Ground’]

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_source_trigger

A string property that represents the external reference trigger mode. This property can be set. Allowed values are:['SIN', 'POS', 'NEG', 'POSTTL', 'NEGTTL']

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 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.

property sensitvity

Access sensitivity attribute with sensitvity (sic) property.

Deprecated since version 0.16.0: Use sensitivity instead.

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:

index	enumeration	parameter
0	X	X output
1	Y	Y output
2	R	R output
3	THeta	output
4	IN1	Aux In1
5	IN2	Aux In2
6	IN3	Aux In3
7	IN4	Aux In4
8	XNOise	Xnoise
9	YNOise	Ynoise
10	OUT1	Aux Out1
11	OUT2	Aux Out2
12	PHAse	Reference Phase
13	SAMp	Sine Out Amplitude
14	LEVel	DC Level
15	FInt	Int. Ref. Frequency
16	FExt	Ext. Ref. Frequency

Parameters

- **val1** – parameter enumeration/index
- **val2** – parameter enumeration/index
- **val3** – parameter enumeration/index (optional)

Defaults:

val1 = “X” val2 = “Y” val3 = None

snap_all()

snap X,Y,R,THETA parameters at once

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

7.57 T&C Power Conversion

This section contains specific documentation on the instruments from T&C Power Conversion that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.57.1 T&C Power Conversion AG Series Plasma Generator CXN

class pymeasure.instruments.tcpowerconversion.**CXN**(*adapter*, *name='T&C RF sputtering power supply'*, *address=0*, ***kwargs*)

Bases: *Instrument*

T&C Power Conversion AG Series Plasma Generator CXN (also rebranded by AJA International Inc as 0113 GTC or 0313 GTC)

Connection to the device is made through an RS232 serial connection. The communication settings are fixed in the device at 38400, stopbit one, parity none. The device uses a command response system where every receipt of a command is acknowledged by returning a '*'. A '?' is returned to indicates the command was not recognized by the device.

A command messages always consists of the following bytes (B): 1B - header (always 'C'), 1B - address (ignored), 2B - command id, 2B - parameter 1, 2B - parameter, 2B - checksum

A response message always consists of: 1B - header (always 'R'), 1B - address of the device, 2B - length of the data package, variable length data, 2B - checksum response messages are received after the acknowledge byte.

Parameters

- **adapter** – pyvisa resource name of the instrument or adapter instance
- **name** (*string*) – Name of the instrument.
- **kwargs** – Any valid key-word argument for Instrument

Note

In order to enable setting any parameters one has to request control and periodically (at least once per 2s) poll any value from the device. Failure to do so will mean loss of control and the device will reset certain parameters (setpoint, disable RF, ...). If no value should be polled but control should remain active one can also use the ping method.

preset_1

Channel
PresetChannel

preset_2

Channel
PresetChannel

preset_3

Channel
PresetChannel

preset_4

Channel
PresetChannel

preset_5

Channel
PresetChannel

preset_6

Channel
PresetChannel

preset_7

Channel
PresetChannel

preset_8

Channel
PresetChannel

preset_9

Channel
PresetChannel

class Status(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `IntFlag`

IntFlag type used to represent the CXN status.

The used bits correspond to: bit 14: Analog interface enabled, bit 11: Interlock open, bit 10: Over temperature, bit 9: Reverse power limit, bit 8: Forward power limit, bit 6: MCG mode active, bit 5: load power leveling active, bit 4, External RF source active, bit 0: RF power on.

property dc_voltage

Get the DC voltage in volts.

property firmware_version

Get the UI-processor and RF-processor firmware version numbers.

property frequency

Get operating frequency in Hz.

property id

Get the device identification string.

property load_capacity

Control the percentage of full-scale value of the load capacity. It can be set only when manual_mode is True.

property manual_mode

Control the manual tuner mode.

property operation_mode

Control the operation mode.

ping()

Send a ping to the instrument.

property power

Get power readings for forward/reverse/load power in watts.

property power_limit

Get maximum power of the power supply.

property preset_slot

Control which preset slot will be used for auto-tune mode. Valid values are 0 to 9. 0 means no preset will be used

property pulse_params

Get pulse on/off time of the pulse waveform.

property ramp_rate

Control the ramp rate in watts/second.

property ramp_start_power

Control the ramp starting power in watts.

read()

Reads a response message from the instrument.

This method determines the length of the message from the automatically by reading the message header and also checks for a correct checksum.

Returns
the data fields

Return type
bytes

Raises
ValueError – if a checksum error is detected

release_control()

Release instrument control.

This will reset certain properties to safe defaults and disable the RF output.

request_control()

Request control of the instrument.

This is required to be able to set any properties.

property reverse_power_limit

Get maximum reverse power.

property rf_enabled

Control the RF output.

property serial

Get the serial number of the instrument.

property setpoint

Control the setpoint power level in watts.

property status

Get status field. The return value is represented by the IntFlag type Status.

property temperature

Get heat sink temperature in deg Celsius.

property tune_capacity

Control the percentage of full-scale value of the tune capacity. It can be set only when manual_mode is True.

property tuner

Get type of the used tuner.

values(*command*, *cast=<class 'int'>*, *separator=','*, *preprocess_reply=None*, ***kwargs*)

Write a command to the instrument and return a list of formatted values from the result.

This is derived from CommonBase.values and adapted here for use with bytes communication messages (no str conversion and strip). It is implemented as a general method to allow using it equally in PresetChannel and CXN. See Github issue #784 for details.

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.

Returns

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

write(*command*)

Writes a command to the instrument and includes needed required header and address.

Parameters

command (*str*) – command to be sent to the instrument

class pymeasure.instruments.tcpowerconversion.tccxn.**PresetChannel**(*parent*, *id*)

Bases: *Channel*

property load_capacity

Control the percentage of full-scale value of the load capacity preset.

property tune_capacity

Control the percentage of full-scale value of the tune capacity preset.

values(*command*, *cast=<class 'int'>*, *separator=','*, *preprocess_reply=None*, ***kwargs*)

Write a command to the instrument and return a list of formatted values from the result.

This is derived from CommonBase.values and adapted here for use with bytes communication messages (no str conversion and strip). It is implemented as a general method to allow using it equally in PresetChannel and CXN. See Github issue #784 for details.

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.

Returns

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

7.58 TDK Lambda

This section contains specific documentation on the TDK Lambda instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.58.1 TDK Lambda Genesys 40-38 DC power supply

class pymeasure.instruments.tdk.tdk_gen40_38.**TDK_Gen40_38**(*adapter*, *name='TDK Lambda Gen40-38'*, *address=6*, ***kwargs*)

Bases: TDK_Lambda_Base

Represents the TDK Lambda Genesys 40-38 DC power supply. Class inherits commands from the TDK_Lambda_Base parent class and utilizes dynamic properties adjust valid values on various properties.

```
psu = TDK_Gen40_38("COM3", 6)       # COM port and daisy-chain address
psu.remote = "REM"                  # PSU in remote mode
psu.output_enabled = True           # Turn on output
psu.ramp_to_current(2.0)            # Ramp to 2.0 A of current
print(psu.current)                  # Measure actual PSU current
print(psu.voltage)                  # Measure actual PSU voltage
psu.shutdown()                      # Run shutdown command
```

The initialization of a TDK instrument requires the current address of the TDK power supply. The default address for the TDK Lambda is 6.

Parameters

- **adapter** – VISAAdapter instance
- **name** – Instrument name. Default is "TDK Lambda Gen40-38"
- **address** – Serial port daisy chain number. Default is 6.

property address

Set the address of the power supply.

Valid values are integers between 0 - 30 (inclusive).

property auto_restart_enabled

Control the auto restart mode, which restores the power supply to the last output voltage and current settings with output enabled on startup.

Valid values are `True` to restore output settings with output enabled on startup and `False` to disable restoration of settings and output disabled on startup.

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Only use this command for setting commands, i.e. non-querying commands.

Any non-querying commands (i.e., a command that does NOT have the "?" symbol in it like the instrument command "PV 10") will automatically return an "OK" reply for valid command or an error code. This is done to confirm that the instrument has received the command. Any querying commands (i.e., a command that does have the "?" symbol in it like the instrument command "PV?") will return the requested value, not the confirmation.

clear()

Clear FEVE and SEVE registers to zero.

property complete

Get the synchronization bit.

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.

Returns a float with five digits of precision.

property current_setpoint

Control the programmed (set) output current.(dynamic)

property display

Get the displayed voltage and current.

Returns a list of floating point numbers in the order of [measured voltage, programmed voltage, measured current, programmed current, over voltage set point, under voltage set point].

property foldback_delay

Control the fold back delay.

Adds an additional delay to the standard fold back delay (250 ms) by multiplying the set value by 0.1. Valid values are integers between 0 to 255.

property foldback_enabled

Control the fold back protection of the power supply.

Valid values are `True` to arm the fold back protection and `False` to cancel the fold back protection.

foldback_reset()

Reset the fold back delay to 0 s, restoring the standard 250 ms delay.

Property is UNTESTED.

property id

Get the identity of the instrument.

Returns a list of instrument manufacturer and model in the format: `["LAMBDA", "GENX-Y"]`

property last_test_date

Get the date of the last test, possibly calibration date.

Returns a string in the format: yyyy/mm/dd.

property master_slave_setting

Get the master and slave settings.

Possible master return values are 1, 2, 3, and 4. The slave value is 0.

Property is UNTESTED.

property mode

Measure the output mode of the power supply.

When power supply is on, the returned value will be either `'CV'` for control voltage or `'CC'` for or control current. If the power supply is off, the returned value will be `'OFF'`.

property multidrop_capability

Get whether the multi-drop option is available on the power supply.

If return value is `False`, the option is not available, if `True` it is available.

Property is UNTESTED.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get the device options installed.

property output_enabled

Control the output of the power supply.

Valid values are `True` to turn output on and `False` to turn output off, shutting down any voltage or current.

property over_voltage

Control the over voltage protection. (dynamic)

property pass_filter

Control the low pass filter frequency of the A to D converter for voltage and current measurement.

Valid frequency values are 18, 23, or 46 Hz. Default value is 18 Hz.

ramp_to_current(*target_current*, *steps=20*, *pause=0.2*)

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** – Target current in amps
- **steps** – Integer number of steps
- **pause** – Pause duration in seconds to wait between steps

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).

recall()

Recall last saved instrument settings.

property remote

Control the current remote operation of the power supply.

Valid values are `'LOC'` for local mode, `'REM'` for remote mode, and `'LLO'` for local lockout mode.

property repeat

Measure the last command again.

Returns output of the last command.

reset()

Reset the instrument to default values.

save()

Save current instrument settings.

property serial

Get the serial number of the instrument.

Returns the serial number of of the instrument as an ASCII string.

set_max_over_voltage()

Set the over voltage protection to the maximum level for the power supply.

shutdown()

Safety shutdown the power supply.

Ramps the power supply down to zero current using the `self.ramp_to_current(0.0)` method and turns the output off.

property status

Get the power supply status.

Returns a list in the order of [actual voltage (MV), the programmed voltage (PV), the actual current (MC), the programmed current (PC), the status register (SR), and the fault register (FR)].

property under_voltage

Control the under voltage limit.

Property is UNTESTED. (dynamic)

property version

Get the software version on instrument.

Returns the software version as an ASCII string.

property voltage

Measure the actual output voltage.

property voltage_setpoint

Control the programmed (set) output voltage.(dynamic)

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.58.2 TDK Lambda Genesys 80-65 DC power supply

class pymeasure.instruments.tdk.tdk_gen80_65.**TDK_Gen80_65**(*adapter*, *name='TDK Lambda Gen80-65'*, *address=6*, ***kwargs*)

Bases: TDK_Lambda_Base

Represents the TDK Lambda Genesys 80-65 DC power supply. Class inherits commands from the TDK_Lambda_Base parent class and utilizes dynamic properties adjust valid values on various properties.

```
psu = TDK_Gen80_65("COM3", 6)        # COM port and daisy-chain address
psu.remote = "REM"                   # PSU in remote mode
psu.output_enabled = True            # Turn on output
psu.ramp_to_current(2.0)             # Ramp to 2.0 A of current
print(psu.current)                   # Measure actual PSU current
print(psu.voltage)                   # Measure actual PSU voltage
psu.shutdown()                       # Run shutdown command
```

The initialization of a TDK instrument requires the current address of the TDK power supply. The default address for the TDK Lambda is 6.

Parameters

- **adapter** – VISAAdapter instance
- **name** – Instrument name. Default is "TDK Lambda Gen80-65"
- **address** – Serial port daisy chain number. Default is 6.

property address

Set the address of the power supply.

Valid values are integers between 0 - 30 (inclusive).

property auto_restart_enabled

Control the auto restart mode, which restores the power supply to the last output voltage and current settings with output enabled on startup.

Valid values are `True` to restore output settings with output enabled on startup and `False` to disable restoration of settings and output disabled on startup.

check_errors()

Read all errors from the instrument and log them.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Only use this command for setting commands, i.e. non-querying commands.

Any non-querying commands (i.e., a command that does NOT have the "?" symbol in it like the instrument command "PV 10") will automatically return an "OK" reply for valid command or an error code. This is done to confirm that the instrument has received the command. Any querying commands (i.e., a command that does have the "?" symbol in it like the instrument command "PV?") will return the requested value, not the confirmation.

clear()

Clear FEVE and SEVE registers to zero.

property complete

Get the synchronization bit.

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.

Returns a float with five digits of precision.

property current_setpoint

Control the programmed (set) output current.(dynamic)

property display

Get the displayed voltage and current.

Returns a list of floating point numbers in the order of [measured voltage, programmed voltage, measured current, programmed current, over voltage set point, under voltage set point].

property foldback_delay

Control the fold back delay.

Adds an additional delay to the standard fold back delay (250 ms) by multiplying the set value by 0.1. Valid values are integers between 0 to 255.

property foldback_enabled

Control the fold back protection of the power supply.

Valid values are `True` to arm the fold back protection and `False` to cancel the fold back protection.

foldback_reset()

Reset the fold back delay to 0 s, restoring the standard 250 ms delay.

Property is UNTESTED.

property id

Get the identity of the instrument.

Returns a list of instrument manufacturer and model in the format: `["LAMBDA", "GENX-Y"]`

property last_test_date

Get the date of the last test, possibly calibration date.

Returns a string in the format: yyyy/mm/dd.

property master_slave_setting

Get the master and slave settings.

Possible master return values are 1, 2, 3, and 4. The slave value is 0.

Property is UNTESTED.

property mode

Measure the output mode of the power supply.

When power supply is on, the returned value will be either `'CV'` for control voltage or `'CC'` for or control current. If the power supply is off, the returned value will be `'OFF'`.

property multidrop_capability

Get whether the multi-drop option is available on the power supply.

If return value is `False`, the option is not available, if `True` it is available.

Property is UNTESTED.

property next_error

Get the next error of the instrument (tuple of code and message).

property options

Get the device options installed.

property output_enabled

Control the output of the power supply.

Valid values are `True` to turn output on and `False` to turn output off, shutting down any voltage or current.

property over_voltage

Control the over voltage protection. (dynamic)

property pass_filter

Control the low pass filter frequency of the A to D converter for voltage and current measurement.

Valid frequency values are 18, 23, or 46 Hz. Default value is 18 Hz.

ramp_to_current(*target_current*, *steps=20*, *pause=0.2*)

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** – Target current in amps
- **steps** – Integer number of steps
- **pause** – Pause duration in seconds to wait between steps

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).

recall()

Recall last saved instrument settings.

property remote

Control the current remote operation of the power supply.

Valid values are `'LOC'` for local mode, `'REM'` for remote mode, and `'LLO'` for local lockout mode.

property repeat

Measure the last command again.

Returns output of the last command.

reset()

Reset the instrument to default values.

save()

Save current instrument settings.

property serial

Get the serial number of the instrument.

Returns the serial number of of the instrument as an ASCII string.

set_max_over_voltage()

Set the over voltage protection to the maximum level for the power supply.

shutdown()

Safety shutdown the power supply.

Ramps the power supply down to zero current using the `self.ramp_to_current(0.0)` method and turns the output off.

property status

Get the power supply status.

Returns a list in the order of [actual voltage (MV), the programmed voltage (PV), the actual current (MC), the programmed current (PC), the status register (SR), and the fault register (FR)].

property under_voltage

Control the under voltage limit.

Property is UNTESTED. (dynamic)

property version

Get the software version on instrument.

Returns the software version as an ASCII string.

property voltage

Measure the actual output voltage.

property voltage_setpoint

Control the programmed (set) output voltage.(dynamic)

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.59 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*.

7.59.1 TDS2000 Oscilloscope

class pymeasure.instruments.tektronix.**TDS2000**(*adapter*, *name='Tektronix TDS 2000 Oscilloscope'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Tektronix TDS 2000 Oscilloscope and provides a high-level for interacting with the instrument

7.59.2 AFG3152C Arbitrary function generator

class pymeasure.instruments.tektronix.**AFG3152C**(*adapter*, *name='Tektronix AFG3152C arbitrary function generator'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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
```

class pymeasure.instruments.tektronix.afg3152c.**AFG3152CChannel**(*parent*, *id*)

Bases: *Channel*

property amp_dbm

Control the output amplitude in dBm. (float)

property amp_vpp

Control the output amplitude in Vpp. (float)

property amp_vrms

Control the output amplitude in Vrms. (float)

property duty

Control the duty cycle of pulse. (float))

property frequency

Control the frequency. (float)

property impedance

Control the output impedance of the channel. Be careful with this.

insert_id(*command*)

Prepend the channel id for most writes.

property offset

Control the amplitude offset. It is always in Volt. (float)

property shape

Control the shape of the output. (str)

property unit

Control the amplitude unit. (str)

waveform(*shape='SIN'*, *frequency=1000000.0*, *units='VPP'*, *amplitude=1*, *offset=0*)

General setting method for a complete wavefunction

7.60 Teledyne

This section contains specific documentation on the Teledyne instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

If the instrument you are looking for is not here, also check *LeCroy* for older instruments.

7.60.1 Teledyne T3AFG Arbitrary Waveform Generator

class pymeasure.instruments.teledyne.**TeledyneT3AFG**(*adapter*, *name='Teledyne T3AFG'*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents the Teledyne T3AFG series of arbitrary waveform generator interface for interacting with the instrument.

Initially targeting T3AFG80, some features may not be available on lower end models and features from higher end models are not included here yet.

Future improvements (help welcomed):

- Add other OUTPut related controls like Load and Polarity
- Add other Basic Waveform related controls like Period
- Add frequency ranges per model
- Add channel coupling control

ch_1

Channel
SignalChannel

ch_2

Channel
SignalChannel

check_errors()

Read all errors from the instrument.

Returns
List of error entries.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get 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()

Reset the instrument.

shutdown()

Brings the instrument to a safe and stable state

property status

Get the status byte and Master Summary Status bit.

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.teledyne.teledyneT3AFG.**SignalChannel**(*parent*, *id*)

Bases: *Channel*

property amplitude

Control the amplitude of waveform to be output in volts peak-to-peak. Has no effect when WVTP is NOISE or DC. Max amplitude depends on offset, frequency, and load. Amplitude is also limited by the channel max output amplitude.(dynamic)

property frequency

Control the frequency of waveform to be output in Hertz. Has no effect when WVTP is NOISE or DC.(dynamic)

property max_output_amplitude

Control the maximum output amplitude of the channel in volts peak to peak.(dynamic)

property offset

Control the offset of waveform to be output in volts. Has no effect when WVTP is NOISE. Max offset depends on amplitude, frequency, and load. Offset is also limited by the channel max output amplitude.(dynamic)

property output_enabled

Control whether the channel output is enabled (boolean).

property wavetype

Control the type of waveform to be output. Options are: {'SINE', 'SQUARE', 'RAMP', 'PULSE', 'NOISE', 'ARB', 'DC', 'PRBS', 'IQ'}

There are shared base classes for Teledyne oscilloscopes. These base classes already work directly for some devices, the following are confirmed:

- *TeledyneMAUI*
  - Teledyne LeCroy HDO6xxx series (e.g. HDO6054B)

If your device is not listed, the base class might already work well enough anyway. If adding a new device, these base classes should limit the amount of new code necessary.

7.60.2 Teledyne Oscilloscope base classes

Teledyne Oscilloscope

class pymeasure.instruments.teledyne.**TeledyneOscilloscope**(*adapter*, *name='Teledyne Oscilloscope'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

A base abstract class for any Teledyne Lecroy oscilloscope.

All Teledyne oscilloscopes have a very similar interface, hence this base class to combine them. Note that specific models will likely have conflicts in their interface.

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.

ch_1

Channel
TeledyneOscilloscopeChannel

ch_2

Channel
TeledyneOscilloscopeChannel

ch_3

Channel
TeledyneOscilloscopeChannel

ch_4

Channel
TeledyneOscilloscopeChannel

autoscale()

Autoscale displayed channels.

property bwlimit

Set the internal low-pass filter for all channels.(dynamic)

center_trigger()

Set 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.

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.

Parameters

- **source** – measurement source. It can be "C1", "C2", "C3", "C4", "MATH".
- **requested_points** – number of points to acquire. If None the number of points requested in the previous call will be assumed, i.e. the value of the number of points stored in the oscilloscope memory. If 0 the maximum number of points will be returned.
- **sparsing** – interval between data points. For example if sparsing = 4, only one point every 4 points is read. If 0 or None the sparsing of the previous call is assumed, i.e. the value of the sparsing stored in the oscilloscope memory.

Returns

data_ndarray, time_ndarray, waveform_preamble_dict: see waveform_preamble property for dict format.

property intensity

Set the intensity level of the grid or the trace in percent

measure_parameter(*parameter*, *channel*)

Same as the measure_parameter method in the Channel subclass

property memory_size

Control the maximum depth of memory (float or string). Assign for example 500, 100e6, "100K", "25MA".

The reply will always be a float.

run()

Starts repetitive acquisitions.

This is the same as pressing the Run key on the front panel.

single()

Causes the instrument to acquire a single trigger of data.

This is the same as pressing the Single key on the front panel.

stop()

Stops the acquisition. This is the same as pressing the Stop key on the front panel.

property timebase

Get 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)

property timebase_offset

Control the time interval in seconds between the trigger event and the reference position (at center of screen by default).

property timebase_scale

Control the horizontal scale (units per division) in seconds for the main window (float).

timebase_setup(*scale=None*, *offset=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.

property trigger

Get 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

Control the trigger sweep mode (string).

<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

Control the condition that will trigger the acquisition of waveforms (string).

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.
   - {ti} for drop trigger.
   - {ps, pl, p2, p1} for glit/runt trigger.
   - {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.

Parameters

- **mode** – trigger sweep mode [auto, normal, single, stop]
- **source** – trigger source [c1, c2, c3, c4, line]
- **trigger_type** – condition that will trigger the acquisition of waveforms [edge,slew,glit,intv,runt,drop]
- **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 waveform_first_point

Control the address of the first data point to be sent (int). 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

Control the number of waveform points to be transferred with the digitize method (int). 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:

- "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".
- "grid_number": number of horizontal grids (it is a read-only property)
- "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

Control the interval between data points (integer). For example:

SP = 0 sends all data points. SP = 4 sends 1 point every 4 data points.

write(*command*, ***kwargs*)

Write 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

Teledyne Channel

class pymeasure.instruments.teledyne.teledyne_oscilloscope.**TeledyneOscilloscopeChannel**(*parent*, *id*)

Bases: *Channel*

A base abstract class for channel on a *TeledyneOscilloscope* device.

property bwlimit

Control the internal low-pass filter for this channel.

The current bandwidths can only be read back for all channels at once! (dynamic)

property coupling

Control the coupling with a string parameter ("ac 1M", "dc 1M", "ground").

property current_configuration

Get channel configuration as a dict containing the following keys:

- "channel": channel number (int)
- "attenuation": probe attenuation (float)

- "bandwidth_limit": bandwidth limiting enabled (bool)
- "coupling": "ac 1M", "dc 1M", "ground" coupling (str)
- "offset": vertical offset (float)
- "skew_factor": channel-tochannel skew factor (float)
- "display": currently displayed (bool)
- "unit": "A" or "V" units (str)
- "volts_div": vertical divisions (float)
- "inverted": inverted (bool)
- "trigger_coupling": trigger coupling can be "dc" "ac" "highpass" "lowpass" (str)
- "trigger_level": trigger level (float)
- "trigger_level2": trigger lower level for SLEW or RUNT trigger (float)
- "trigger_slope": trigger slope can be "negative" "positive" "window" (str)

property display

Control the display enabled state. (strict bool)

property display_parameter

Set the waveform processing of this channel with the specified algorithm and the result is displayed on the front panel.

The command accepts the following parameters:

Parameter	Description
PKPK	vertical peak-to-peak
MAX	maximum vertical value
MIN	minimum vertical value
AMPL	vertical amplitude
TOP	waveform top value
BASE	waveform base value
CMEAN	average value in the first cycle
MEAN	average value
RMS	RMS value
CRMS	RMS value in the first cycle
OVSN	overshoot of a falling edge
FPRE	preshoot of a falling edge
OVSP	overshoot of a rising edge
RPRE	preshoot of a rising edge
PER	period
FREQ	frequency
PWID	positive pulse width
NWID	negative pulse width
RISE	rise-time
FALL	fall-time
WID	Burst width
DUTY	positive duty cycle
NDUTY	negative duty cycle
ALL	All measurement

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.

measure_parameter(*parameter: str*)

Process a waveform with the selected algorithm and returns the specified measurement.

Parameters
parameter – same as the display_parameter property

property offset

Control the center of the screen in Volts by a a float parameter. The range of legal values varies depending on range and scale. If the specified value is outside of the legal range, the offset value is automatically set to the nearest legal value.

property probe_attenuation

Control the probe attenuation. The probe attenuation may be from 0.1 to 10000.

property scale

Control the vertical scale (units per division) in Volts.

setup(***kwargs*)

Setup channel. Unspecified settings are not modified.

Modifying values such as probe attenuation will modify offset, range, etc. Refer to oscilloscope documentation and make multiple consecutive calls to setup() if needed. See property descriptions for more information.

Parameters

- **bwlimit**
- **coupling**
- **display**
- **invert**
- **offset**
- **skew_factor**
- **probe_attenuation**
- **scale**
- **unit**
- **trigger_coupling**
- **trigger_level**
- **trigger_level2**
- **trigger_slope**

property trigger_coupling

Control the input coupling for the selected trigger sources (string).

- ac: AC coupling block DC component in the trigger path, removing dc offset voltage from the trigger waveform. Use AC coupling to get a stable edge trigger when your waveform has a large dc offset.
- dc: DC coupling allows dc and ac signals into the trigger path.

- lowpass: HFREJ coupling places a lowpass filter in the trigger path.
- highpass: LFREJ coupling places a highpass filter in the trigger path.

property trigger_level

Control the trigger level voltage for the active trigger source (float).

When there are two trigger levels to set, this command is used to set the higher trigger level voltage for the specified source. `trigger_level2` is used to set the lower trigger level voltage.

When setting the trigger level it must be divided by the probe attenuation. This is not documented in the datasheet and it is probably a bug of the scope firmware. An out-of-range value will be adjusted to the closest legal value.

property trigger_slope

Control the trigger slope of the specified trigger source (string).

<trig_slope>:={NEG,POS,WINDOW} for edge trigger <trig_slope>:={NEG,POS} for other trigger

parameter	trigger slope
negative	Negative slope for edge trigger or other trigger
positive	Positive slope for edge trigger or other trigger
window	Window slope for edge trigger

(dynamic)

Teledyne MAUI Oscilloscope

class pymeasure.instruments.teledyne.**TeledyneMAUI**(*adapter*, *name='Teledyne Oscilloscope'*, ***kwargs*)

Bases: *TeledyneOscilloscope*

A base class for the MAUI-type of Teledyne oscilloscopes.

This base class works out of the box. Some properties, especially the number of channels, might have to be adjusted to the actual device.

The manual detailing the API is "MAUI Oscilloscopes Remote Control and Automation Manual" (link).

This class of Teledyne oscilloscopes also support direct VBS commands. See *vbs_ask()* and *vbs_write()*.

ch_1

Channel
TeledyneMAUIChannel

ch_2

Channel
TeledyneMAUIChannel

ch_3

Channel
TeledyneMAUIChannel

ch_4

Channel
TeledyneMAUIChannel

download_image(***kwargs*)

Get a BMP image of oscilloscope screen in bytearray of specified file format.

The hardcopy destination is set to "REMOTE" by default.

Parameters

****kwargs** – Keyword arguments for *hardcopy_setup()*

force_trigger()

Make one acquisition if in active trigger mode.

No action is taken if the device is in 'Stop trigger mode'.

hardcopy_setup(***kwargs*)

Specify hardcopy settings.

Connect a printer or define how to save to file. Set any or all of the following parameters.

Parameters

- **device** – {BMP, JPEG, PNG, TIFF}
- **format** – {PORTRAIT, LANDSCAPE}
- **background** – {Std, Print, BW}
- **destination** – {PRINTER, CLIPBOARD, EMAIL, FILE, REMOTE}
- **area** – {GRIDAREAONLY, DSOWINDOW, FULLSCREEN}
- **directory** – Any legal DOS path, for FILE mode only
- **filename** – Filename string, no extension, for FILE mode only
- **printername** – Valid printer name, for PRINTER mode only
- **portname** – {GPIB, NET}

property hardcopy_setup_current

Get current hardcopy config.

property trigger

Get 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
- "slope": trigger slope of the specified trigger source

vbs_ask(*name: str*) → str

Return the value of a VBS variable.

Only the target needs to be specified, a query is formatted by this method. Note: the target name is not escaped!

See *vbs_write()* for more info.

A very basic example of usage:

```
instrument.vbs_ask("app.Display.GridMode")
```

vbs_write(*message: str*)

Write a VBS command directly to the device.

This class of oscilloscopes also allows the direct usage of Visual Basic Scripting (VBScript). With this method a literal VBS command is sent. You can use the 'MAUI Browser' on the oscilloscope to list all available variables.

A very basic example of usage:

```
instrument.vbs_write("app.Display.GridMode = Dual")
```

Teledyne MAUI Channel

class pymeasure.instruments.teledyne.teledyneMAUI.**TeledyneMAUIChannel**(*parent*, *id*)

Bases: *TeledyneOscilloscopeChannel*

Base class for channels on a *TeledyneMAUI* device.

autoscale()

Perform auto-setup command for channel.

property current_configuration

Get channel configuration as a dict containing the following keys:

- "channel": channel number (int)
- "attenuation": probe attenuation (float)
- "bandwidth_limit": bandwidth limiting, parsed for this channel (str)
- "coupling": "ac 1M", "dc 1M", "ground" coupling (str)
- "offset": vertical offset (float)
- "display": currently displayed (bool)
- "volts_div": vertical divisions (float)
- "trigger_coupling": trigger coupling can be "dc" "ac" "highpass" "lowpass" (str)
- "trigger_level": trigger level (float)
- "trigger_slope": trigger slope can be "negative" "positive" "window" (str)

setup(***kwargs*)

Setup channel. Unspecified settings are not modified.

Modifying values such as probe attenuation will modify offset, range, etc. Refer to oscilloscope documentation and make multiple consecutive calls to setup() if needed. See property descriptions for more information.

Parameters

- **bwlimit**
- **coupling**
- **display**
- **offset**
- **probe_attenuation**
- **scale**
- **trigger_coupling**
- **trigger_level**
- **trigger_slope**

7.61 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*.

7.61.1 Temptronic Base Class

class pymeasure.instruments.temptronic.**ATSBase**(*adapter*, *name='ATSBase'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

The base class for Temptronic ATSXXX instruments.

property air_temperature

Get air temperature in 0.1 °C increments.

Type
float

at_temperature()

Returns
True if at temperature.

property auxiliary_condition_code

Get 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

Refer to chapter 4 in the manual

clear()

Clear device-specific errors.

See *error_code* for further information.

property compressor_enable

Control whether the compressor is enabled: `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

Set active setup file (0) to copy to setup n (1 - 12).

Type
int

property current_cycle_count

Get the number of cycles to do

Type
int

property cycling_enable

Control whether cycling is enabled.

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

Control DUT mode ("On" or "OFF")

Type
string

property dut_temperature

Get 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

Get 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

Get 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

Set which setup file to load and load it.

Valid range is between 1 to 12.

Type
int

property local_lockout

Control whether to lock locally: `True` disables TS GUI, `False` enables it.

property main_air_flow_rate

Get 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

Get 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

Get 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

Control whether it is in remote mode.

reset()

Reset (force) the System to the Operator screen.

Returns
self

property set_point_number

Control the setpoint index 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

Get 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

Get temperature condition status register.

Type
TemperatureStatusCode

property temperature_event_status

Get temperature event status register.

Type
TemperatureStatusCode

Hint: Reading will clear register content.

property temperature_limit_air_dut

Get 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

Control 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

Control 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

Control 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

Control 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

Control current cycle count (1 - 9999).

Type
int

Sending 0 will stop cycling

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters
query_delay – Delay between writing and reading in seconds. None is default delay.

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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

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*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

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

7.61.2 Temptronic ATS525 Thermostream

class pymeasure.instruments.temptronic.**ATS525**(*adapter*, *name='Temptronic ATS-525 Thermostream'*, ***kwargs*)

Bases: *ATSBase*

Represent the TemptronicATS525 instruments.

property system_current

Get operating current.

7.61.3 Temptronic ATS545 Thermostream

class pymeasure.instruments.temptronic.**ATS545**(*adapter*, *name='Temptronic ATS-545 Thermostream'*, ***kwargs*)

Bases: *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

7.61.4 Temptronic ECO560 Thermostream

class pymeasure.instruments.temptronic.**ECO560**(*adapter*, *name='Temptronic ECO-560 Thermostream'*, ***kwargs*)

Bases: *ATSBase*

Represent the TemptronicECO560 instruments.

7.62 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*.

7.62.1 TEXIO PSW-360L30 Power Supply

class pymeasure.instruments.texio.**TexioPSW360L30**(*adapter*, *name='TEXIO PSW-360L30 Power Supply'*, ***kwargs*)

Bases: *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)
```

property applied

Control voltage (volts) and current (amps) simultaneously. 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.

check_errors()

Log any system errors reported by the instrument.

check_get_errors()

Check for errors after having gotten a property and log them.

Called if `check_get_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

clear()

Clear the instrument status byte.

property complete

Get the synchronization bit.

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

Get the current (in Ampere) the dc power supply is putting out.

property current_limit

Control 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. (float)

property enabled

Control whether the output is enabled, see *output_enabled*.

property error

Get the next error of the instrument (list of code and message).

property id

Get the identification of the instrument.

property next_error

Get the next error in the queue. If you want to read and log all errors, use *check_errors()* instead.

property options

Get the device options installed.

property output_enabled

Control whether the source is enabled, takes values True or False. (bool)

property power

Get the power (in Watt) the dc power supply is putting out.

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()

Reset the instrument.

shutdown()

Disable output, call parent function

property status

Get the status byte and Master Summary Status bit.

property voltage

Get the voltage (in Volt) the dc power supply is putting out.

property voltage_setpoint

Control 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. (float)

wait_for(*query_delay=None*)

Wait for some time. Used by 'ask' to wait before reading.

Parameters

query_delay – Delay between writing and reading in seconds. None is default delay.

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.

7.63 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*.

7.63.1 Thermotron 3800 Oven

class pymeasure.instruments.thermotron.**Thermotron3800**(*adapter*, *name='Thermotron 3800'*, ***kwargs*)

Bases: *Instrument*

Represents the Thermotron 3800 Oven. For now, this driver only supports using Control Channel 1. There is a 1000ms built in wait time after all write commands.

class Thermotron3800Mode(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Bit	Mode
0	Program mode
1	Edit mode (controller in stop mode)
2	View program mode
3	Edit mode (controller in hold mode)
4	Manual mode
5	Delayed start mode
6	Unused
7	Calibration mode

property id

Get the instrument identification

Returns
String

initalize_oven(*wait=True*)

The manufacturer recommends a 3 second wait time after after initializing the oven. The optional "wait" variable should remain true, unless the 3 second wait time is taken care of on the user end. The wait time is split up in the following way: 1 second (built into the write function) + 2 seconds (optional wait time from this function (initialize_oven)).

Returns
None

property mode

Get the operating mode of the oven.

Returns
Tuple(String, int)

run()

Starts temperature forcing. The oven will ramp to the setpoint.

Returns
None

property setpoint

Control the setpoint of the oven in Celsius. (float) "setpoint" will not update until the "run()" command is called. After setpoint is set to a new value, the "run()" command must be called to tell the oven to run to the new temperature.

Returns
None

stop()

Stops temperature forcing on the oven.

Returns
None

property temperature

Get the current temperature of the oven via built in thermocouple. Default unit is Celsius, unless changed by the user.

Returns
float

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.

7.64 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*.

7.64.1 Thorlabs PM100USB Powermeter

class pymeasure.instruments.thorlabs.**ThorlabsPM100USB**(*adapter*, *name='ThorlabsPM100USB powermeter'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents Thorlabs PM100USB powermeter.

property energy

Measure the energy in J.

property power

Measure the power in W.

property wavelength

Control the wavelength in nm.

property wavelength_max

Measure maximum wavelength, in nm

property wavelength_min

Measure minimum wavelength, in nm

7.64.2 Thorlabs Pro 8000 modular laser driver

class pymeasure.instruments.thorlabs.**ThorlabsPro8000**(*adapter*, *name='Thorlabs Pro 8000'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents Thorlabs Pro 8000 modular laser driver

property LDCCurrent

Control 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

Control TEC temperature

property TEDStatus

Control TEC status. Allowed values are: ['ON', 'OFF']

property slot

Control slot selection. Allowed values are: range(1, 9)

7.65 Thyracont

This section contains specific documentation on the Thyracont instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.65.1 Smartline V1 Transmitter Series

class pymeasure.instruments.thyracont.smartline_v1.**SmartlineV1**(*adapter*, *name='Thyracont Vacuum Gauge V1'*, *address=1*, *baud_rate=9600*, ***kwargs*)

Bases: *Instrument*

Thyracont Vacuum Instruments Smartline gauges with Communication Protocol V1.

Devices using Protocol V1 were manufactured until 2017.

Connection to the device is made through an RS485 serial connection. The default communication settings are baudrate 9600, 8 data bits, 1 stop bit, no parity, no handshake.

A communication packages is structured as follows:

Characters 0-2: Address for communication Character 3: Command character, uppercase letter for reading and lowercase for writing Characters 4-n: Data for the command, can be empty. Character n+1: Checksum calculated by: (sum of the decimal value of bytes 0-n) mod 64 + 64 Character n+2: Carriage return

Parameters

- **adapter** – pyvisa resource name of the instrument or adapter instance
- **name** (*string*) – Name of the instrument.
- **address** (*int*) – RS485 adddress of the instrument 1-15.
- **baud_rate** (*int*) – baudrate used for the communication with the device.
- **kwargs** – Any valid key-word argument for Instrument

property cathode_enabled

Control the hot/cold cathode state of the pressure gauge.

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

If you override this method, you may choose to raise an Exception for certain errors.

Returns
List of error entries.

property device_type

Get the device type.

property display_unit

Control the display's pressure unit.

property pressure

Get the pressure measurement in mbar.

read()

Reads a response message from the instrument.

This method also checks for a correct checksum.

Returns
the data fields

Return type
string

Raises
ValueError – if a checksum error is detected

write(*command*)

Writes a command to the instrument.

This method adds the required address and checksum.

Parameters
command (*str*) – command to be sent to the instrument

7.65.2 Smartline V2 Transmitter Series

class pymeasure.instruments.thyracont.smartline_v2.**SmartlineV2**(*adapter*, *name='Thyracont SmartlineV2 Transmitter'*, *baud_rate=115200*, *address=1*, *timeout=250*, ***kwargs*)

Bases: *Instrument*

A Thyracont vacuum sensor transmitter of the Smartline V2 series.

You may subclass this Instrument and add the appropriate channels, see the following example.

```
from pymeasure.instruments import Instrument
from pymeasure.instruments.thyractont import SmartlineV2

PiezoAndPiraniInstrument(SmartlineV2):
    piezo = Instrument.ChannelCreator(Piezo)
    pirani = Instrument.ChannelCreator(Pirani)
```

Communication Protocol v2 via RS485:

- Everything is sent as ASCII characters
- **Package (bytes and usage):**
  - 0-2 address, 3 access code, 4-5 command, 6-7 data length.
  - if data: 8-n data to be sent, n+1 checksum, n+2 carriage return
  - if no data: 8 checksum, 9 carriage return
- **Access codes (request: master->transmitter, response: transmitter->master):**
  - read: 0, 1
  - write: 2, 3
  - factory default: 4,5
  - error: -, 7
  - binary 8, 9
- Data length is number of data in bytes (padding with zeroes on left)
- Checksum: Add the decimal numbers of the characters before, mod 64, add 64, show as ASCII.

Parameters
adress – The device address in the range 1-16.

class Sources(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

property analog_output_setting

Get current analog output setting. See manual.

ask(*command_message*, *query_delay=None*)

Ask for some value and check that the response matches the original command.

Parameters
command_message (*str*) – Access code, command, length, and content. The command sent is compared to the response command.

ask_manually(*accessCode*, *command*, *data=''*, *query_delay=None*)

Send a message to the transmitter and return its answer.

Parameters

- **accessCode** – How to access the device.
- **command** – Command to send to the device.
- **data** – Data for the command.
- **query_delay** (*float*) – Time to wait between writing and reading in seconds.

Return str
Response from the device after error checking.

property baud_rate

Set the device baud rate.

property bootloader_version

Get the bootloader version.

check_set_errors()

Check the errors after setting a property.

property device_address

Set the device address.

property device_serial

Get the transmitter device serial number.

property device_type

Get the device type, like 'VSR205'.

property device_version

Get the device hardware version.

property display_data

Control the display data source (strict SOURCES).

property display_orientation

Control the orientation of the display in relation to the pipe ('top', 'bottom').

property display_unit

Control the unit shown in the display. ('mbar', 'Torr', 'hPa')

property firmware_version

Get the firmware version.

get_sensor_transition()

Get the current sensor transition between sensors.

return interpretation:

- **direct**
  switch at 1 mbar.

- **continuous**
  switch between 5 and 15 mbar.
- **F[float]T[float]**
  switch between low and high value.
- **D[float]**
  switch at value.

property operating_hours

Measure the operating hours.

property pressure

Get the current pressure of the default sensor in mbar

property product_name

Get the product name (article number).

property range

Get the measurement range in mbar.

read(*command=None*)

Read from the device and do error checking.

Parameters
command (*str*) – Original command sent to the device to compare it with the response. None deactivates the check.

property sensor_serial

Get the sensor head serial number.

set_continuous_sensor_transition(*low*, *high*)

Set the sensor transition mode to "continuous" mode between *low* and *high* (floats).

set_default_sensor_transition()

Set the senstor transition mode to the default value, depends on the device.

set_direct_sensor_transition(*transition_point*)

Set the sensor transition to "direct" mode.

Parameters
transition_point (*float*) – Switch between the sensors at that value.

set_high(*high=''*)

Set the high pressure to *high* pressure in mbar.

set_low(*low=''*)

Set the low pressure to *low* pressure in mbar.

write(*command*)

Write a command to the device.

write_composition(*accessCode*, *command*, *data=''*)

Write a command with an accessCode and optional data to the device.

Parameters

- **accessCode** – How to access the device.
- **command** – Two char command string to send to the device.

- **data** – Data for the command.

class pymeasure.instruments.thyracont.smartline_v2.**VSH**(*adapter*, *name='Thyracont SmartlineV2 Transmitter'*, *baud_rate=115200*, *address=1*, *timeout=250*, ***kwargs*)

Bases: *SmartlineV2*

Vacuum transmitter of VSH series with both a pirani and a hot cathode sensor.

hotcathode

Channel
HotCathode

pirani

Channel
Pirani

class pymeasure.instruments.thyracont.smartline_v2.**VSM**(*adapter*, *name='Thyracont SmartlineV2 Transmitter'*, *baud_rate=115200*, *address=1*, *timeout=250*, ***kwargs*)

Bases: *SmartlineV2*

Vacuum transmitter of VSM series with both piece and cold cathode sensor.

coldcathode

Channel
ColdCathode

piezo

Channel
Piezo

class pymeasure.instruments.thyracont.smartline_v2.**VSP**(*adapter*, *name='Thyracont SmartlineV2 Transmitter'*, *baud_rate=115200*, *address=1*, *timeout=250*, ***kwargs*)

Bases: *SmartlineV2*

Vacuum transmitter of VSP/VCP series with a piezo sensor.

piezo

Channel
Piezo

class pymeasure.instruments.thyracont.smartline_v2.**VSR**(*adapter*, *name='Thyracont SmartlineV2 Transmitter'*, *baud_rate=115200*, *address=1*, *timeout=250*, ***kwargs*)

Bases: *SmartlineV2*

Vacuum transmitter of VSR/VCR series with both a piezo and a pirani sensor.

piezo

Channel
Piezo

pirani

Channel
Pirani

7.66 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*.

7.66.1 Toptica IBeam Smart Laser diode

class pymeasure.instruments.toptica.ibeamsmart.**IBeamSmart**(*adapter*, *name='Toptica IBeam Smart laser diode'*, *baud_rate=115200*, ***kwargs*)

Bases: *Instrument*

IBeam Smart laser diode

For the usage of the different diode driver channels, see the manual

```
laser = IBeamSmart("SomeResourceString")
laser.emission = True
laser.ch_2.power = 1000  # µW
laser.ch_2.enabled = True
laser.shutdown()
```

Parameters

- **adapter** – pyvisa resource name or adapter instance.
- **baud_rate** – The baud rate you have set in the instrument.
- ****kwargs** – Any valid key-word argument for VISAAdapter.

ch_1

Channel
DriverChannel

ch_2

Channel
DriverChannel

ch_3

Channel
DriverChannel

ch_4

Channel
DriverChannel

ch_5

Channel
DriverChannel

property channel1_enabled

Control status of Channel 1 of the laser (bool).

Deprecated since version 0.12: Use `ch_1.enabled` instead.

property channel2_enabled

Control status of Channel 2 of the laser (bool).

Deprecated since version 0.12: Use `ch_2.enabled` instead.

check_set_errors()

Check for errors after having gotten a property and log them.

Checks if the last reply is only '[OK]', otherwise a ValueError is raised and the read buffer is flushed because one has to assume that some communication is out of sync.

property current

Measure the laser diode current in mA.

disable()

Shutdown all laser operation.

property emission

Control emission status of the laser diode driver (bool).

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

Control emission status of the laser diode driver (bool).

Deprecated since version 0.12: Use attr:*emission* instead.

property power

Control actual output power in µW of the laser system. In pulse mode this means that the set value might not correspond to the readback one (float up to 200000).

read()

Read a reply of the instrument and extract the values, if possible.

Reads a reply of the instrument which consists of at least two lines. The initial ones are the reply to the command while the last one should be '[OK]' which acknowledges that the device is ready to receive more commands.

Note: '[OK]' is always returned as last message even in case of an invalid command, where a message indicating the error is returned before the '[OK]'

Value extraction: extract <value> from 'name = <value> [unit]'. If <value> can not be identified the original string is returned.

Returns
string containing the ASCII response of the instrument (without '[OK]').

property serial

Get Serial number of the laser system.

shutdown()

Brings the instrument to a safe and stable state.

property system_temp

Measure base plate (heatsink) temperature in degree centigrade.

property temp

Measure the temperature of the laser diode in degree centigrade.

property version

Get Firmware version number.

class pymeasure.instruments.toptica.ibeamsmart.**DriverChannel**(*parent*, *id*)

Bases: *Channel*

A laser diode driver channel for the IBeam Smart laser.

property enabled

Control the enabled state of the driver channel.

property power

Set the output power in µW (float up to 200000).

7.67 Velleman

This section contains specific documentation on the Velleman instruments that are implemented. If you are interested in an instrument not included, please consider *adding the instrument*.

7.67.1 Velleman K8090 8-channel relay board

class pymeasure.instruments.velleman.**VellemanK8090**(*adapter*, *name='Velleman K8090'*, *timeout=100*, ***kwargs*)

Bases: *Instrument*

For usage with the K8090 relay board, by Velleman.

View the "K8090/VM8090 PROTOCOL MANUAL" for the serial command instructions.

The communication is done by serial USB. The IO settings are fixed:

Baud rate	19200
Data bits	8
Parity	None
Stop bits	1
Flow control	None

A short timeout is recommended, since the device is not consistent in giving status messages and serial timeouts will occur also in normal operation.

Use the class like:

```
from pymeasure.instruments.velleman import VellemanK8090, VellemanK8090Switches as␣
↪Switches

instrument = VellemanK8090("ASRL1::INSTR")

# Get status update from device
last_on, curr_on, time_on = instrument.status

# Toggle a selection of channels on
instrument.switch_on = Switches.CH3 | Switches.CH4 | Switches.CH5
```

check_set_errors()

Check for errors after having set a property and log them.

Called if `check_set_errors=True` is set for that property.

The K8090 replies with a status after a switch command, but **only** after any switch actually changed. In order to guarantee the buffer is empty, we attempt to read it fully here. No actual error checking is done here!

Returns
List of error entries.

read(***kwargs*)

The read command specifically for the protocol of the K8090.

This overrides the method from the `instrument` class.

See *write()*, replies from the machine use the same format.

A read will return a list of CMD, MASK, PARAM1 and PARAM2.

property status

Get current relay status. The reply has a different command byte than the request.

Three items (*VellemanK8090Switches* flags) are returned:

- Previous state: the state of each relay before this event
- Current state: the state of each relay now
- Timer state: the state of each relay timer

property switch_off

Set channels to off state. See *switch_on* for more details.

property switch_on

Set channels to on state. Other channels are unaffected. Pass either a list or set of channel numbers (starting at 1), or pass a bitmask.

After switching this waits for a reply from the device. This is only send when a relay actually toggles, otherwise expect a blocking time equal to the communication timeout If speed is important, avoid calling *switch_* unnecessarily.

property version

Get firmware version, as (year - 2000, week). E.g. `(10, 1)`

write(*command*, ***kwargs*)

The write command specifically for the protocol of the K8090.

This overrides the method from the `Instrument` class.

Each packet to the device is 7 bytes:

STX (0x04) - CMD - MASK - PARAM1 - PARAM2 - CHK - ETX (0x0F)

Where *CHK* is checksum of the package.

Parameters
command (*str*) – String like "CMD[, MASK, PARAM1, PARAM2]" - only CMD is mandatory

class pymeasure.instruments.velleman.**VellemanK8090Switches**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntFlag

Use to identify switch channels.

7.68 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*.

7.68.1 Yokogawa 7651 Programmable Supply

class pymeasure.instruments.yokogawa.**Yokogawa7651**(*adapter*, *name='Yokogawa 7651 Programmable DC Source'*, ***kwargs*)

Bases: SCPIUnknownMixin, *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

Control the compliance current in Amps,which can take values from 5 to 120 mA.

property compliance_voltage

Control 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

Get 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

Control the source current in Amps, if that mode is active. (float)

property source_current_range

Control the current voltage range in Amps, which can take values: 1 mA, 10 mA, and 100 mA. Currents are truncated to an appropriate value if needed.

property source_enabled

Get 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

Control 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

Control the source voltage in Volts, if that mode is active. (float)

property source_voltage_range

Control the source voltage range in Volts, which can take values: 10 mV, 100 mV, 1 V, 10 V, and 30 V. Voltages are truncated to an appropriate value if needed.

7.68.2 Yokogawa GS200 Source

class pymeasure.instruments.yokogawa.**YokogawaGS200**(*adapter*, *name='Yokogawa GS200 Source'*, ***kwargs*)

Bases: SCPIUnknownMixin, *Instrument*

Represents the Yokogawa GS200 source and provides a high-level interface for interacting with the instrument.

property current_limit

Control 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

Control whether the source is enabled. (bool)

property source_level

Control the output level, either a voltage or a current, depending on the source mode. (float)

property source_mode

Control the source mode. Can be either 'current' or 'voltage'.

property source_range

Control the range (either in voltage or current) of the output. "Range" refers to the maximum source level. (float)

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

Control 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

7.68.3 Yokogawa AQ6370 Series of Optical Spectrum Analyzers

class pymeasure.instruments.yokogawa.aq6370series.**AQ6370Series**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *SCPIMixin*, *Instrument*

Represents Yokogawa AQ6370 Series of optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

abort()
Stop operations such as measurements and calibration.

property active_trace
Control the active trace (str 'A', 'B', 'C', ...).

authenticate_ethernet(*username*, *password=''*)
Authenticate for an ethernet connection.

property automatic_sample_number
Control the automatic sample number (bool).

property calc_result
Get the results of the last analysis.

copy_trace(*source*, *destination*)
Copy the data of specified trace to the another trace.

Parameters

- **source** – Source trace (str 'A', 'B', 'C', ...).
- **destination** – Destination trace (str 'A', 'B', 'C', ...).

delete_trace(*trace*)
Delete the specified trace.

Parameters
trace – Trace to be deleted (str 'ALL', 'A', 'B', 'C', ...).

execute_analysis()
Execute the analysis with the current analysis settings.

`get_analysis`()

Query the analysis results of latest analysis. If no analysis has been performed, returns query error.

`get_xdata`(*trace='TRA'*)

Measure the x-axis data of specified trace, output wavelength in m.

Parameters
`trace` – Trace to measure (str 'A', 'B', 'C', ...).

Returns
The x-axis data of specified trace.

`get_ydata`(*trace='TRA'*)

Measure the y-axis data of specified trace, output power in dBm.

Parameters
`trace` – Trace to measure (str 'A', 'B', 'C', ...).

Returns
The y-axis data of specified trace.

`initiate_sweep`()

Initiate a sweep.

`property level_position`

Control the reference level position regarding divisions (int, smaller than total number of divisions which is either 8, 10 or 12).(dynamic)

`property reference_level`

Control the reference level of main scale of level axis (float in dBm).

`property resolution_bandwidth`

Control the measurement resolution (float in m, discrete values: [0.02e-9, 0.05e-9, 0.1e-9, 0.2e-9, 0.5e-9, 1e-9, 2e-9] m).(dynamic)

`property sample_number`

Control the sample number (int from 51 to 50001).

`property sensitivity`

Control the sweep sensitivity (str 'NHLD', 'NAUT', 'NORM', 'MID', 'HIGH1', 'HIGH2', 'HIGH3')

`set_level_position_to_max`()

Set the reference level position to the maximum value.

`property sweep_complete`

Get the completion status of the sweep (bool, True if complete).

`property sweep_mode`

Control the sweep mode (str 'SINGLE', 'REPEAT', 'AUTO', 'SEGMENT').

`property sweep_time_interval`

Control the sweep time interval (int from 0 to 99999 s).

`property transfer_format`

Control the data transfer format. It returns to default ASCII at reset.

`trigger`()

Perform a single sweep according to previous conditions.

wait_for_sweep_complete(*should_stop=<function AQ6370Series.<lambda>>*, *timeout=300*)

Block the program, waiting for the sweep to complete.

Parameters

- **should_stop** – Function that returns True to stop waiting.
- **timeout** – Maximum waiting time, in seconds.

Returns

True when sweep completed, False if stopped by should_stop.

Raises

TimeoutError – If the temperature does not stabilize within the timeout period.

property wavelength_automatic_center

Control whether the wavelength center follows the maximum (bool).

property wavelength_center

Control measurement condition center wavelength (float in m).(dynamic)

property wavelength_span

Control wavelength span (float from 0 to 1100e-9 m).(dynamic)

property wavelength_start

Control the measurement start wavelength (float from 50e-9 to 2250e-9 in m).(dynamic)

property wavelength_stop

Control the measurement stop wavelength (float from 50e-9 to 2250e-9 in m).(dynamic)

class pymeasure.instruments.yokogawa.aq6370series.**AQ6370E**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6370Series*

Represents Yokogawa AQ6370E optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

property sensitivity_level

Control the sweep sensitivity by specifying the sensitivity level you want to measure at, in dBm. The sensitivity closest to that level, and the sweep speed are automatically selected.

property sweep_speed

Control the sweep speed (str '1x' or '2x' for double speed).

class pymeasure.instruments.yokogawa.aq6370series.**AQ6370D**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6370Series*

Represents Yokogawa AQ6370D optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

property sweep_speed

Control the sweep speed (str '1x' or '2x' for double speed).

class pymeasure.instruments.yokogawa.aq6370series.**AQ6370C**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6370Series*

Represents Yokogawa AQ6370C optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

property sweep_speed

Control the sweep speed (str '1x' or '2x' for double speed).

class pymeasure.instruments.yokogawa.aq6370series.**AQ6373**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6370Series*

Represents Yokogawa AQ6373 optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

class pymeasure.instruments.yokogawa.aq6370series.**AQ6373B**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6373*

Represents Yokogawa AQ6373B variant optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

property sweep_speed

Control the sweep speed (str '1x' or '2x' for double speed).

class pymeasure.instruments.yokogawa.aq6370series.**AQ6375**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6370Series*

Represents Yokogawa AQ6375 optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

class pymeasure.instruments.yokogawa.aq6370series.**AQ6375B**(*adapter*, *name='Yokogawa AQ3670D OSA'*, *baud_rate=115200*, ***kwargs*)

Bases: *AQ6375*

Represents Yokogawa AQ6375B variant optical spectrum analyzer.

TRA

Channel
Trace

TRB

Channel
Trace

TRC

Channel
Trace

TRD

Channel
Trace

TRE

Channel
Trace

TRF

Channel
Trace

TRG

Channel
Trace

property sweep_speed

Control the sweep speed (str '1x' or '2x' for double speed).

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.

Make sure `git --version` works from comand line. If you are using Github Desktop, you may need to add Git to your path in each terminal window. On windows anaconda prompt it would be `set PATH="C:Users<USERNAME>AppDataLocalGitHubDesktopapp-<#######>resourcesappgitcmd";%PATH%`

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 and select optional extras.

```
cd /path/for/code/pymeasure
pip install -e .[tests,docs]
```

The `-e` will allow you to edit the files of PyMeasure and see the changes reflected. The square brackets include extra groups that allow you to run tests and build the documentation locally. 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!

Depending on your Python installation you may get an error messages saying that the file setup.py is missing or similar. Updating pip may solve the problem.

```
python -m pip install pip --upgrade
```

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.

Pymeasure instruments communicate with the devices via transfer of bytes or ASCII characters encoded as bytes. For ease of use, we have *property creators* to easily create python properties. Similarly, we have creators to easily implement *channels*. Finally, for a smoother implementation process and better maintenance, we have *tests*.

The following sections 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.

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) 2013-2025 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 *Instruments with channels*.

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.

Generic types mixins

The *generic_types* module contains mixin classes for common types. For example, if an instrument complies to SCPI standards, you can add *SCPIMixin* to your instrument:

```
from pymeasure.instruments.generic_types import SCPIMixin

class SomeSCPIInstrument(SCPIMixin, Instrument):
    """This instrument has properties and methods defined for all SCPI instruments"""
```

This mixin adds default SCPI properties like *id*, *status* and default methods like *clear()* and *reset()* to `SomeSCPIInstrument`.

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. Note that starting from Python version 3.11, the printed format of the `IntEnum` and `IntFlag` has been changed to return numeric value; however, the symbolic name can be obtained by printing its `repr` or the `.name` property, or returning the value in a REPL.

```
>>> 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:
...     current_mode
...     print(repr(current_mode))
...     print(f'Mode value: {current_mode}')
...
<InstrMode.HEATING: 1>
<InstrMode.HEATING: 1>
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
>>> ErrorCode(received_from_device)
<ErrorCode.TEMPSENSOR_FAILURE|COOLER_FAILURE|HEATER_FAILURE: 7>
```

`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,
        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 `:TEMP?` 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.setting* 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* functions 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).""",
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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 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 methods *check_get_errors()* and *check_set_errors()*, respectively, will be called after device communication has concluded. In the default implementation, for simplicity both methods call *check_errors()*. To read the automatic response of instruments that respond to every set command with an acknowledgment or error, override *check_set_errors()* as needed.

10.5.8 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 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.

10.5.9 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 accommodating 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 occurrence 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".

Please add any created channel classes to the documentation. In the instrument's documentation file, you may add

```
.. autoclass:: pymeasure.instruments.MANUFACTURER.INSTRUMENT.CHANNEL
    :members:
    :show-inheritance:
```

MANUFACTURER is the folder name of the manufacturer and *INSTRUMENT* the file name of the instrument definition, which contains the *CHANNEL* class. You may link in the instrument's docstring to the channel with `:class:`CHANNEL``

To simplify and standardize the creation of channels in an `Instrument` class, there are two classes that can be used. For instruments with fewer than 16 channels, *ChannelCreator* should be used to explicitly declare each individual channel. For instruments with more than 16 channels, the *MultiChannelCreator* can create multiple channels in a single declaration.

10.7.1 Adding a channel with `ChannelCreator`

For instruments with fewer than 16 channels the class *ChannelCreator* should be used to assign each channel interface to a class attribute. *ChannelCreator* constructor accepts two parameters, the channel class for this channel interface, and the instrument's channel id for the channel interface.

In this example, we are defining a channel class and an instrument driver class. The `VoltageChannel` channel class will be used for controlling two channels in our `ExtremeVoltage5000` instrument. In the `ExtremeVoltage5000` class we declare two class attributes with *ChannelCreator*, `output_A` and `output_B`, which will become our channel interfaces.

```
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 ExtremeVoltage5000(Instrument):
    """An instrument with channels."""
    output_A = Instrument.ChannelCreator(VoltageChannel, "A")
    output_B = Instrument.ChannelCreator(VoltageChannel, "B")
```

At instrument class instantiation, the instrument class will create an instance of the channel class and assign it to the class attribute name. Additionally the channels will be collected in a dictionary, by default named `channels`. We can access the channel interface through that class name:

```
extreme_inst = ExtremeVoltage5000('COM3')
# Set channel A voltage
extreme_inst.output_A.voltage = 50
# Read channel B voltage
chan_b_voltage = extreme_inst.output_B.voltage
```

Or we can access the channel interfaces through the `channels` collection:

```
# Set channel A voltage
extreme_inst.channels['A'].voltage = 50
# Read channel B voltage
chan_b_voltage = extreme_inst.channels['B'].voltage
```

10.7.2 Adding multiple channels with `MultiChannelCreator`

For instruments greater than 16 channels the class *`MultiChannelCreator`* can be used to easily generate a list of channels from one class attribute declaration.

The *`MultiChannelCreator`* constructor accepts a single channel class or list of channel classes, and a list of corresponding channel 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 generate channel interfaces as class attribute names composing of the prefix (default "ch_") and channel id, e.g. the channel with id "A" will be added as attribute `ch_A`. While *`ChannelCreator`* creates a channel interface for each class attribute, *`MultiChannelCreator`* creates a channel collection for the assigned class attribute. It is recommended you use the class attribute name `channels` to keep the codebase homogenous.

To modify our example, we will use *`MultiChannelCreator`* to generate 24 channels of the `VoltageChannel` class.

```
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 MultiExtremeVoltage5000(Instrument):
    """An instrument with channels."""
    channels = Instrument.MultiChannelCreator(VoltageChannel, list(range(1,25)))
```

We can now access the channel interfaces through the generated class attributes:

```
extreme_inst = MultiExtremeVoltage5000('COM3')
# Set channel 5 voltage
extreme_inst.ch_5.voltage = 50
# Read channel 16 voltage
chan_16_voltage = extreme_inst.ch_16.voltage
```

Because we use *channels* as the class attribute for `MultiChannelCreator`, we can access the channel interfaces through the `channels` collection:

```
# Set channel 10 voltage
extreme_inst.channels[10].voltage = 50
# Read channel 22 voltage
chan_b_voltage = extreme_inst.channels[22].voltage
```

10.7.3 Advanced channel management

Adding / removing channels

In order to add or remove programmatically channels, use the parent's *add_child()*, *remove_child()* methods.

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()`).

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.MultiChannelCreator(
        (VoltageChannel, VoltageChannelPrefix),
        ("A", "B"),
        prefix="an_")
    digital = Instrument.MultiChannelCreator(VoltageChannel, (0, 1, 2), prefix="di_")
    power = Instrument.ChannelCreator(PowerChannel)
```

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):
```

(continues on next page)

(continued from previous page)

```
        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.

        :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]
```

(continues on next page)

(continued from previous page)

```
        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:
            # 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():
```

(continues on next page)

(continued from previous page)

```
    """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`

Test generator

In order to facilitate writing tests, if you already have working code and a device at hand, we have a *Generator* for tests. You can control your instrument with the TestGenerator as a middle man. It logs the method calls, the device communication and the return values, if any, and writes tests according to these log entries.

```
from pymeasure.generator import Generator
from pymeasure.instruments.hcp import TC038

generator = Generator()
inst = generator.instantiate(TC038, adapter, 'hcp', adapter_kwargs={'baud_rate': 9600})
```

As a first step, this code imports the Generator and generates a middle man instrument. The `instantiate()` method creates an instrument instance and logs the communication at startup. The Generator creates a special adapter for the communication with the device. It cannot inspect the instrument's `__init__()`, however. Therefore you have to specify the **all** connection settings via the `adapter_kwargs` dictionary, even those, which are defined in `__init__()`. These adapter arguments are not written to tests. If you have arguments for the instrument itself, e.g. a RS485 address, you may give it as a keyword argument. These additional keyword arguments are included in the tests.

Now we can use `inst` as if it were created the normal way, i.e. `inst = TC038(adapter)`, where `adapter` is some resource string. Having gotten and set some properties, and called some methods, we can write the tests to a file.

```
inst.information  # returns the 'information' property, e.g. 'UT150333 V01.
↪R001111222233334444'
inst.setpoint = 20
assert inst.setpoint == 20
inst.setpoint = 60

generator.write_file(file)
```

The following data will be written to `file`:

```
import pytest

from pymeasure.test import expected_protocol
from pymeasure.instruments.hcp import TC038


def test_init():
    with expected_protocol(
            TC038,
            [(b'\x0201010WRS01D0002\x03', b'\x020101OK\x03')],
    ):
        pass  # Verify the expected communication.


def test_information_getter():
    with expected_protocol(
            TC038,
            [(b'\x0201010WRS01D0002\x03', b'\x020101OK\x03'),
             (b'\x0201010INF6\x03', b'\x020101OKUT150333 V01.R001111222233334444\x03')],
    ) as inst:
        assert inst.information == 'UT150333 V01.R001111222233334444'


@pytest.mark.parametrize("comm_pairs, value", (
    ([(b'\x0201010WRS01D0002\x03', b'\x020101OK\x03'),
      (b'\x0201010WWRD0120,01,00C8\x03', b'\x020101OK\x03')],
     20),
    ([(b'\x0201010WRS01D0002\x03', b'\x020101OK\x03'),
      (b'\x0201010WWRD0120,01,0258\x03', b'\x020101OK\x03')],
     60),
))
def test_setpoint_setter(comm_pairs, value):
    with expected_protocol(
            TC038,
            comm_pairs,
    ) as inst:
        inst.setpoint = value


def test_setpoint_getter():
    with expected_protocol(
```

(continues on next page)

(continued from previous page)

```
            TC038,
            [(b'\x0201010WRS01D0002\x03', b'\x020101OK\x03'),
             (b'\x0201010WRDD0120,01\x03', b'\x020101OK00C8\x03')],
    ) as inst:
        assert inst.setpoint == 20.0
```

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.

10.10 Solutions for implementation challenges

This is a list of less common challenges, their solutions, and example instruments.

10.10.1 General issues

- Small numbers (<1e-5) are shown as 0 with `%f`. If an instrument understands exponential notation, you can use `%g`, which switches between floating point and exponential format, depending on the exponent.

10.10.2 Communication protocol issues

- The instrument answers every message, even a setting command. You can set the setting's `check_set_errors = True` parameter and redefine `check_set_errors()` to read an answer, see *hcp.TC038D*
- Binary, frame-based communication, see *hcp.TC038D*
- All replies have the same length, see *aja.DCXS*
- The device generates garbage messages at startup, cluttering the buffer, see *aja.DCXS*
- An instrument and its channel need to override *values*, but it has to use the correct *ask* method as well, see *tcpowerconversion.CXN*

10.10.3 Channels

- Not all channels have the same features, see *MKS937B*
- Channel names in the communication (1, 2, 3) differ from front panel (A, B, C), see *AdvantestR624X*
- A family of instruments in which a property of the channels is different for different members of the family , see *AnritsuMS464xB*
- If you want to document the type of your instrument's channels (with a clickable link), check out the source of the *Aim-TTI PL Series Power Supplies* page.

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.

You may add type hints as you see fit. All type hints should adhere to the guidelines set out in the typing package.

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 preferred.

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 properties, start them with "Control", "Get", "Measure", or "Set" to indicate the kind of property, as it is not visible after import, whether a property is gettable ("Get" or "Measure", e.g. for a `measurement()`), settable ("Set", for a `setting()`), or both ("Control", for a `control()`). 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. For example a docstring could be `"""Control the voltage in Volts (float strictly from -1 to 1)."""`.

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
Juraj Jašík
Max Herbold
Alexander Wichers
Ashok Bruno
Robert Roos
Sebastien Weber
Sebastian Neusch
Felix Thouin
Ulrich Sauter
Guus Kuiper
Bernhard Lang
Armindo Pinto
Frank Wu
Heinz-Alexander Fütterer
Per-Olof Svensson
Karl Komierowski
Alec Vercruysse
Matthew Zenaldin
Canyon Clark
Connor Carr
Jannis Kleine-Schönepauck
Douwe den Blanken
Till Zürner
J. A. Wilcox
Nick James Kirkby
Konrad Gralher
David Ziliak
```

(continues on next page)

(continued from previous page)

```
David Sun
Kévin Petit
Félix Thouin
Orion Smedley
Samuel Simpson
Isha Sharma
Dawid Maślanka
Felix Thouin
Emmanuel Ferdman
Johannes Rothmayr
Fred Gillard
Dimitrios Simatos
Dongkai Pan
Julian van Doorn
Jörg Wunsch
Falk Korndörfer
Henk Vergone
Sergey Ilyev
Alfons Schuck
Patrick Mischke
```

CHAPTER

THIRTEEN

LICENSE

CHAPTER

FOURTEEN

CHANGELOG

14.1 Upcoming Release

14.1.1 Removed

- Remove support for Python 3.8
- Remove deprecated Adapter and Instrument parameters `preprocess_reply`, `query_delay`
- Remove deprecated Adapter methods `ask`, `values`, and `binary_values`
- Remove deprecated Telnet and VXI adapters

14.1.2 Changed features

- `Instrument.control` does not apply `get_process` to a returned list anymore, only to a single value. Use `get_process_list` parameter instead for processing a list of values.

14.1.3 Deprecated

- Replaced `sensitvity` attribute of `pymeasure/instruments/srs/SR860.py` by `sensitivity`
- Replaced `filer_synchronous` attribute of `pymeasure/instruments/srs/SR860.py` by `filter_synchronous`

14.2 Version 0.15.0 (2025-01-15)

Main items of this new release:

- pyproject.toml replaces setup.cfg
- 12 new instruments

14.2.1 Deprecated features

- The FSL class imported from `pymeasure/instruments/rohdeschwarz/fsl.py` is deprecated, the current version can be found in `pymeasure/instruments/rohdeschwarz/fsseries.py`.

14.2.2 Instruments

- Add Agilent E5062A (@AlecVercruysse, #1146)
- Add Hewlett Packard 8753E VNA (@Sionwage, #1004)
- Add Keithley DAQ6510 (@Aphelion82, #1128)
- Add Keithley2281S (@PfannenHans, #1054)
- Add LD400P electronic load (@RobertoRoos, #1168)
- Add Philips PM6669 Universal Frequency Counter (@dirkjankrijnders, #570)
- Add Rohde&Schwarz FSSeries class for instruments such as FSL (previously stand-alone class), FSW and others (@jnnskls, #1156) Previously, the instrument class FSL could be imported from `pymeasure/instruments/rohdeschwarz/fsl.py`, which can now be imported from `pymeasure/instruments/rohdeschwarz/fsseries.py` alongside the base class FSSeries and FSW.
- Add Rigol dg822 waveform generator (@fthouin, #1159)
- Add Siglent SDS1072CML oscilloscope (@fthouin, #1080, 1195)
- Add more Thyracont vacuum transducer device types (@dkriegner, #1143)
- Fix Agilent E5062A: docs + rename *active_traces* to *visible_traces* (@AlecVercruysse, #1174)
- Fix AnritsuMS2090A: Confirm SCPIMixin (@icchalmers, #1191)
- Fix race condition in test suite due to HP8116A tests (@kpet, #1145)
- Fix description of `Keithley2000.measure_resistance` method (@J3NZ0L, #1193)
- Fix test for Keithley6510 (@BenediktBurger, #1135)
- Fix Kepco BOP power supplies documentation (@JAW90, #1199)
- Update AFG3152CChannel to docs (@ssimpson-ens, #1176)
- Update Keysight E3631A with added `output_enabled` property to individual channels (@inonRAAAM, #1209)
- Update Teledyne with VBS methods (@RobertoRoos, #1166)

14.2.3 Automation

- Explicitly set encoding to utf8 when writing and reading data to file, allowing the use of special characters. Previously the encoding was not explicitly set, this could potentially disrupt loading old data-files; if this is required, the encoading can be changed by changing (e.g., monkey-patching) the `pymeasure.experiment.Results.ENCODING` property. (@CasperSchippers, #1123)
- Add `has_next` method to Procedure class (@Did-Mas, #1185)
- Change logger from root to module-specific logger. (@mmerlo, #1165)

14.2.4 Documentation

- Fix a couple of typos in procedure.rst (@kpet, #1141)
- Update contribute.rst also for GitHub Desktop (@bernhardlang, @OrionSmedley, #983, #1171)
- Fix documentation errors (@msmttchr, @kpet, #1133, #1148)
- Update InstrumentKit reference (@emmanuel-ferdman, #1187)
- Change copyright year to 2025 (@BenediktBurger, #1210)

14.2.5 Miscellaneous

- Prepare for v0.14.0 (@BenediktBurger, #1104)
- add "Reaveal in File Explorer" to browser item context menu (@Did-Mas, #1188)
- Update numpy 2 test (@CasperSchippers, #1126)
- Added an install group for building documentation (@RobertoRoos, #1180)
- Removed setup.cfg, replaced by pyproject.toml (@RobertoRoos, #1182)

14.2.6 New Contributors

@Sionwage, @bernhardlang, @Aphelion82, @kpet, @mmerlo, @OrionSmedley, @ssimpson-ens, @Did-Mas, @emmanuel-ferdman, @icchalmers, @fthouin, @PfannenHans, @dirkjankrijnders, @J3NZ0L, @inonRAAAM,

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.14.0...v0.15.0

14.3 Version 0.14.0 (2024-05-22)

Main items of this new release:

- Add support for numpy 2.0
- Add support for python 3.12
- Improve academic quotability with an up to date Zenodo DOI and with citation information.
- Add default `queue` method and a `FileInputWidget`, allowing to more quickly get started with the PyMeasure user interface (`ManagedWindow`).
- Add a `SCPIMixin` base class for instruments instead of defining `includeSCPI=True`
- Instrument manufacturer modules are no longer imported in the `pymeasure/instruments/__init__.py` file. Previously, when importing a single instrument into a procedure, all instruments would be imported into memory through the manufacturer modules in `pymeasure/instruments/__init__.py`. Removing manufacturer modules from that file lowers the memory footprint of pymeasure when importing an instrument. Instrument classes will need to be imported from the manufacturer module or explicitly from the instrument driver file. For example, `from pymeasure.instruments import Extreme5000` will need to change to `from pymeasure.instruments.extreme import Extreme5000` or `from pymeasure.instruments.extreme.extreme5000 import Extreme5000`.
- 17 new instruments

14.3.1 Deprecated features

- Remove `TelnetAdapter`, as its library is deprecated (@BenediktBurger, #1045)
- Replaced `directory_input` keyword-argument of `ManagedWindowBase` by `enable_file_input` (@Casper-Schippers, #964)
- Make parameter `includeSCPI` obligatory for all instruments, even those which use SCPI (@BenediktBurger, #1007)
- Setting *includeSCPI=True* is deprecated, inherit instead the `SCPIMixin` class if the device supports SCPI commands.
- Replaced `celcius` attribute of `LakeShoreTemperatureChannel` by `celsius` (@afuetterer, #1003)
- Replaced `error` property of Keithley instruments by `next_error`.
- Replaced `measurement_time` property of Pendulum CNT-91 by `gate_time`.
- Replaced `sample_rate` keyword-argument of `buffer_frequency_time_series` of Pendulum CNT-91 by `gate_time`.
- Replaced MKS937B `unit` to use `instruments/mksinst/mks937b/Unit` instead of strings (@dkriegner, @BenediktBurger #1034)

14.3.2 Instruments mechanics

- Add a SCPI base class `SCPIMixin` as replacement for `includeSCPI=True` (@BenediktBurger, #905, #1007, #1019, #1047)
- Add `next_error` property to SCPI instruments (@BenediktBurger, #1024)
- Make `query_delay=None` the default for `wait_for` (@BenediktBurger, #1077)
- Fix `expected_protocol` using empty dictionary as default value (@BenediktBurger, #1087)
- Remove auto-importing all instruments in `pymeasure/instruments/__init__.py`` (@mcdo0486, #919)
- Add `find_serial_port` to find a serial port by providing USB information (@BenediktBurger, #982)

14.3.3 Instruments

- Add Agilent4294A (@driftregion, #998)
- Add Agilent 4284A by (@ConnorGCarr #1079)
- Add AimTTI PL series power supplies (@guuskuiper, #942)
- Add HP11713A Switch & Attenuator Driver (@neuschs, #970)
- Add HP437B power meter (@neuschs, #979)
- Add Inficon SQM160 SQM-160 multi-film rate/thickness monitor (@dkriegner, #991)
- Add Keithley 2182 (@ConnorGCarr, #1043)
- Add KeithleyDMM6500 (@fwutw, #963)
- Add Kepco BOP 36-12 Bipolar Power Supply (@JAW90, #1086)
- Add KeysightE3631A (@OptimisticBeliever, #990)
- Add Kuhne Electronic KU SG 2.45 250A microwave generator (@jurajjasik, @BenediktBurger, @1108)

- Add MKS 974B vacuum pressure transducer (@dkriegner, #1034)
- Add Proterial rod4 (@ConnorGCarr, #1044)
- Add Racal-Dana 1992 universal counter (@tomverbeure, #798, #1012)
- Add redpitaya board (@seb5g, #1010, #1035)
- Add Teledyne HDO6xxx (@RobertoRoos, #868)
- Add Yokogawa AQ6370D Optical Spectral Analyzer (@jnnskls, #1059)
- Fix property docstrings of several instruments (@BenediktBurger, #1018)
- Fix checksums of hcp TC038D tests (@BenediktBurger, #987)
- Fix Hp8116a (@BenediktBurger, #1088)
- Fix Hp856x to append amplitude units (@neuschs, #977)
- Fix Keysight E36312A confirmed SCPI functionality (@Konradrundfunk, #1107)
- Fix Stanford Research SR830 output conversion (@dkriegner, #1069)
- Fix SR830 missing get_buffer method (@seb5g, #999)
- Fix set command of SR860 aux output (@wehlgrundspitze, #1048)
- Fix Temptronic test to use ns perf counter (@BenediktBurger, #1109, #1110)
- Fix Toptica Ibeamsmart referencing removed adapter function (@BenediktBurger, #1065)
- Fix typos in docstrings for Keithley instruments (@V0XNIHILI, #1071)
- Link Keysight, Agilent, and HP documentation pages. (@BenediktBurger, #1021)
- Update Agilent33500 Series from `.ch[]` to `.channels[]` (@AlecVercruysse, #945)
- Update AWG401x driver to use 'channels' (@mcdo0486, #944)
- Update HP33120A with new burst modulation parameters (@mzen228, #1056)
- Update HP34401A with new remote control command. (@Rybok, #992)
- Update Keithleys' next_error (@msmttchr, #1030)
- Update pendulum CNT-91 (@bleykauf, #988)

14.3.4 GUI

- Add a `FileInputWidget` to choose if and where the experiment data is stored. (@CasperSchippers, #964)
- Add a default `Queue` method for `ManagedWindowBase` is implemented. (@CasperSchippers, #964)
- Fix `ScientificInput` to be locale compatible (@pyZerrenner, #1074)
- Fix exception if loading result file with an empty parameter (@poje42, #1016)

14.3.5 Miscellaneous

- Add support for python 3.12 (@BenediktBurger, #1051)
- Add support for numpy 2.0 (@CasperSchippers, #1026)
- Add codecov to CI and to readme (@BenediktBurger, #1037, #1052, #1099)
- Add citation file for PyMeasure repository (@mcdo0486, #1092)
- Add release CI (@BenediktBurger, #1039)
- Update readme with permanent Zenodo DOI (@BenediktBurger, #1095)
- Bump CI dependencies to: pyvisa 1.13.0, checkout@v4 (@mcdo0486, #1097)
- Fix/pandas futurewarning (@CasperSchippers, #1062)
- Change copyright year. (@BenediktBurger, #1032)
- Fix typos (@afuetterer, #1003)

14.3.6 New Contributors

@guuskuiper, @OptimisticBeliever, @fwutw, @afuetterer, @poje42, @Rybok, @AlecVercruysse, @ConnorGCarr, @mzen228, @jnnskls, @V0XNIHILI, @pyZerrenner, @JAW90, @driftregion, @jurajjasik, @Konradrundfunk

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.13.1...v0.14.0

14.4 Version 0.13.1 (2023-10-05)

New release to fix ineffective python version restriction in the project metadata (only affected Python<=3.7 environments installing via pip).

14.5 Version 0.13.0 (2023-09-23)

Main items of this new release:

- Dropped support for Python 3.7, added support for Python 3.11.
- Adds a test generator, which observes the communication with an actual device and writes protocol tests accordingly.
- 2 new instrument drivers have been added.

14.5.1 Deprecated features

- Attocube ANC300: The `stepu` and `stepd` properties are deprecated, use the new `move_raw` method instead. (@dkriegner, #938)

14.5.2 Instruments

- Adds a test generator (@bmoneke, #882)
- Adds Thyracont Smartline v2 vacuum sensor transmitter (@bmoneke, #940)
- Adds Thyracont Smartline v1 vacuum gauge (@dkriegner, #937)
- AddsTeledyne base classes with most of *LeCroyT3DSO1204* functionality (@RobertoRoos, #951)
- Fixes instrument documentation (@mcdo0486, #941, #903, @omahs, #960)
- Fixes Toptica Ibeamsmart's __init__ (@waveman68, #959)
- Fixes VISAAdapter flush_read_buffer() (@ileu, #968)
- Updates Keithley2306 and AFG3152C to Channels (@bilderbuchi, #953)

14.5.3 GUI

- Adds console mode (@msmttchr, #500)
- Fixes Dock widget (@msmttchr, #961)

14.5.4 Miscellaneous

- Change CI from conda to mamba (@bmoneke, #947)
- Add support for python 3.11 (@CasperSchippers, #896)

14.5.5 New Contributors

@waveman68, @omahs, @ileu

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.12.0...v0.13.0

14.6 Version 0.12.0 (2023-07-05)

Main items of this new release:

- A `Channel` base class has been added for easier implementation of instruments with channels.
- 19 new instrument drivers have been added.
- Added tests for some commonalities across all instruments.
- We continue to clean up our API in preparation for a future version 1.0. Deprecations and subsequent removals are listed below.

14.6.1 Deprecated features

- HP 34401A: `voltage_ac`, `current_dc`, `current_ac`, `resistance`, `resistance_4w` properties, use `function_` and `reading` properties instead.
- Toptica IBeamSmart: `channel1_enabled`, use `ch_1.enabled` property instead (equivalent for channel2). Also `laser_enabled` is deprecated in favor of `emission` (@bmoneke, #819).
- TelnetAdapter: use `VISAAdapter` instead. VISA supports TCPIP connections. Use the resource_name `TCPIP[board]::<hostname>::<port>::SOCKET` to connect to a server (@Max-Herbold, #835).
- Attocube ANC300: `host` argument, pass a resource string or adapter as `Adapter` passed to `Instrument`. Now communicates through the `VISAAdapter` rather than deprecated `TelnetAdapter`. The initializer now accepts `name` as its second keyword argument so all previous initialization positional arguments (*axisnames*, *passwd*, *query_delay*) should be switched to keyword arguments.
- The property creators `control`, `measurement`, and `setting` do not accept arbitrary keyword arguments anymore. Use the `v_kwargs` parameter for arguments you want to pass on to `values` method, instead.
- The property creators `control`, `measurement`, and `setting` do not accept *command_process* anymore. Use a dynamic property or a *Channel* instead, as appropriate (@bmoneke, #878).
- See also the next section.

14.6.2 New adapter and instrument mechanics

- All instrument constructors are required to accept a `name` argument.
- Changed: `read_bytes` of all Adapters by default does not stop reading on a termination character, unless the new argument `break_on_termchar` is set to *True*.
- Channel class added. `Instrument.channels` and `Instrument.ch_X` (`X` is any channel name) are reserved attributes for channel mechanics.
- The parameters `check_get_errors` and `check_set_errors` enable calling methods of the same name. This enables more systematically dealing with instruments that acknowledge every "set" command.
- Adds Channel feature to instruments (@bmoneke, mcdo0486, #718, #761, #852, #931)
- Adds `maxsplit` parameter to `values` method (@bmoneke, #793)
- Adds (deprecated) global preprocess reply for backward compatibility (@bmoneke, #876)
- Adds fallback version for discarding the read buffer to VISAAdapter (@dkriegner, #836)
- Adds `flush_read_buffer` to SerialAdapter (@RobertoRoos, #865)
- Adds `gpib_read_timeout` to PrologixAdapter (@neuschs, #927)
- Adds command line option to pass resource address for instrument tests (@bleykauf, #789)
- Adds "find all instruments" and channels for testing (@bmoneke, #909, @mcdo0486, #911, #912)
- Adds test that an instrument hands kwargs to the adapter (@bmoneke, #814)
- Adds property docstring check (@bmoneke, #895)
- Improves property factories' docstrings (@bmoneke, #843)
- Improves property factories: do not allow undefined kwargs (@bmoneke, #856)
- Improves property factories: check_set/get_errors argument to call methods of the same name (@bmoneke, #883)

- Improves `read_bytes` of Adapter (@bmoneke, #839)
- Improves the ProtocolAdapter with a mock connection (@bmoneke, #782), and enable it to have empty messages in the protocol (@bmoneke, #818)
- Improves Prologix adapter documentation (@bmoneke, #813) and configurable settings (@bmoneke, #845)
- Improves behavior of `read_bytes(-1)` for `SerialAdapter` (@RobertoRoos, #866)
- Improves all instruments with name kwarg (@bmoneke, #877)
- Improves VisaAdapter: close manager only when using pyvisa-sim (@dkriegner, #900)
- Harmonises instrument name definition pattern, consistently name the instrument connection argument "adapter" (@bmoneke, #659)
- Fixes ProtocolAdapter has list in signature (@bmoneke, #901)
- Fixes VISAAdapter's `read_bytes` (@bmoneke, #867)
- Fixes query_delay usage in VISAAdapter (@bmoneke, #765)
- Fixes VisaAdapter: close resource manager only when using pyvisa-sim (@dkriegner, #900)

14.6.3 Instruments

- New Advantest R624X DC Voltage/Current Sources/Monitors (@wichers, #802)
- New AJA International DC sputtering power supply (@dkriegner, #778)
- New Anritus MS2090A (@aruznieto, #787)
- New Anritsu MS4644B (@CasperSchippers, #827)
- New DSP 7225 and new DSPBase instrument (@mcdo0486, #902)
- New HP 8560A / 8561B Spectrum Analyzer (@neuschs, #888)
- New IPG Photonics YAR Amplifier series (@bmoneke, #851)
- New Keysight E36312A power supply (@scandey, #785)
- New Keithley 2200 power supply (@ashokbruno, #806)
- New Lake Shore 211 Temperature Monitor (@mcdo0486, #889)
- New Lake Shore 224 and improves Lakeshore instruments (@samcondon4, #870)
- New MKS Instruments 937B vacuum gauge controller (@dkriegner, @bilderbuchi, #637, #772, #936)
- New Novanta FPU60 laser power supply unit (@bmoneke, #885)
- New TDK Lambda Genesys 80-65 DC and 40-38 DC power supplies (@mcdo0486, 906)
- New Teledyne T3AFG waveform generator instrument (@scandey, #791)
- New Teledyne (LeCroy) T3DSO1204 Oscilloscope (@LastStartDust, #697, @bilderbuchi, #770)
- New T&C Power Conversion RF power supply (@dkriegner, #800)
- New Velleman K8090 relay device (@RobertoRoos, #859)
- Improves Agilent 33500 with the new channel feature (@JCarl-OS, #763, #773)
- Improves HP 3478A with calibration data related functions (@tomverbeure, #777)
- Improves HP 34401A (@CodingMarco, #810)
- Improves the Oxford instruments with the new channel feature (@bmoneke, #844)

- Improves Siglent SPDxxxxX with the new channel feature (@AidenDawn 758)
- Improves Teledyne T3DSO1204 device tests (@LastStarDust, #841)
- Fixes Ametek DSP 7270 lockin amplifier issues (@seb5g, #897)
- Fixes DSP 7265 erroneously using preprocess_reply (@mcdo0486, #873)
- Fixes print statement in DSPBase.sensitivity (@mcdo0486, #915)
- Fixes Fluke bath commands (@bmoneke, #874)
- Fixes a frequency limitation in HP 8657B (@LongnoseRob, #769)
- Fixes Keithley 2600 channel calling parent's shutdown (@mcdo0486, #795)

14.6.4 Automation

- Adds tolerance for opening result files with missing parameters (@msmttchr, #780)
- Validate DATA_COLUMNS entries earlier, avoid exceptions in a running procedure (@mcdo0486, #796, #934)

14.6.5 GUI

- Adds docking windows (@mcdo0486, #722, #762)
- Adds save plot settings in addition to dock layout (@mcdo0486, #850)
- Adds log widget colouring and format option (@CasperSchippers, #890)
- Adds table widget (@msmttchr, #771)
- New sequencer architecture: decouples it from the graphical tree, adapts it for further expansions (@msmttchr, #518)
- Moves coordinates label to the pyqtgraph PlotItem (@CasperSchippers, #822)
- Fixes crashing ImageWidget at new measurement (@CasperSchippers, #790)
- Fixes checkboxes not working for groups in inputs-widget (@CasperSchippers, #794)

14.6.6 Miscellaneous

- Adds a collection of solutions for instrument implementation challenges (@bmoneke, #853, #861)
- Updates Tutorials/Making_a_measurement/ example_codes (@sansanda, #749)

14.6.7 New Contributors

@JCarl-OS, @aruznieto, @scandey, @tomverbeure, @wichers, @Max-Herbold, @RobertoRoos

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.11.1...v0.12.0

14.7 Version 0.11.1 (2022-12-31)

14.7.1 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

14.7.2 Dependency updates

- Required version of PyQtGraph is increased from `pyqtgraph >= 0.9.10` to `pyqtgraph >= 0.12` to support new PyMeasure display widgets.

14.7.3 GUI

- Added ManagedDockWindow to allow multiple dockable plots (@mcdo0486, @CasperSchippers, #722)
- Move coordinates label to the pyqtgraph PlotItem (@CasperSchippers, #822)
- New sequencer architecture (@msmttchr, @CasperSchippers, @mcdo0486, #518)
- Added "Save Dock Layout" functionality to DockWidget context menu. (@mcdo0486, #762)

14.8 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.

14.8.1 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.

14.8.2 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.

14.8.3 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)

14.8.4 Automation

- Add storing unitful experiment results (@bmoneke, #642)
- Add storing conditions in file (@CasperSchippers, #503)

14.8.5 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)

14.8.6 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.

14.8.7 New Contributors

@jcarbelbide, @mroeleke, @bmoneke, @garzetti, @AsafYagoda, @AidenDawn, @LastStarDust, @sansanda

Full Changelog: https://github.com/pymeasure/pymeasure/compare/v0.10.0...v0.11.0

14.9 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

14.9.1 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)

14.9.2 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)

14.9.3 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)

14.9.4 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

14.10 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)

14.11 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)

14.12 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)

14.13 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)

14.14 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)

14.15 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)

14.16 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)

14.17 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

14.18 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)

14.19 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)

14.20 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

14.21 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

14.22 Version 0.4.1 – released 7/31/16

- Critical fix in setup.py for importing instruments (also added to documentation)

14.23 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

14.24 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

14.25 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

14.26 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

14.27 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

14.28 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

14.29 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

14.30 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

14.31 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

14.32 Version 0.1.0 – released 7/15/14

- Initial release

CHAPTER

FIFTEEN

RELATED PROJECTS

PyMeasure is part of a broader ecosystem of laboratory automation tools:

- LECO Protocol: Open standard for laboratory equipment communication using modern protocols
- pyleco: Python library implementing LECO for distributed instrument control systems
- Python Lab Automation Landscape: Curated collection of Python tools for laboratory automation and instrumentation

These projects provide complementary capabilities to PyMeasure for building robust laboratory automation systems.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

D

E

F

G

H

I

J

K

M

O

P

Q

R

S

T

U

V

W

X

Y

Z