

---

# **PyMeasure Documentation**

***Release 0.1.dev577+ga064eef.d20231006***

**PyMeasure Developers**

**Oct 06, 2023**



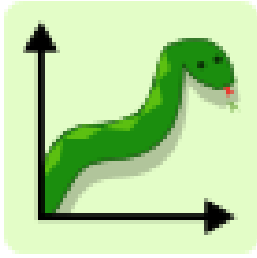
# LEARNING PYMEASURE

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Instrument ready . . . . .	3
1.2	Graphical displays . . . . .	3
<b>2</b>	<b>Quick start</b>	<b>5</b>
2.1	Setting up Python . . . . .	5
2.2	Installing PyMeasure . . . . .	5
<b>3</b>	<b>Tutorials</b>	<b>7</b>
3.1	Connecting to an instrument . . . . .	7
3.2	Making a measurement . . . . .	9
3.3	Using a graphical interface . . . . .	18
<b>4</b>	<b>pymeasure.adapters</b>	<b>47</b>
4.1	Adapter base class . . . . .	47
4.2	VISA adapter . . . . .	49
4.3	Serial adapter . . . . .	53
4.4	Prologix adapter . . . . .	56
4.5	VXI-11 adapter . . . . .	61
4.6	Telnet adapter . . . . .	63
4.7	Test adapters . . . . .	66
<b>5</b>	<b>pymeasure.experiment</b>	<b>71</b>
5.1	Experiment class . . . . .	71
5.2	Listener class . . . . .	72
5.3	Procedure class . . . . .	73
5.4	Parameter classes . . . . .	74
5.5	Worker class . . . . .	79
5.6	Results class . . . . .	79
<b>6</b>	<b>pymeasure.display</b>	<b>83</b>
6.1	Browser classes . . . . .	83
6.2	Console class . . . . .	83
6.3	Curves classes . . . . .	84
6.4	Inputs classes . . . . .	85
6.5	Listeners classes . . . . .	87
6.6	Log classes . . . . .	87
6.7	Manager classes . . . . .	87
6.8	Plotter class . . . . .	89
6.9	Qt classes . . . . .	89
6.10	Thread classes . . . . .	89

6.11	Widget classes . . . . .	90
6.12	Windows classes . . . . .	98
<b>7</b>	<b>pymeasure.instruments</b>	<b>103</b>
7.1	Instrument classes . . . . .	103
7.2	Validator functions . . . . .	114
7.3	Comedi data acquisition . . . . .	116
7.4	Resource Manager . . . . .	116
7.5	Active Technologies . . . . .	116
7.6	Advantest . . . . .	122
7.7	Agilent . . . . .	152
7.8	AJA International . . . . .	196
7.9	Ametek . . . . .	198
7.10	AMI . . . . .	200
7.11	Anaheim Automation . . . . .	202
7.12	Anapico . . . . .	204
7.13	Andeen Hagerling . . . . .	205
7.14	Anritsu . . . . .	208
7.15	Attocube . . . . .	223
7.16	BK Precision . . . . .	226
7.17	Danfysik . . . . .	227
7.18	Delta Elektronika . . . . .	230
7.19	Edwards . . . . .	231
7.20	EURO TEST . . . . .	231
7.21	Fluke . . . . .	234
7.22	F.W. Bell . . . . .	234
7.23	Heidenhain . . . . .	237
7.24	HC Photonics . . . . .	238
7.25	Hewlett Packard . . . . .	239
7.26	IPG Photonics . . . . .	286
7.27	Keithley . . . . .	287
7.28	Keysight . . . . .	339
7.29	Lake Shore Cryogenics . . . . .	351
7.30	LeCroy . . . . .	360
7.31	MKS Instruments . . . . .	375
7.32	Newport . . . . .	377
7.33	National Instruments . . . . .	378
7.34	Novanta Photonics . . . . .	392
7.35	Oxford Instruments . . . . .	393
7.36	Parker . . . . .	403
7.37	Pendulum . . . . .	404
7.38	Razorbill . . . . .	405
7.39	Rohde & Schwarz . . . . .	406
7.40	Siglent Technologies . . . . .	423
7.41	Signal Recovery . . . . .	427
7.42	Stanford Research Systems . . . . .	440
7.43	T&C Power Conversion . . . . .	452
7.44	TDK Lambda . . . . .	456
7.45	Tektronix . . . . .	464
7.46	Teledyne . . . . .	465
7.47	Temptronic . . . . .	476
7.48	TEXIO . . . . .	484
7.49	Thermotron . . . . .	487
7.50	Thorlabs . . . . .	489

7.51	Thyracont . . . . .	490
7.52	Toptica . . . . .	495
7.53	Velleman . . . . .	497
7.54	Yokogawa . . . . .	499
<b>8</b>	<b>Contributing</b>	<b>503</b>
8.1	Using the development version . . . . .	503
8.2	Working on a new feature . . . . .	504
8.3	Making a pull request . . . . .	504
8.4	Unit testing . . . . .	505
<b>9</b>	<b>Reporting an error</b>	<b>507</b>
<b>10</b>	<b>Adding instruments</b>	<b>509</b>
10.1	File structure . . . . .	509
10.2	Instrument file . . . . .	510
10.3	Your instrument's user interface . . . . .	511
10.4	Defining default connection settings . . . . .	513
10.5	Writing properties . . . . .	515
10.6	Instruments with similar features . . . . .	522
10.7	Instruments with channels . . . . .	524
10.8	Advanced communication protocols . . . . .	527
10.9	Writing tests . . . . .	529
10.10	Solutions for implementation challenges . . . . .	533
<b>11</b>	<b>Coding Standards</b>	<b>535</b>
11.1	Python style guides . . . . .	535
11.2	Documentation . . . . .	535
11.3	Usage of getter and setter functions . . . . .	536
11.4	Docstrings . . . . .	536
<b>12</b>	<b>Authors</b>	<b>537</b>
<b>13</b>	<b>License</b>	<b>539</b>
<b>14</b>	<b>Changelog</b>	<b>541</b>
14.1	Version 0.13.1 (2023-10-05) . . . . .	541
14.2	Version 0.13.0 (2023-09-23) . . . . .	541
14.3	Version 0.12.0 (2023-07-05) . . . . .	542
14.4	Version 0.11.1 (2022-12-31) . . . . .	545
14.5	Version 0.11.0 (2022-11-19) . . . . .	546
14.6	Version 0.10.0 (2022-04-09) . . . . .	548
14.7	Version 0.9 – released 2/7/21 . . . . .	551
14.8	Version 0.8 – released 3/29/19 . . . . .	552
14.9	Version 0.7 – released 8/4/19 . . . . .	552
14.10	Version 0.6.1 – released 4/21/19 . . . . .	553
14.11	Version 0.6 – released 1/14/19 . . . . .	553
14.12	Version 0.5.1 – released 4/14/18 . . . . .	553
14.13	Version 0.5 – released 10/18/17 . . . . .	553
14.14	Version 0.4.6 – released 8/12/17 . . . . .	554
14.15	Version 0.4.5 – released 7/4/17 . . . . .	554
14.16	Version 0.4.4 – released 6/4/17 . . . . .	554
14.17	Version 0.4.3 – released 3/30/17 . . . . .	554
14.18	Version 0.4.2 – released 8/23/16 . . . . .	555
14.19	Version 0.4.1 – released 7/31/16 . . . . .	555

14.20	Version 0.4 – released 7/29/16 . . . . .	555
14.21	Version 0.3 – released 4/8/16 . . . . .	555
14.22	Version 0.2 – released 12/16/15 . . . . .	556
14.23	Version 0.1.6 – released 4/19/15 . . . . .	556
14.24	Version 0.1.5 – release 10/22/14 . . . . .	556
14.25	Version 0.1.4 – released 8/2/14 . . . . .	556
14.26	Version 0.1.3 – released 7/20/14 . . . . .	556
14.27	Version 0.1.2 – released 7/18/14 . . . . .	557
14.28	Version 0.1.1 – released 7/16/14 . . . . .	557
14.29	Version 0.1.0 – released 7/15/14 . . . . .	557
<b>Python Module Index</b>		<b>559</b>
<b>Index</b>		<b>561</b>



# 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](#)



## 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:

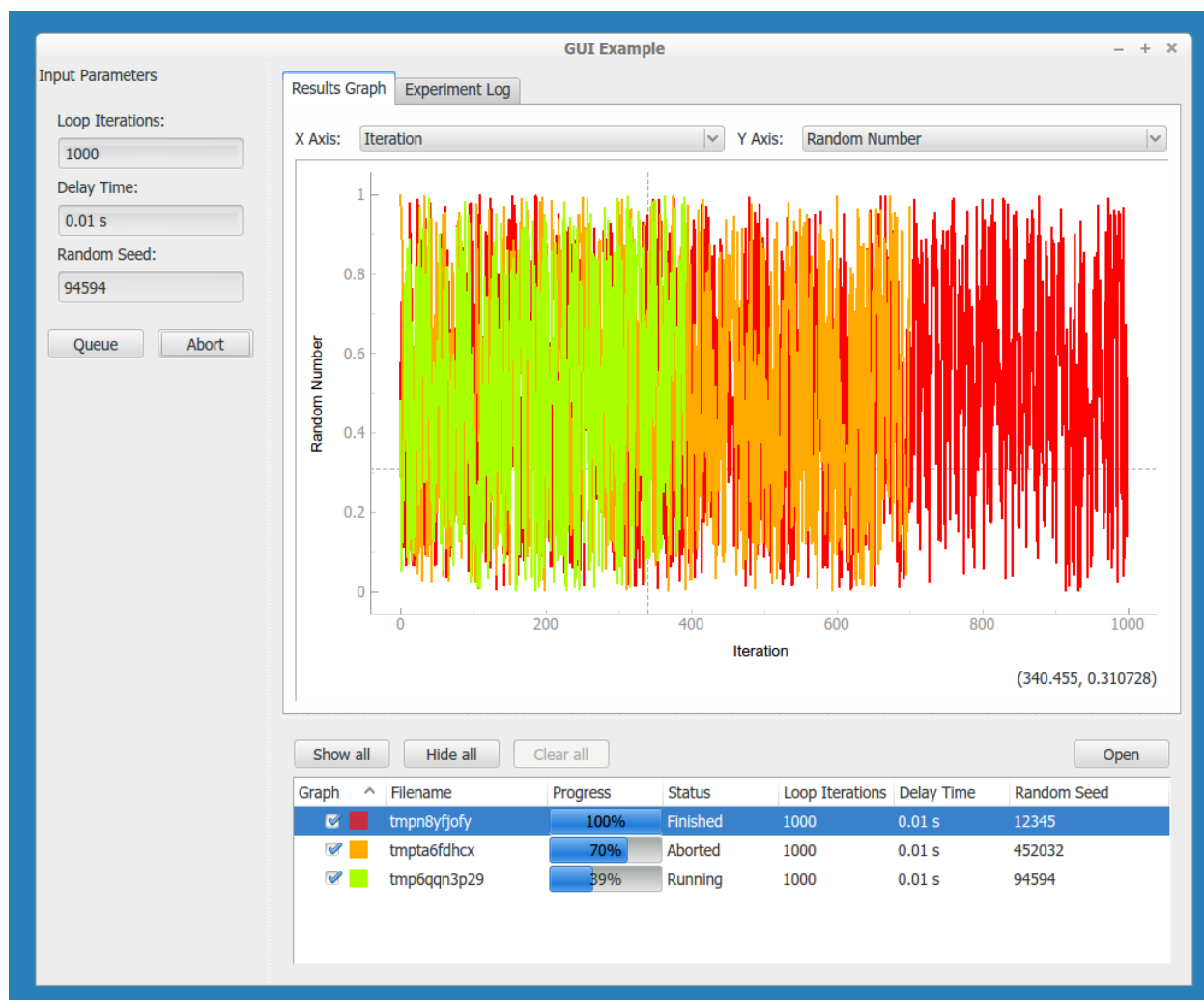
```
from pymeasure.instruments.keithley import Keithley2400
```

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.

## 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*?

## TUTORIALS

The following sections provide instructions for getting started with PyMeasure.

### 3.1 Connecting to an instrument

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

First import the instrument of interest.

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

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

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

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

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

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

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

```
sourcemeter.id
```

This is equivalent to manually calling the SCPI command.

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

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

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

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

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

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

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

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

### 3.1.1 Using adapters

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

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

```
from pymeasure.adapters import VISAAdapter

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

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

```
from pymeasure.adapters import PrologixAdapter

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

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

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

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

### 3.1.2 Modifying connection settings

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

When using a string or integer to connect to an instrument, a *VISAAdapter* is used internally. Additional settings need to be passed in as keyword arguments. For example, to use a fast baud rate on a quick connection when connecting to the Keithley2400 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 an undetermined state
- Exceptions also end the system in an undetermined state

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

### 3.2.2 Using Procedures

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

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

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

class SimpleProcedure(Procedure):

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

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

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

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

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

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

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

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

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

Next we will show how to run the procedure.

## Running Procedures

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

We first construct a Results object for our Procedure.

```
from pymeasure.experiment import Results

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

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

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

```
from pymeasure.experiment import Worker

worker = Worker(results)
```

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

Now we are ready to start the worker.

```
worker.start()
```

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

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

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

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

class SimpleProcedure(Procedure):

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

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

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

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

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

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

    worker = Worker(results)
    worker.start()

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

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

## Using Logs

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

Let's extend our SimpleProcedure with logging.

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

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

class SimpleProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')

    DATA_COLUMNS = ['Iteration']

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

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

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

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

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

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

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

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

## Storing metadata

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

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

class SimpleProcedure(Procedure):

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

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

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

    def startup(self):
        self.custom_metadata = 20

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

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

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

## Modifying our script

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

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

from pymeasure.log import log, console_log

class IVProcedure(Procedure):

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

```

The parentheses in the COLUMN entries indicate the physical unit of the data in the corresponding column, e.g. 'Voltage Std (V)' indicates Volts. If you want to indicate a dimensionless value, e.g. Mach number, you can use (*I*) 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()
            }
```

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

```

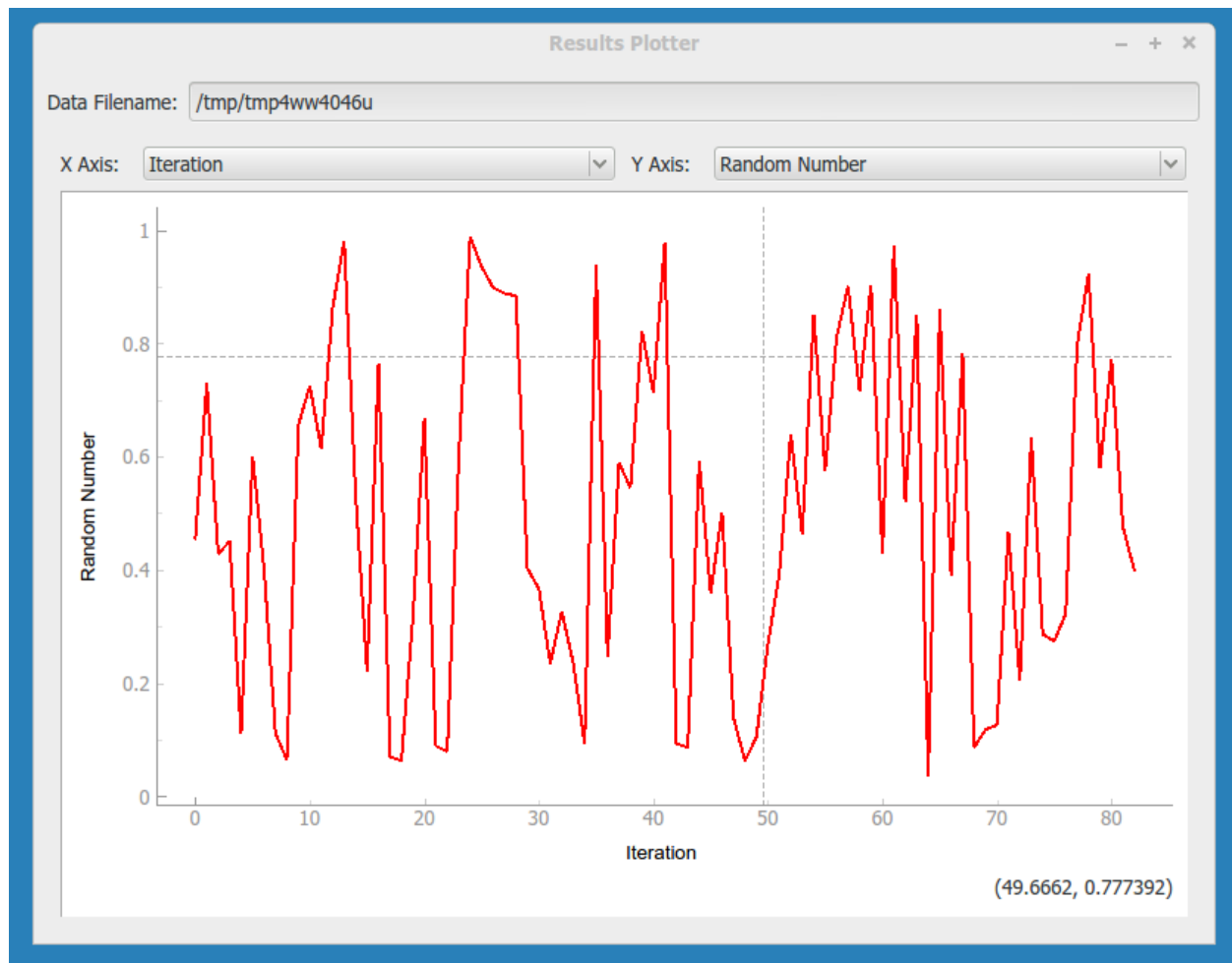
The important addition is the construction of the Plotter from the Results object.

```

plotter = Plotter(results)
plotter.start()

```

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



### 3.3.2 Using the ManagedWindow

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

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

Below we adapt our previous example to use a ManagedWindow.

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

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

(continues on next page)

(continued from previous page)

```

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

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations', 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')

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

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

```

(continues on next page)

(continued from previous page)

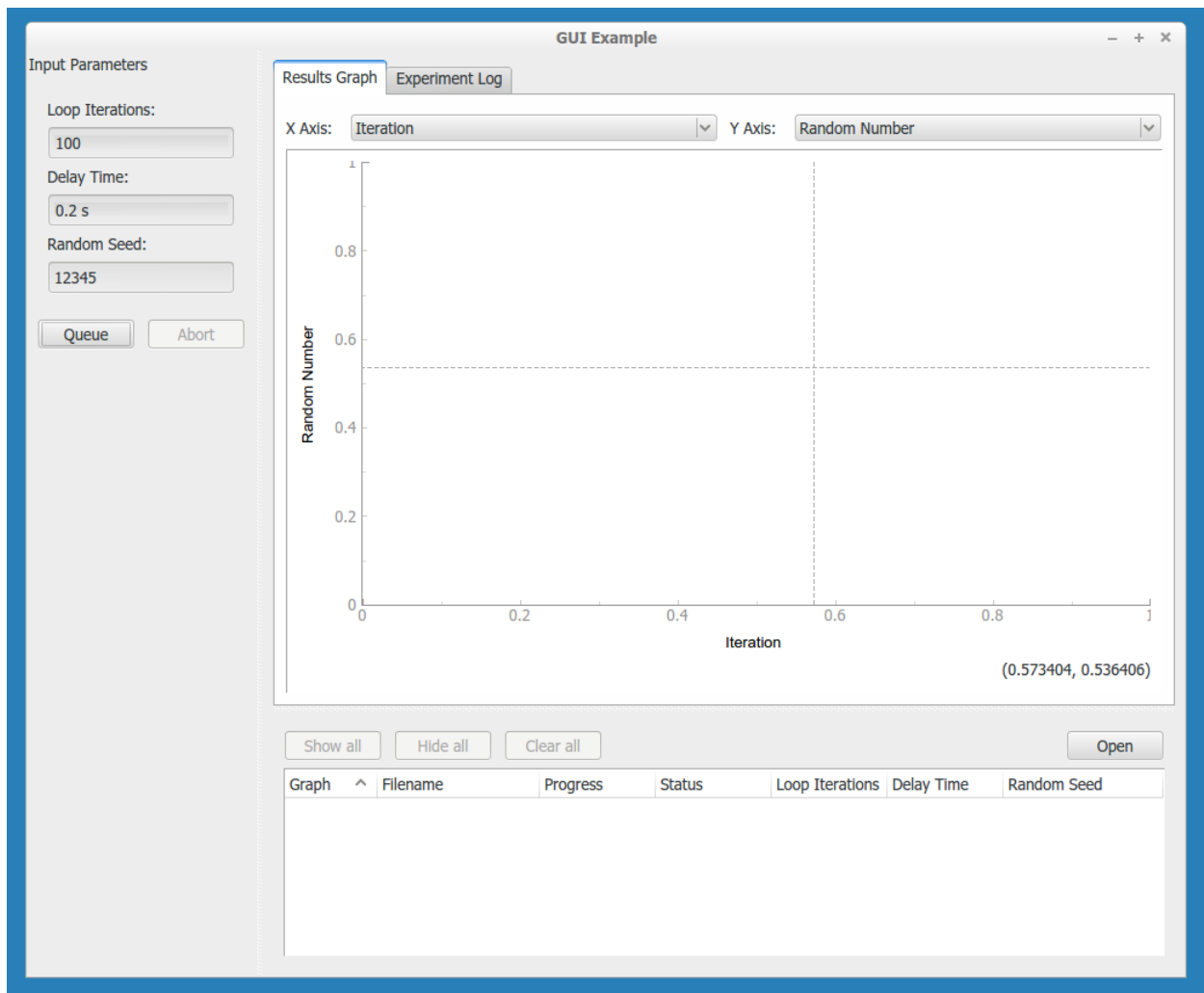
```

self.manager.queue(experiment)

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

```

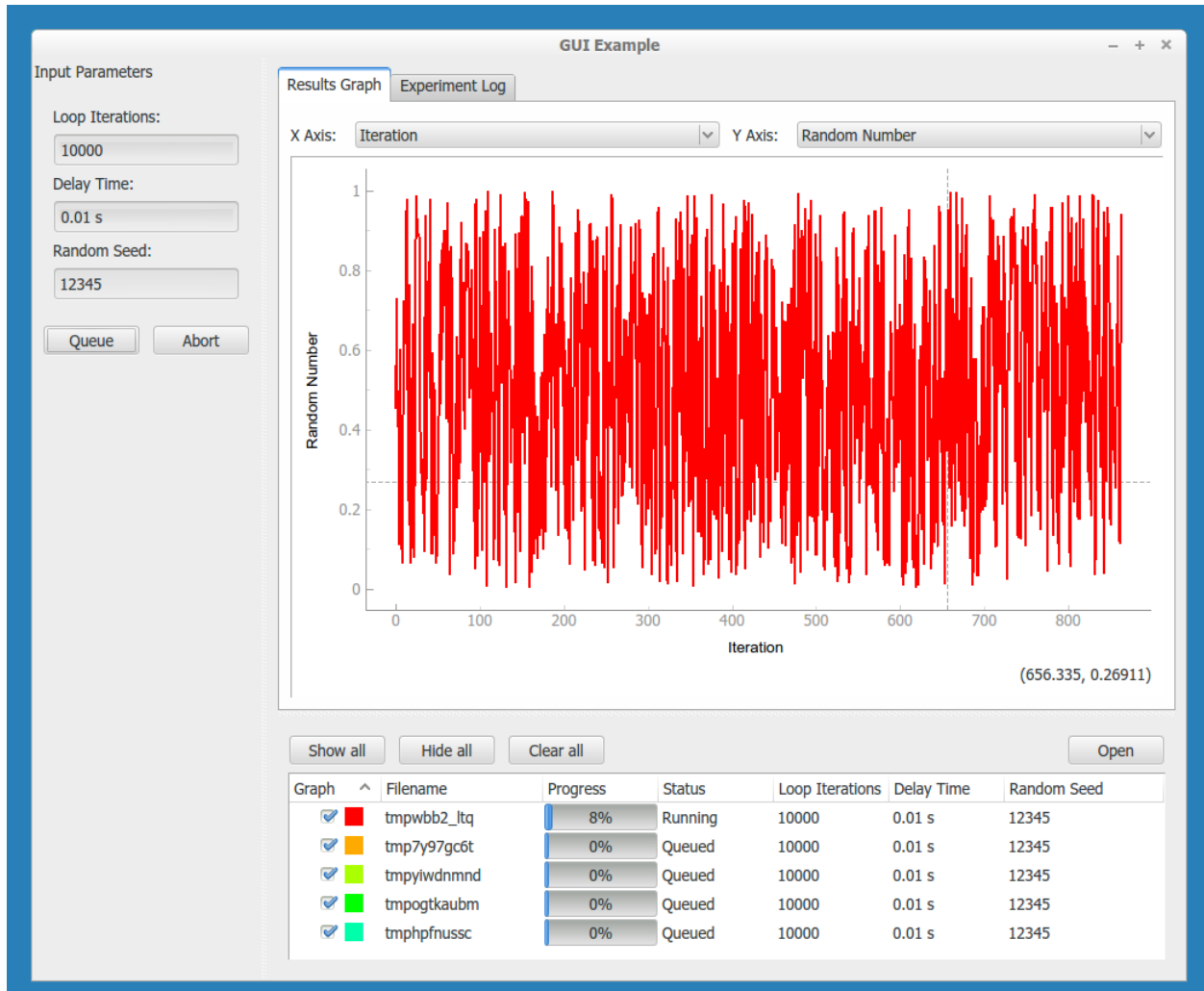
This results in the following graphical display.



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

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

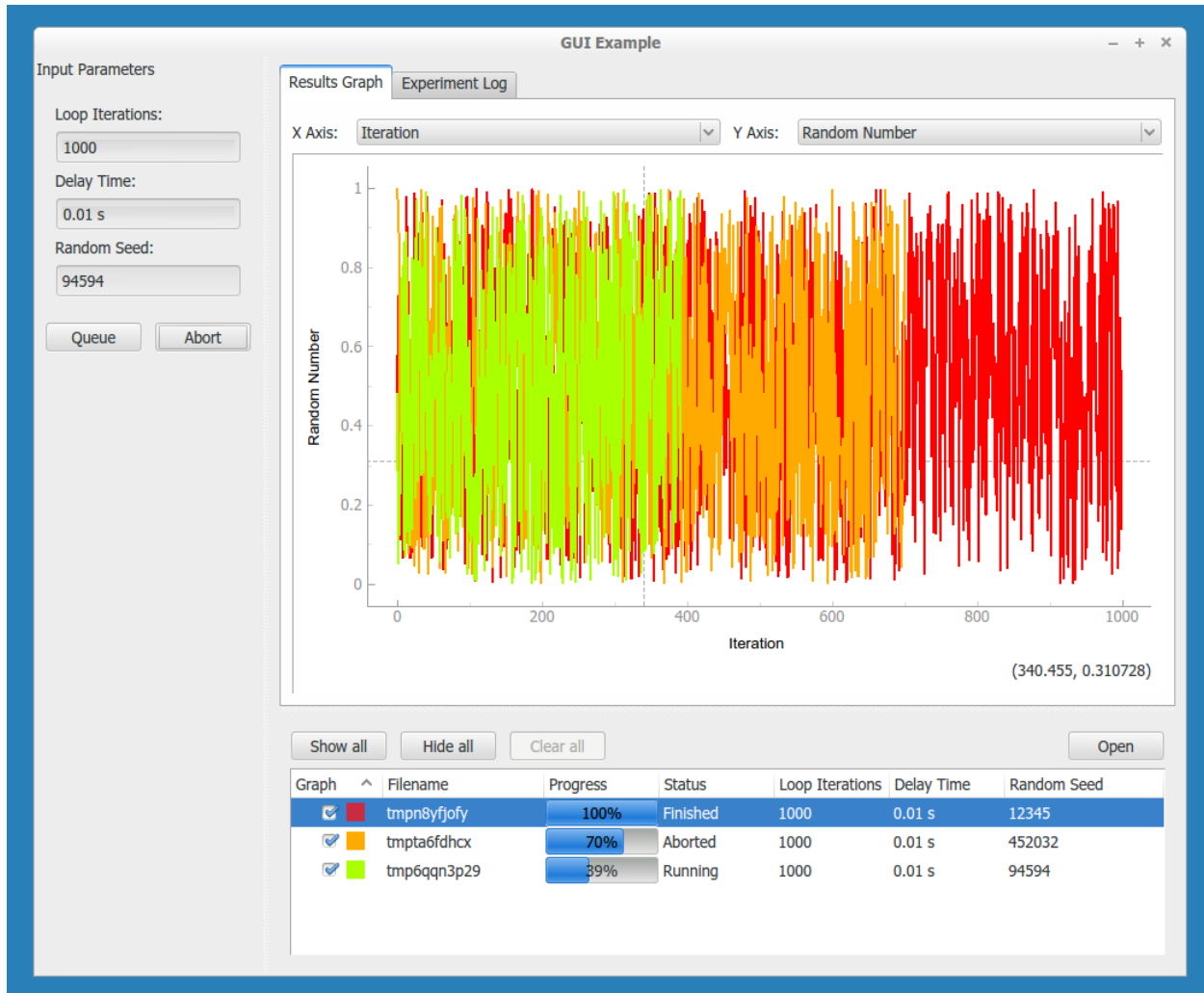
on the queue. In this context, the Manager uses an Experiment object to keep track of the Procedure, Results, and its associated graphical representations in the browser and live-graph. This is then given to the Manager to queue the experiment.



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



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



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

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

### 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']
```

(continues on next page)

(continued from previous page)

```

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):

    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')

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

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

        self.manager.queue(experiment)

if __name__ == "__main__":

```

(continues on next page)

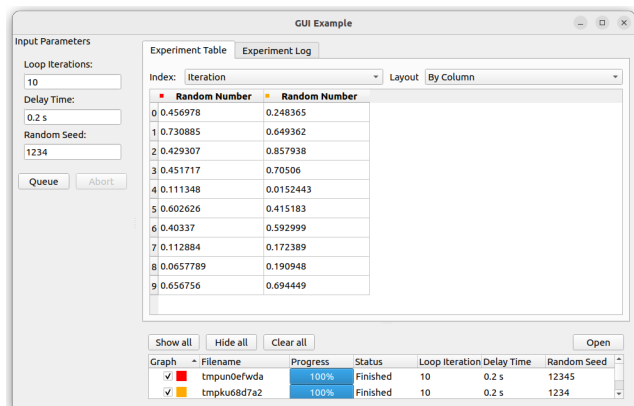
(continued from previous page)

```

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')

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

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

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

        self.manager.queue(experiment)
```

This adds the sequencer underneath the input panel.

Input Parameters

Loop Iterations:

Delay Time:

Random Seed:

Sequencer

Level	Parameter	Sequence
0	Delay Time	arange(0.25, 1, 0.25)
1	Random Seed	[1, 4, 8]
2	Loop Iterations	exp(linspace(1, 5, 3))
1	Random Seed	arange(10, 100, 10)

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 directory input

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

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

```
class MainWindow(ManagedWindow):

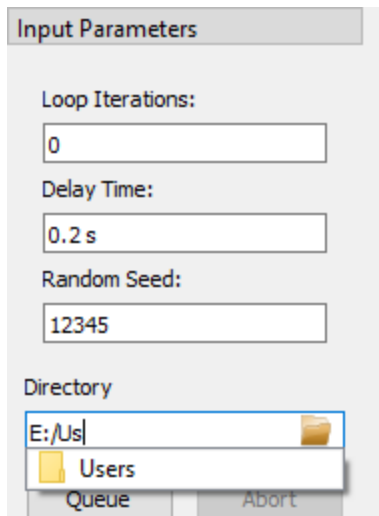
    def __init__(self):
        super().__init__(
            procedure_class=TestProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number',
            directory_input=True,    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

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

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

        self.manager.queue(experiment)
```

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



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

### 3.3.8 Using the estimator widget

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

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

```
class RandomProcedure(Procedure):  
  
    # ...  
  
    def get_estimates(self, sequence_length=None, sequence=None):  
  
        return self.iterations * self.delay
```

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

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

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

```
class RandomProcedure(Procedure):  
  
    # ...  
  
    def get_estimates(self, sequence_length=None, sequence=None):
```

(continues on next page)

(continued from previous page)

```

duration = self.iterations * self.delay

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

return estimates

```

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

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

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

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

### 3.3.9 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`).

### 3.3.10 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())
```

(continues on next page)

(continued from previous page)

```

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

class RandomProcedure(Procedure):

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

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

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

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

class MainWindow(ManagedDockWindow):

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

    def queue(self):

```

(continues on next page)

(continued from previous page)

```

filename = tempfile.mktemp()

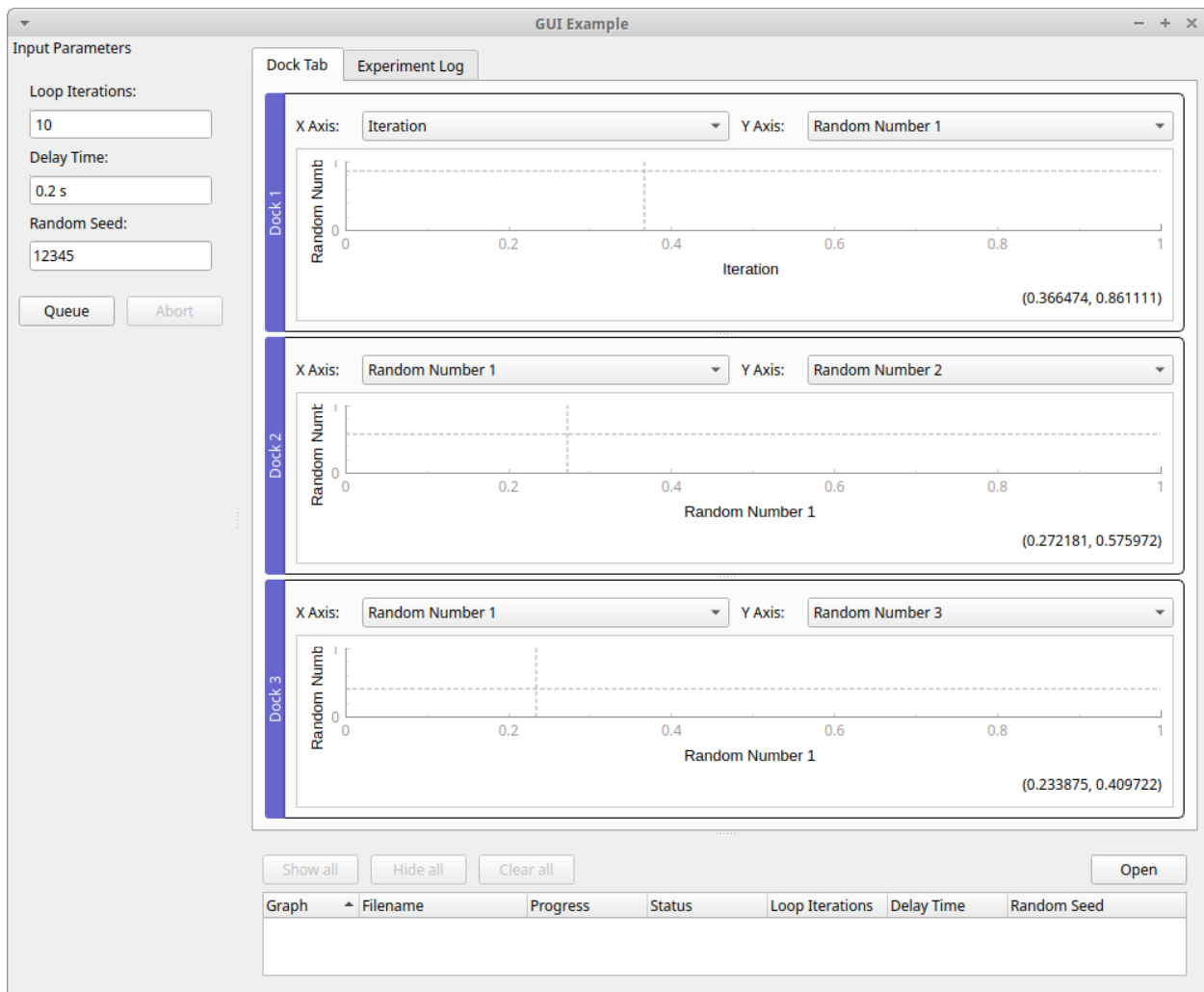
procedure = self.make_procedure()
results = Results(procedure, filename)
experiment = self.new_experiment(results)

self.manager.queue(experiment)

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

```

Now we can see our ManagedDockWindow:



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

1. **X Axis:** “Iteration” **Y Axis:** “Random Number 1”

2. **X Axis:** “Random Number 1” **Y Axis:** “Random Number 2”

3. **X Axis:** “Random Number 1” **Y Axis:** “Random Number 3”

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

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

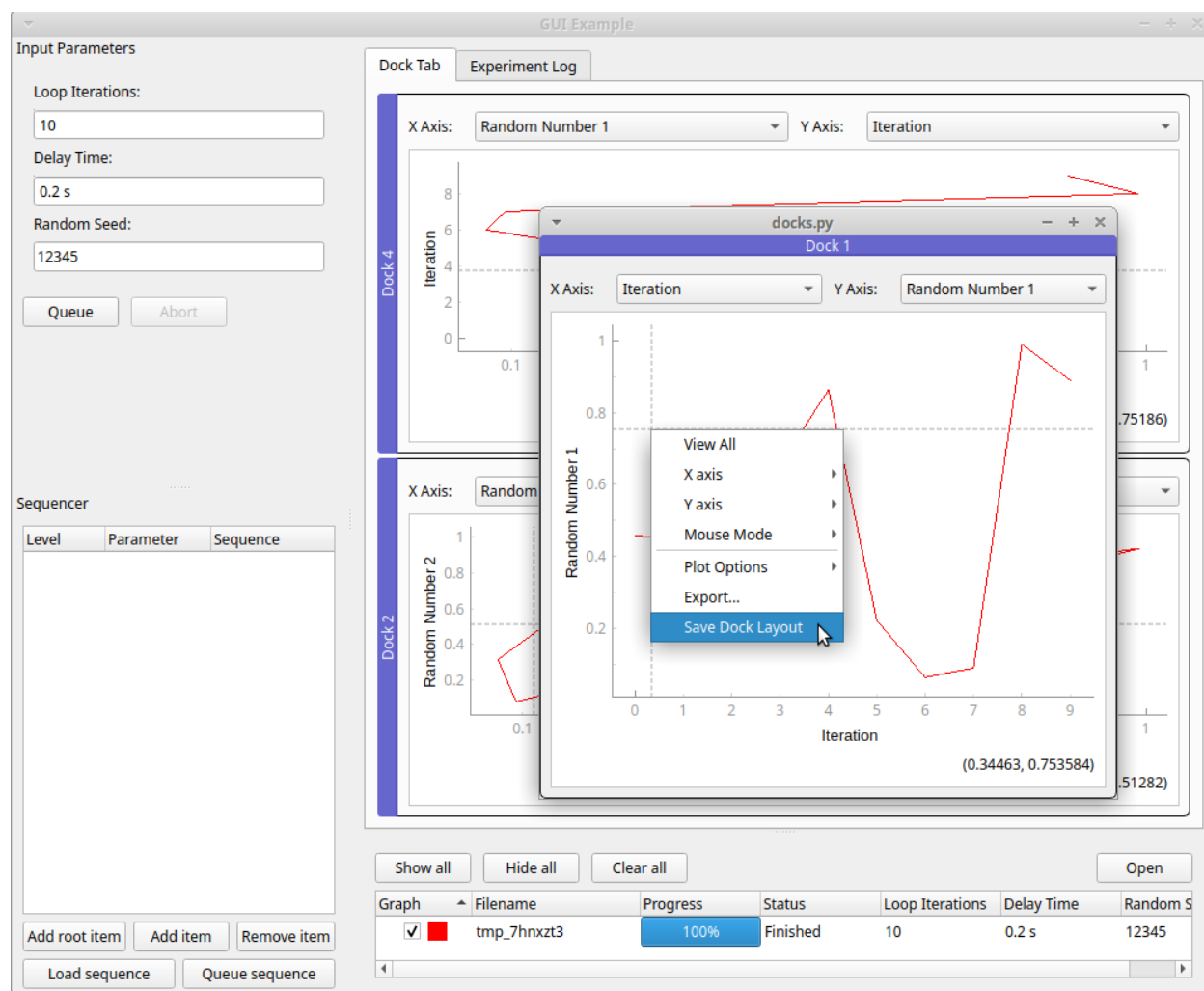
If one list is longer than the last element if the other list is used as the default label for the rest of the dockable plots. In our example that is why we have two **X Axis** labels with “Random Number 1”. The longest list between `x_axis` and `y_axis` determines the number of plots. In our example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

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

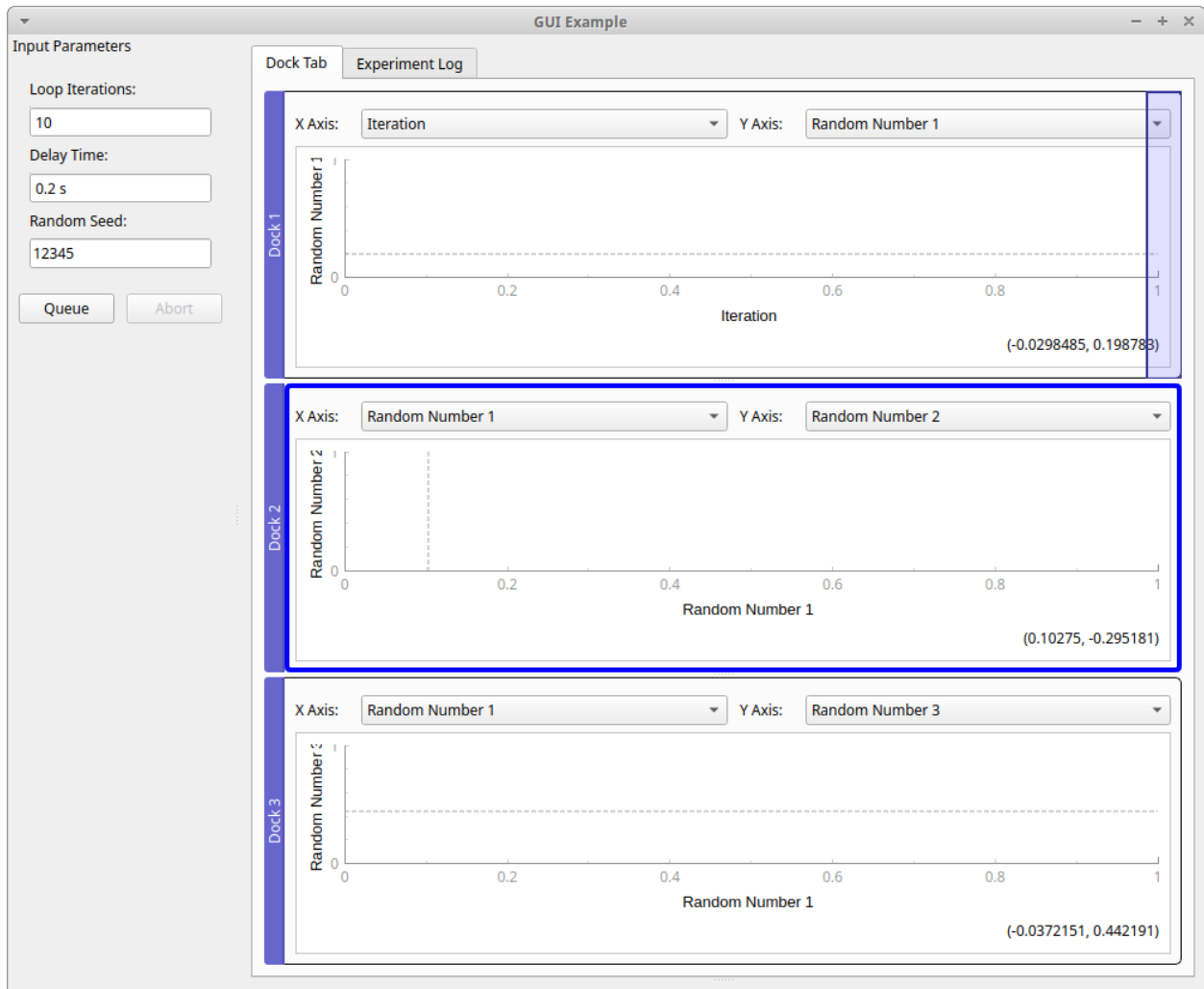
You can return the popped out window to the main window by clicking the close icon X in the top right.

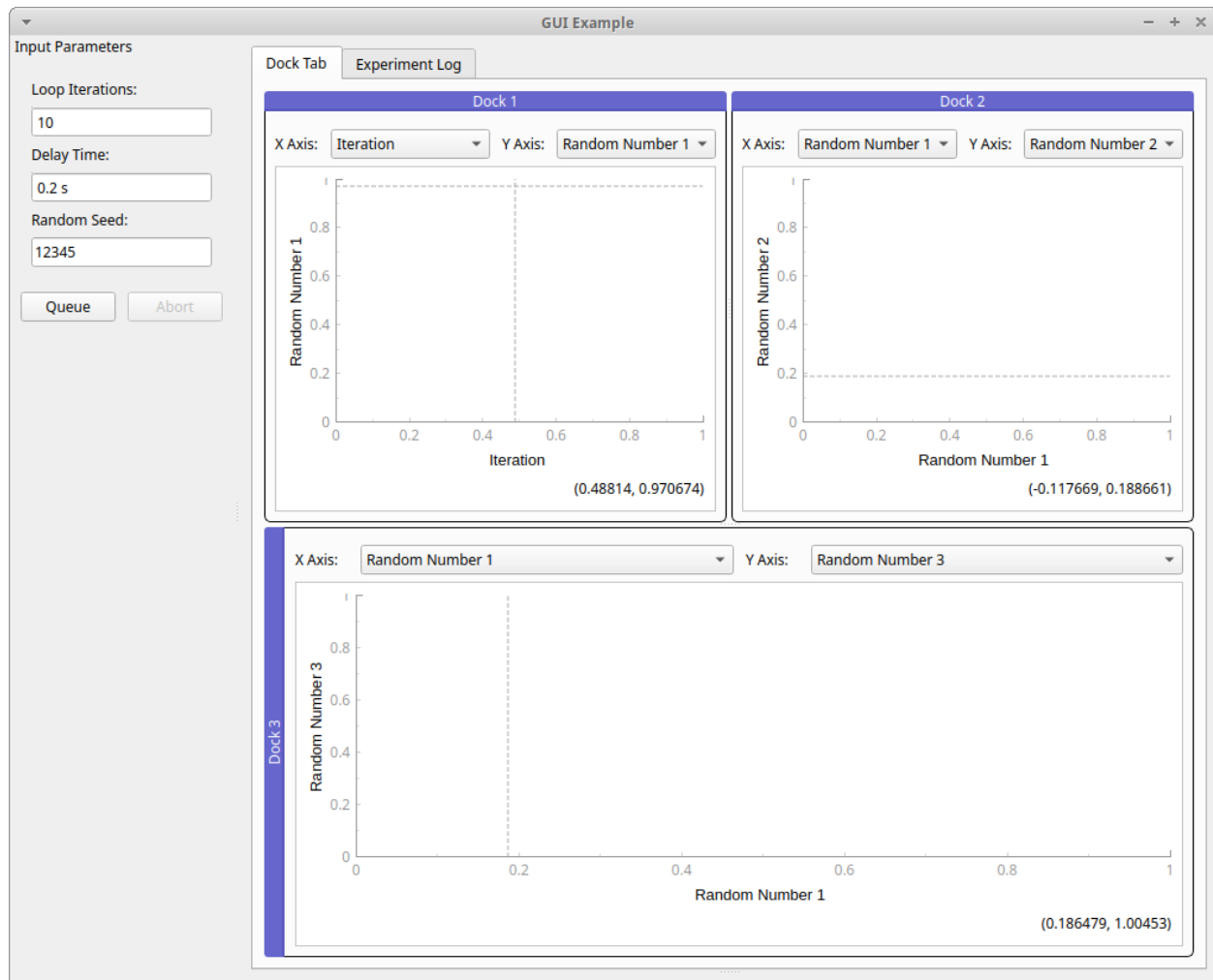
After positioning your dock windows, you can save the layout by right-clicking a dock widget and select “Save Dock Layout” from the context menu. This will save the layout of all docks and the settings for each plot to a file. By default the file path is the current working directory of the python file 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.

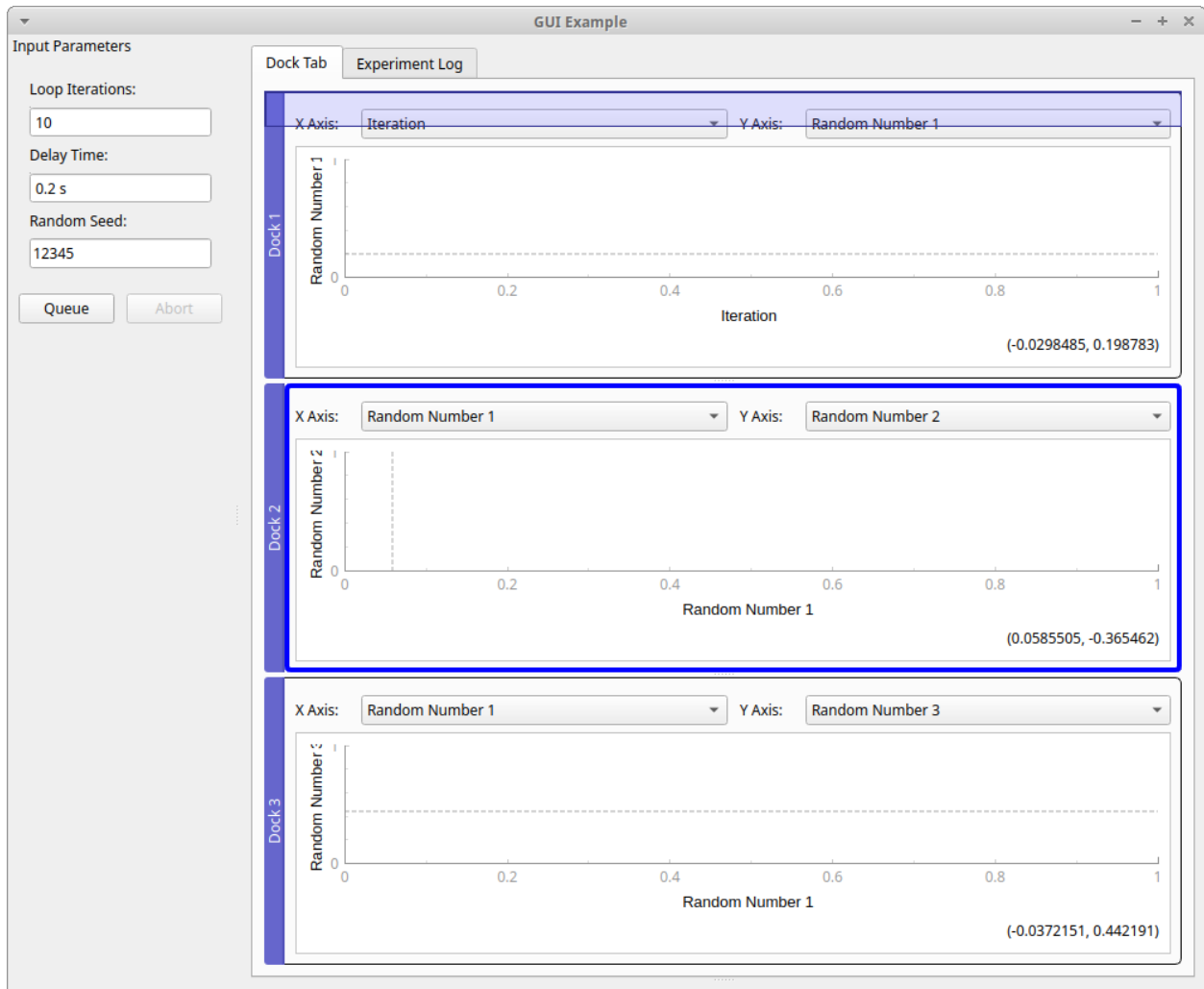


You can drag a dockable plot to reposition it in reference to other plots in the main dock window in several ways. You can drag the blue “Dock #” title bar to the left or right side of another plot to reposition a plot to be side by side with another plot:

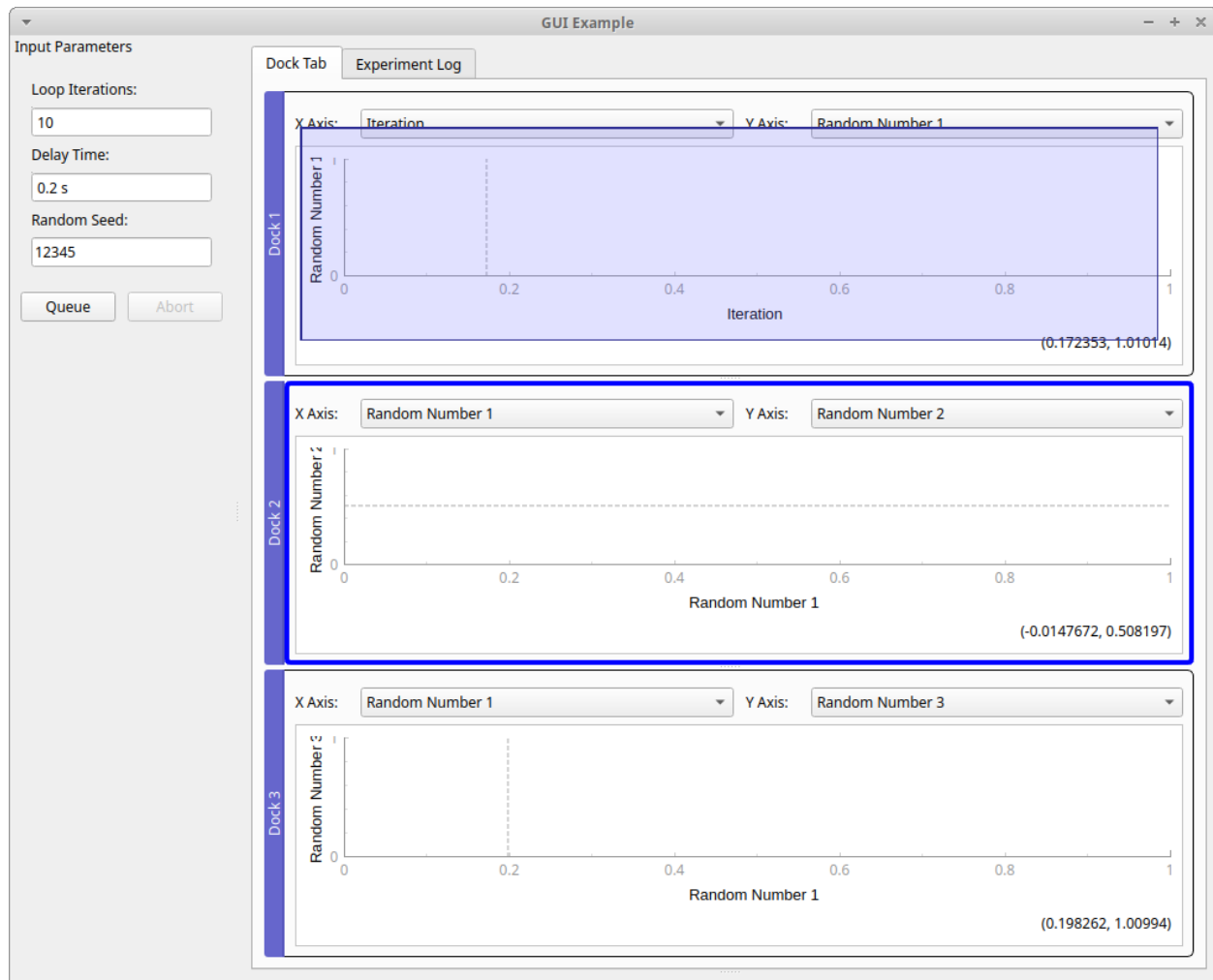


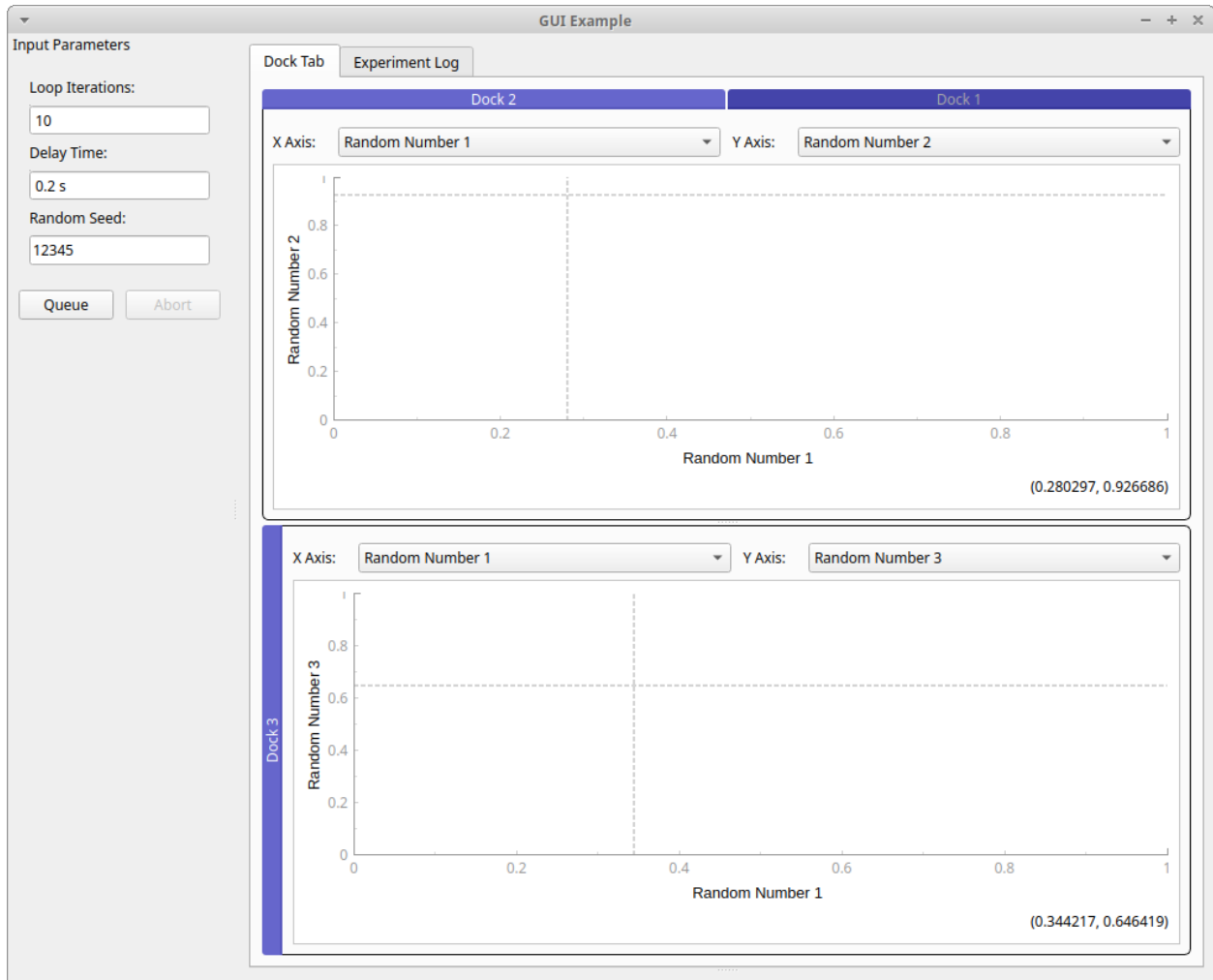


You can also drag the blue “Dock #” title bar to the top or bottom side of another plot to reposition a plot to rearrange the vertical order of the plots:



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:





### 3.3.11 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')

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

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

    self.manager.queue(experiment)

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()

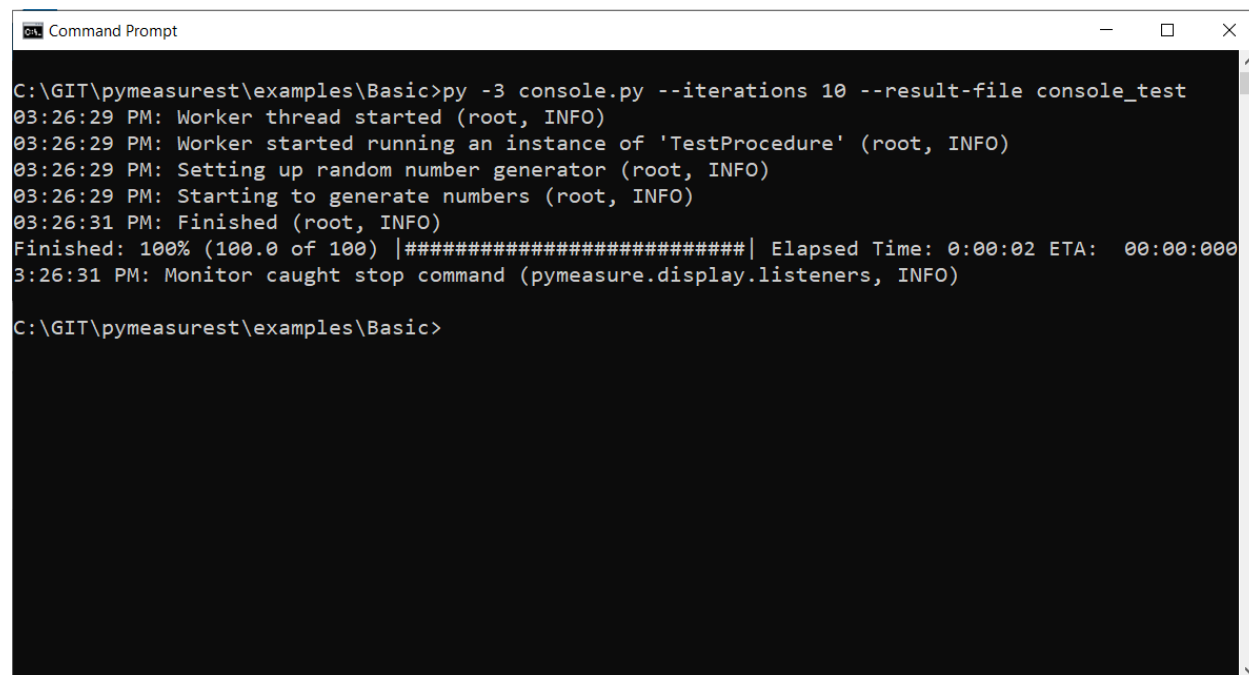
    sys.exit(app.exec())

```

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](#)):



```

C:\GIT\pymeasurest\examples\Basic>py -3 console.py --iterations 10 --result-file console_test
03:26:29 PM: Worker thread started (root, INFO)
03:26:29 PM: Worker started running an instance of 'TestProcedure' (root, INFO)
03:26:29 PM: Setting up random number generator (root, INFO)
03:26:29 PM: Starting to generate numbers (root, INFO)
03:26:31 PM: Finished (root, INFO)
Finished: 100% (100.0 of 100) |#####| Elapsed Time: 0:00:02 ETA: 00:00:00
3:26:31 PM: Monitor caught stop command (pymeasure.display.listeners, INFO)

C:\GIT\pymeasurest\examples\Basic>

```

### 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
```

## 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`(*preprocess\_reply=None, log=None, \*\*kwargs*)

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

This class should only be inherited from.

#### Parameters

- **preprocess\_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.  
Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **log** – Parent logger of the 'Adapter' logger.
- **\*\*kwargs** – Keyword arguments just to be cooperative.

**ask**(*command*)

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

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

#### Parameters

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

#### Returns

String ASCII response of the instrument

**binary\_values**(*command, header\_bytes=0, dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

#### Parameters

- **command** – SCPI command to be sent to the instrument
- **header\_bytes** – Integer number of bytes to ignore in header

- **dtype** – The NumPy data type to format the values with

**Returns**

NumPy array of values

**close()**

Close the connection.

**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'>, \*\*kwargs*)

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes**(*count=-1, 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).

**values**(*command, separator=', ', cast=<class 'float'>, preprocess\_reply=None*)

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

- **command** – SCPI command to be sent to the instrument

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

**Returns**

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

**write**(*command*, *\*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

**Parameters**

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

**write\_binary\_values**(*command*, *values*, *termination=""*, *\*\*kwargs*)

Write binary data to the instrument, e.g. waveform for signal generators

**Parameters**

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

**Returns**

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

**Parameters**

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

## 4.2 VISA adapter

```
class pymeasure.adapters.VISAAdapter(resource_name, visa_library="", preprocess_reply=None,
                                     query_delay=0, log=None, **kwargs)
```

Bases: [Adapter](#)

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

The workhorse of our library, used by most instruments.

**Parameters**

- **resource\_name** – A [VISA resource string](#) or GPIB address integer that identifies the target of the connection

- **visa\_library** – PyVISA VisaLibrary Instance, path of the VISA library or VisaLibrary spec string (@py or @ivi). If not given, the default for the platform will be used.
- **preprocess\_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.  
Deprecated since version 0.11: Implement it in the instrument’s *read* method instead.
- **query\_delay** (*float*) – Time in s to wait after writing and before reading.  
Deprecated since version 0.11: Implement it in the instrument’s *wait\_for* method instead.
- **log** – Parent logger of the ‘Adapter’ logger.
- **\*\*kwargs** – Keyword arguments for configuring the PyVISA connection.

### Kwargs

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

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

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

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

See [Modifying connection settings](#) for how to tweak settings when *connecting* to an instrument. See [Defining default connection settings](#) for how to best define default settings when *implementing an instrument*.

### `ask(command)`

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

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

#### Parameters

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

#### Returns

String ASCII response of the instrument

### `ask_values(command, **kwargs)`

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

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

#### Parameters

- **command** – SCPI command to be sent to the instrument
- **\*\*kwargs** – Key-word arguments to pass onto *query\_ascii\_values*

#### Returns

Formatted response of the instrument.

**binary\_values**(*command*, *header\_bytes*=0, *dtype*=<class 'numpy.float32'>)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

**Parameters**

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

**Returns**

NumPy array of values

**close()**

Close the connection.

---

**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 receive buffer contents and if data was present in the read buffer and no END-indicator was present, read from the device until encountering an END indicator (which causes loss of data).

**read(\*\*kwargs)**

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

Do not override in a subclass!

**Parameters**

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

**Returns str**

ASCII response of the instrument (excluding *read\_termination*).

**read\_binary\_values**(*header\_bytes*=0, *termination\_bytes*=None, *dtype*=<class 'numpy.float32'>, **\*\*kwargs**)

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes**(*count*=-1, *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).

**values**(*command*, *separator*=' ', *cast*=<class 'float'>, *preprocess\_reply*=None)

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

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

**Returns**

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

**wait\_for\_srq**(*timeout*=25, *delay*=0.1)

Block until a SRQ, and leave the bit high

**Parameters**

- **timeout** – Timeout duration in seconds
- **delay** – Time delay between checking SRQ in seconds

**write**(*command*, **\*\*kwargs**)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

**Parameters**

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

**write\_binary\_values**(*command*, *values*, *termination*="", **\*\*kwargs**)

Write binary data to the instrument, e.g. waveform for signal generators

**Parameters**

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

**Returns**

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

**Parameters**

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

## 4.3 Serial adapter

**class** `pymeasure.adapters.SerialAdapter`(*port*, *preprocess\_reply=None*, *write\_termination=""*,  
*read\_termination=""*, *\*\*kwargs*)

Bases: [Adapter](#)

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

**Parameters**

- **port** – Serial port
- **preprocess\_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.  
Deprecated since version 0.11: Implement it in the instrument's *read* method instead.
- **write\_termination** – String appended to messages before writing them.
- **read\_termination** – String expected at end of read message and removed.
- **\*\*kwargs** – Any valid key-word argument for `serial.Serial`

**\_format\_binary\_values**(*values*, *datatype='f'*, *is\_big\_endian=False*, *header\_fmt='ieee'*)

Format values in binary format, used internally in [Adapter.write\\_binary\\_values\(\)](#).

**Parameters**

- **values** – data to be written to the device.
- **datatype** – the format string for a single element. See `struct` module.
- **is\_big\_endian** – boolean indicating endianness.
- **header\_fmt** – Format of the header prefixing the data (“ieee”, “hp”, “empty”).

**Returns**

binary string.

**Return type**

bytes

**ask**(*command*)

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

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

**Parameters**

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

**Returns**

String ASCII response of the instrument

**binary\_values**(*command*, *header\_bytes*=0, *dtype*=<class 'numpy.float32'>)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

**Parameters**

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

**Returns**

NumPy array of values

**close()**

Close the connection.

**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'>, \*\**kwargs*)

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes**(*count*=-1, *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).

**values**(*command*, *separator*=' ', *cast*=<class 'float'>, *preprocess\_reply*=None)

Write a command to the instrument and returns a list of formatted values from the result.

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

#### Parameters

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

#### Returns

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

**write**(*command*, *\*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

#### Parameters

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

**write\_binary\_values**(*command*, *values*, *termination*='', *\*\*kwargs*)

Write binary data to the instrument, e.g. waveform for signal generators

#### Parameters

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

#### Returns

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

#### Parameters

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

## 4.4 Prologix adapter

```
class pymeasure.adapters.PrologixAdapter(resource_name, address=None, rw_delay=0,
                                         serial_timeout=None, preprocess_reply=None, 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.
- **rw\_delay** – An optional delay to set between a write and read call for slow to respond instruments.

Deprecated since version 0.11: Implement it in the instrument’s *wait\_for* method instead.

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

Deprecated since version 0.11: Implement it in the instrument’s *read* method instead.

- **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)
sourceter = Keithley2400(adapter) # at GPIB address 7
# generate another instance with a different GPIB address:
adapter2 = adapter.gpib(9)
multimeter = Keithley2000(adapter2) # at GPIB address 9
```

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

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

Then reload the udev rules with:

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

**\_format\_binary\_values**(*values*, *datatype*='f', *is\_big\_endian*=False, *header\_fmt*='ieee')

Format values in binary format, used internally in [write\\_binary\\_values\(\)](#).

**Parameters**

- **values** – data to be 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

**ask**(*command*)

Ask the Prologix controller.

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

**Parameters**

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

**ask\_values**(*command*, *\*\*kwargs*)

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

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

**Parameters**

- **command** – SCPI command to be sent to the instrument
- **\*\*kwargs** – Key-word arguments to pass onto *query\_ascii\_values*

**Returns**

Formatted response of the instrument.

**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.

**binary\_values**(*command*, *header\_bytes*=0, *dtype*=<class 'numpy.float32'>)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

**Parameters**

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

**Returns**

NumPy array of values

**close()**

Close the connection.

---

**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 receive 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'>, \*\*kwargs*)

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes**(*count=-1, 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.

**values**(*command, separator=', ', cast=<class 'float'>, preprocess\_reply=None*)

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

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

**Returns**

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

**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 VXI-11 adapter

**class** `pymeasure.adapters.VXI11Adapter`(*host*, *preprocess\_reply=None*, *\*\*kwargs*)

Bases: [`Adapter`](#)

**VXI11 Adapter class. Provides a adapter object that**

wraps around the read, write and ask functionality of the vx11 library.

Deprecated since version 0.11: Use `VISAAdapter` instead.

### Parameters

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

**ask**(*command*)

Wrapper function for the ask command using the vx11 interface.

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

### Parameters

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

:returns string containing a response from the device.

**ask\_raw**(*command*)

Wrapper function for the ask\_raw command using the vx11 interface.

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

### Parameters

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

:returns binary string containing the response from the device.

**binary\_values**(*command*, *header\_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

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

### Parameters

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

### Returns

NumPy array of values

**close**()

Close the connection.

**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'>, \*\*kwargs*)

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes**(*count=-1, 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).

**read\_raw()**

Read bytes from the device.

Deprecated since version 0.11: Use *read\_bytes* instead.

**values**(*command, separator=', ', cast=<class 'float'>, preprocess\_reply=None*)

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

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

**Returns**

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

**write**(*command*, *\*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

**Parameters**

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

**write\_binary\_values**(*command*, *values*, *termination=""*, *\*\*kwargs*)

Write binary data to the instrument, e.g. waveform for signal generators

**Parameters**

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

**Returns**

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

**Parameters**

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

**write\_raw**(*command*)

Write bytes to the device.

Deprecated since version 0.11: Use *write\_bytes* instead.

## 4.6 Telnet adapter

**class** `pymeasure.adapters.TelnetAdapter`(*host*, *port=0*, *query\_delay=0*, *preprocess\_reply=None*, *\*\*kwargs*)

Bases: [\*Adapter\*](#)

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

Deprecated since version 0.11.2: The Python telnetlib module is deprecated since Python 3.11 and will be removed in Python 3.13 release. As a result, TelnetAdapter is deprecated, use VISAAdapter instead. The VISAAdapter supports TCPIP socket connections. When using the VISAAdapter, the *resource\_name* argument should be *TCPIP[board]::<host>::<port>::SOCKET*. see here, <<https://pyvisa.readthedocs.io/en/latest/introduction/names.html>>

**Parameters**

- **host** – host address of the instrument
- **port** – TCPIP port

- **query\_delay** – delay in seconds between write and read in the ask method
- **preprocess\_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.  
Deprecated since version 0.11: Implement it in the instrument’s *read* method instead.
- **kwargs** – Valid keyword arguments for telnetlib.Telnet, currently this is only ‘timeout’

**ask**(*command*)

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

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

**Parameters**

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

**Returns**

String ASCII response of the instrument

**binary\_values**(*command*, *header\_bytes*=0, *dtype*=<class 'numpy.float32'>)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

**Parameters**

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

**Returns**

NumPy array of values

**close**()

Close the connection.

**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'>, \*\**kwargs*)

Returns a numpy array from a query for binary data

**Parameters**

- **header\_bytes** (*int*) – Number of bytes to ignore in header.
- **termination\_bytes** (*int*) – Number of bytes to strip at end of message or None.
- **dtype** – The NumPy data type to format the values with.

- **\*\*kwargs** – Further arguments for the NumPy fromstring method.

**Returns**

NumPy array of values

**read\_bytes**(*count=-1, 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).

**values**(*command, separator=', ', cast=<class 'float'>, preprocess\_reply=None*)

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

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

**Returns**

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

**write**(*command, \*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

**Parameters**

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

**write\_binary\_values**(*command, values, termination="", \*\*kwargs*)

Write binary data to the instrument, e.g. waveform for signal generators

**Parameters**

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

**Returns**

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

**Parameters**

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

## 4.7 Test adapters

These pieces are useful when writing tests.

`pymeasure.test.expected_protocol(instrument_cls, comm_pairs, connection_attributes={}, connection_methods={}, **kwargs)`

Context manager that checks sent/received instrument commands without a device connected.

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

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

**Parameters**

- **instrument\_cls** (*pymeasure.Instrument*) – *Instrument* subclass to instantiate.
- **comm\_pairs** (*list[2-tuples[str]]*) – List of command-response pairs, i.e. 2-tuples like ('VOLT?', '3.14'). 'None' indicates that a pair member (command or response) does not exist, e.g. (None, 'RESPI'). 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=None, preprocess_reply=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`(*preprocess\_reply=None, 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() == ""
assert a.ask("10") == "10"
assert a.values("10") == [10]
```

### **ask**(*command*)

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

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

#### **Parameters**

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

#### **Returns**

String ASCII response of the instrument

**binary\_values**(*command, header\_bytes=0, dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Deprecated since version 0.11: Call *Instrument.binary\_values* instead.

#### **Parameters**

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

#### **Returns**

NumPy array of values

### **close()**

Close the connection.

**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'>, \*\*kwargs)**

Returns a numpy array from a query for binary data

**Parameters**

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

**Returns**

NumPy array of values

**read\_bytes(count=-1, 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).

**values(command, separator=',', cast=<class 'float'>, preprocess\_reply=None)**

Write a command to the instrument and returns a list of formatted values from the result.

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

**Parameters**

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

**Returns**

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

**write**(*command*, *\*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

Do not override in a subclass!

**Parameters**

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

**write\_binary\_values**(*command*, *values*, *termination=""*, *\*\*kwargs*)

Write binary data to the instrument, e.g. waveform for signal generators

**Parameters**

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

**Returns**

number of bytes written

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

Do not override in a subclass!

**Parameters**

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

**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 ‘pymasure.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.

## 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()**

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

**gen\_measurement()**

Create MEASURE and DATA\_COLUMNS variables for get\_datapoint method.

**get\_estimates()**

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

**metadata\_objects()**

Returns a dictionary of all the Metadata objects

**parameter\_objects()**

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

**parameter\_values()**

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

**parameters\_are\_set()**

Returns True if all parameters are set

**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.

**refresh\_parameters()**

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

**set\_parameters(parameters, except\_missing=True)**

Sets a dictionary of parameters and raises an exception if additional parameters are present if `except_missing` is True

**shutdown()**

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

**startup()**

Executes the commands needed at the start-up of the measurement

**class** `pymeasure.experiment.procedure.UnknownProcedure(parameters)`

Handles the case when a *Procedure* object can not be imported during loading in the *Results* class

**startup()**

Executes the commands needed at the start-up of the measurement

## 5.4 Parameter classes

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

**class** `pymeasure.experiment.parameters.BooleanParameter(name, default=None, ui_class=None, group_by=None, group_condition=True)`

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

**Variables**

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

**Parameters**

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

**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)
```

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.

#### 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*, *record*)

Emits data of some topic over TCP

**handle\_abort**()

**handle\_error**()

**join**(*timeout=0*)

Joins the current thread and forces it to stop after the timeout if necessary

**Parameters**

**timeout** – Timeout duration in seconds

**run**()

Method representing the thread's activity.

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

**shutdown**()

**update\_status**(*status*)

## 5.6 Results class

**class** `pymeasure.experiment.results.CSVFormatter`(*columns*, *delimiter=','*)

Formatter of data results

**format**(*record*)

Formats a record as csv.

**Parameters**

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

**Returns**

a string

**class** pymeasure.experiment.results.**Results**(*procedure, data\_filename*)

The Results class provides a convenient interface to reading and writing data in connection with a *Procedure* object.

#### Variables

- **COMMENT** – The character used to identify a comment (default: #)
- **DELIMITER** – The character used to delimit the data (default: ,)
- **LINE\_BREAK** – The character used for line breaks (default n)
- **CHUNK\_SIZE** – The length of the data chunk 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**()

Performs 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.

```
pymethods.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



## PYMEASURE.DISPLAY

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

### 6.1 Browser classes

**class** `pymasure.display.browser.BaseBrowserItem`

Bases: `object`

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

**class** `pymasure.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** `pymasure.display.browser.BrowserItem`(*results, color, parent=None*)

Bases: `QTreeWidgetItem`, [BaseBrowserItem](#)

Represent a row in the [Browser](#) tree widget

### 6.2 Console class

**class** `pymasure.display.console.ConsoleArgumentParser`(*procedure\_class, \*\*kwargs*)

Bases: `ArgumentParser`

**setup\_parser**()

Setup command line arguments parsing from parameters information

**class** `pymasure.display.console.ConsoleBrowserItem`(*progress\_bar*)

Bases: [BaseBrowserItem](#)

```
class pymeasure.display.console.ManagedConsole(procedure_class, log_channel="", log_level=20)
```

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

```
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.

**setEnabled**(*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.

**setEnabled**(*self*) → QAbstractSpinBox.StepEnabled

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

**validate**(*self*, *input: str*, *pos: int*) → Tuple[QValidator.State, str, int]

**valueFromText**(*self*, *text: str*) → 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.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)
```

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, option: QStyleOptionViewItem, index: QModelIndex*) → `QWidget`

**setEditorData**(*self, editor: QWidget, index: QModelIndex*)

**setModelData**(*self, editor: QWidget, model: QAbstractItemModel, index: QModelIndex*)

**updateEditorGeometry**(*self, editor: QWidget, option: QStyleOptionViewItem, index: QModelIndex*)

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

Bases: `QValidator`

**validate**(*self, a0: str, a1: int*) → `Tuple[QValidator.State, str, int]`

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

Bases: `QStyledItemDelegate`

**createEditor**(*self*, *parent*: *QWidget*, *option*: *QStyleOptionViewItem*, *index*: *QModelIndex*) → *QWidget*

**setEditorData**(*self*, *editor*: *QWidget*, *index*: *QModelIndex*)

**setModelData**(*self*, *editor*: *QWidget*, *model*: *QAbstractItemModel*, *index*: *QModelIndex*)

**updateEditorGeometry**(*self*, *editor*: *QWidget*, *option*: *QStyleOptionViewItem*, *index*: *QModelIndex*)

**class** `pymasure.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** `pymasure.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*)

**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)
```

```
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:  $(k*n) \times (\max(l(x) \ x=1..n))$  - By row: column fixed to the number of series, in this case table shape will be:  $k \times (\text{sum of } l(x) \ x=1..n)$

**columnCount**(self, parent: `QModelIndex = QModelIndex()`)  $\rightarrow$  int

**data**(self, index: `QModelIndex`, role: int = `Qt.ItemDataRole.DisplayRole`)  $\rightarrow$  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()`)  $\rightarrow$  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*)

**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,
                                                                    directory_input=False,
                                                                    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 code: *Load sequence* button
- **inputs\_in\_scrollarea** – boolean that display or hide a scrollbar to the input area
- **directory\_input** – specify, if present, where the experiment’s result will be saved.
- **hide\_groups** – a boolean controlling whether parameter groups are hidden (True, default) or disabled/grayed-out (False) when the group conditions are not met.

#### **open\_file\_externally**(filename)

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

#### **queue**(procedure=None)

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

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

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

For example:

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

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

    self.manager.queue(experiment)
```

#### **set\_parameters**(parameters)

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

**class** pymeasure.display.windows.plotter\_window.**PlotterWindow**(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](#)



## PYMEASURE.INSTRUMENTS

This section contains documentation on the instrument classes.

### 7.1 Instrument classes

**class** `pymeasure.instruments.common_base.CommonBase`(*preprocess\_reply=None, \*\*kwargs*)

Base class for instruments and channels.

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

#### Parameters

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

Deprecated since version 0.11: Implement it in the instrument's *read* method instead.

**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=0*)

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

**Parameters**

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

**Returns**

String returned by the device without *read\_termination*.

**binary\_values**(*command*, *query\_delay=0*, *\*\*kwargs*)

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

**Parameters**

- **command** – Command to be sent to the instrument.
- **query\_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for *read\_binary\_values()*.

**Returns**

NumPy array of values.

**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>>,
               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 take a value and allows processing before value mapping, returning the processed value
- **set\_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command\_process** – A function that takes a command and allows processing before executing the command

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=None, get_process=<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 take a value and allows processing before value mapping, returning the processed value
- **command\_process** – A function that take a command and allows processing before executing the command, for getting

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=0*)

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

Implement in subclass!

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**class** pymeasure.instruments.**Instrument**(*adapter, name, includeSCPI=True, preprocess\_reply=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** – A boolean, which toggles the inclusion of standard SCPI commands
- **preprocess\_reply** – An optional callable used to preprocess strings received from the instrument. The callable returns the processed string.  
Deprecated since version 0.11: Implement it in the instrument’s *read* method instead.
- **\*\*kwargs** – 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 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=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command, \*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command, values, \*args, \*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content, \*\*kwargs*)

Write the bytes *content* to the instrument.

**class** pymeasure.instruments.**Channel**(*parent, id*)

The base class for channel definitions.

This class supports dynamic properties like *Instrument*, but requires an *Instrument* instance as a parent for communication.

*insert\_id()* inserts the channel id into the command string sent to the instrument. The default implementation replaces the Channel’s *placeholder* (default “ch”) with the channel id in all command strings (e.g. “CHANNEL{ch}:foo”).

**Parameters**

- **parent** – The instrument (an instance of *Instrument*) to which the channel belongs.
- **id** – Identifier of the channel, as it is used for the communication.

**check\_errors**()

Read all errors from the instrument 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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write(command, \*\*kwargs)**

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument. ‘{ch}’ is replaced by the channel id.
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values(command, values, \*args, \*\*kwargs)**

Write binary values to the instrument.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes(content, \*\*kwargs)**

Write the bytes *content* to the instrument.

```
class pymeasure.instruments.fakes.FakeInstrument(adapter=None, name='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 Validator functions

Validators are used in conjunction with the `Instrument.control` or `Instrument.setting` functions to allow properties with complex restrictions for valid values. They are described in more detail in the [Restricting values with validators](#) section.

`pymeasure.instruments.validators.discreteTruncate(number, discreteSet)`

Truncates the number to the closest element in the positive discrete set. Returns False if the number is larger than the maximum value or negative.

`pymeasure.instruments.validators.joined_validators(*validators)`

Returns a validator function that represents a list of validators joined together.

A value passed to the validator is returned if it passes any validator (not all of them). Otherwise it raises a `ValueError`.

Note: the joined validator expects values to be a sequence of values appropriate for the respective validators (often sequences themselves).

### Example

```
>>> from pymeasure.instruments.validators import strict_discrete_set, strict_range
>>> from pymeasure.instruments.validators import joined_validators
>>> joined_v = joined_validators(strict_discrete_set, strict_range)
>>> values = [['MAX', 'MIN'], range(10)]
>>> joined_v(5, values)
5
>>> joined_v('MAX', values)
'MAX'
>>> joined_v('NONSENSE', values)
Traceback (most recent call last):
...
ValueError: Value of NONSENSE does not match any of the joined validators
```

### Parameters

**validators** – an iterable of other validators

`pymeasure.instruments.validators.modular_range(value, values)`

Provides a validator function that returns the value if it is in the range. Otherwise it returns the value, modulo the max of the range.

### Parameters

- **value** – a value to test
- **values** – A set of values that are valid

`pymeasure.instruments.validators.modular_range_bidirectional(value, values)`

Provides a validator function that returns the value if it is in the range. Otherwise it returns the value, modulo the max of the range. Allows negative values.

### Parameters

- **value** – a value to test
- **values** – A set of values that are valid

`pymeasure.instruments.validators.strict_discrete_range(value, values, step)`

Provides a validator function that returns the value if its value is less than the maximum and greater than the minimum of the range and is a multiple of step. Otherwise it raises a `ValueError`.

**Parameters**

- **value** – A value to test
- **values** – A range of values (range, list, etc.)
- **step** – Minimum stepsize (resolution limit)

**Raises**

`ValueError` if the value is out of the range

`pymeasure.instruments.validators.strict_discrete_set(value, values)`

Provides a validator function that returns the value if it is in the discrete set. Otherwise it raises a `ValueError`.

**Parameters**

- **value** – A value to test
- **values** – A set of values that are valid

**Raises**

`ValueError` if the value is not in the set

`pymeasure.instruments.validators.strict_range(value, values)`

Provides a validator function that returns the value if its value is less than or equal to the maximum and greater than or equal to the minimum of values. Otherwise it raises a `ValueError`.

**Parameters**

- **value** – A value to test
- **values** – A range of values (range, list, etc.)

**Raises**

`ValueError` if the value is out of the range

`pymeasure.instruments.validators.truncated_discrete_set(value, values)`

Provides a validator function that returns the value if it is in the discrete set. Otherwise, it returns the smallest value that is larger than the value.

**Parameters**

- **value** – A value to test
- **values** – A set of values that are valid

`pymeasure.instruments.validators.truncated_range(value, values)`

Provides a validator function that returns the value if it is in the range. Otherwise it returns the closest range bound.

**Parameters**

- **value** – A value to test
- **values** – A set of values that are valid

## 7.3 Comedi data acquisition

The Comedi libraries provide a convenient method for interacting with data acquisition cards, but are restricted to Linux compatible operating systems.

`pymeasure.instruments.comedi.getAI(device, channel, range=None)`

Returns the analog input channel as specified for a given device

`pymeasure.instruments.comedi.getAO(device, channel, range=None)`

Returns the analog output channel as specified for a given device

`pymeasure.instruments.comedi.readAI(device, channel, range=None, count=1)`

Reads a single measurement (count==1) from the analog input channel of the device specified. Multiple readings can be preformed with count not equal to one, which are seperated by an arbitrary time

`pymeasure.instruments.comedi.writeAO(device, channel, voltage, range=None)`

Writes a single voltage to the analog output channel of the device specified

## 7.4 Resource Manager

The `list_resources` function provides an interface to check connected instruments interactively.

`pymeasure.instruments.list_resources()`

Prints the available resources, and returns a list of VISA resource names

```
resources = list_resources()
#prints (e.g.)
#0 : GPIB0::22::INSTR : Agilent Technologies,34410A,*****
#1 : GPIB0::26::INSTR : Keithley Instruments Inc., Model 2612, *****
dmm = Agilent34410(resources[0])
```

Instruments by manufacturer:

## 7.5 Active Technologies

This section contains specific documentation on the Active Technologies instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

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

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

Bases: `AWG401x_base`

Represents the Active Technologies AWG-401x Arbitrary Waveform Generator in AFG mode.

```
wfg = AWG401x_AFG("TCPIP::192.168.0.123::INSTR")

wfg.reset()                                # Reset the instrument at default state

wfg.ch[1].shape = "SINUSOID"               # Sets a sine waveform on CH1
```

(continues on next page)

(continued from previous page)

```

wfg.ch[1].frequency = 4.7e3      # Sets the frequency to 4.7 kHz on CH1
wfg.ch[1].amplitude = 1         # Set amplitude of 1 V on CH1
wfg.ch[1].offset = 0           # Set the amplitude to 0 V on CH1
wfg.ch[1].enabled = True       # Enables the CH1

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

wfg.enabled = True             # Enable output of waveform generator
wfg.beep()                    # "beep"

print(wfg.check_errors())      # Get the error queue

```

**ch\_1**

**Channel**

*ChannelAFG*

**ch\_2**

**Channel**

*ChannelAFG*

**property enabled**

A boolean property that enables the generation of signals.

**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].ch[i].voltage_high = 1      # Sets high voltage = 1
    wfg.entries[1].ch[i].voltage_low = 0       # Sets low voltage = 1
    wfg.entries[1].ch[i].waveform = "SQUARE"   # Sets a square wave
    wfg.setting_ch[i].enabled = True           # Enable channel

wfg.entries.resize(2)        # Resize the number of entries to 2

wfg.entries[2].ch[1].waveform = "MyWaveform"  # Set custom waveform

wfg.enabled = True           # Enable output of waveform generator
wfg.beep()                   # "beep"

print(wfg.check_errors())    # Get the error queue

```

**class DummyEntriesElements**(parent, number\_of\_channel)

Bases: 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**

This property sets or queries the burst count parameter.(dynamic)

**property burst\_count\_max**

This property queries the maximum burst count parameter.

**property burst\_count\_min**

This property queries the minimum burst count parameter.

**property enabled**

A boolean property that enables the generation of signals.

**property entry\_level\_strategy**

This property sets or or returns the Entry Length Strategy. This strategy manages the length of the sequencer entries in relationship with the length of the channel waveforms defined for each entry. The possible values are:

- ADAPTL<ONGER>: the length of an entry of the sequencer by default will be equal to the length of the longer channel waveform, among all analog channels, assigned to the entry.
- ADAPTS<HORTER>: the length of an entry of the sequencer by default will be equal to the length of the shorter channel waveform, among all analog channels, assigned to the entry.
- DEF<AULT>:the length of an entry of the sequencer by default will be equal to the value specified in the Sequencer Item Default Length [N] parameter

**list\_files**(path=None)

Return a List of tuples with all file found in a directory. If the path is not specified the current directory will be used

**property num\_ch**

This property queries the number of analog channels.

**property num\_dch**

This property queries the number of digital channels.

**remove\_file**(file\_name, path=None)

Remove a specified file

**property run\_mode**

This property sets or returns the AWG run mode. The possible values are:

- CONT<INUOUS>: each waveform will loop as written in the entry repetition parameter and the entire sequence is repeated circularly

- **BURS<T>**: the AWG waits for a trigger event. When the trigger event occurs each waveform will loop as written in the entry repetition parameter and the entire sequence will be repeated circularly many times as written in the Burst Count[N] parameter. If you set Burst Count[N]=1 the instrument is in Single mode and the sequence will be repeated only once.
- **TCON<TINUOUS>**: the AWG waits for a trigger event. When the trigger event occurs each waveform will loop as written in the entry repetition parameter and the entire sequence will be repeated circularly.
- **STEP<PED>**: the AWG, for each entry, waits for a trigger event before the execution of the sequencer entry. The waveform of the entry will loop as written in the entry repetition parameter. After the generation of an entry has completed, the last sample of the current entry or the first sample of the next entry is held until the next trigger is received. At the end of the entire sequence the execution will restart from the first entry.
- **ADVA<NCED>**: it enables the “Advanced” mode. In this mode the execution of the sequence can be changed by using conditional and unconditional jumps (JUMPTO and GOTO commands) and dynamic jumps (PATTERN JUMP commands).

The **\*RST** command sets this parameter to CONTinuous.

#### **property run\_status**

This property returns the run state of the AWG. The possible values are: STOPPED, WAITING\_TRIGGER, RUNNING

#### **property sample\_decreasing\_strategy**

This property sets or returns the Sample Decreasing Strategy. The “Sample decreasing strategy” parameter defines the strategy used to adapt the waveform length to the sequencer entry length in the case where the original waveform length is longer than the sequencer entry length. Can be set to: DECIM<ATION>, CUTT<AIL>, CUTH<EAD>

#### **property sample\_increasing\_strategy**

This property sets or or returns the Sample Increasing Strategy. The “Sample increasing strategy” parameter defines the strategy used to adapt the waveform length to the sequencer entry length in the case where the original waveform length is shorter than the sequencer entry length. Can be set to: INTER<POLATION>, RETURN<ZERO>, HOLD<LAST>, SAMPLESM<ULTIPLICATION>

#### **property sampling\_rate**

This property sets or queries the sample rate for the Sampling Clock.(dynamic)

#### **property sampling\_rate\_max**

This property queries the maximum sample rate for the Sampling Clock.

#### **property sampling\_rate\_min**

This property queries the minimum sample rate for the Sampling Clock.

#### **save\_file(file\_name, data, path=None, override\_existing=False)**

Write a string in a file in the instrument

#### **trigger()**

Force a trigger event to occur.

#### **property trigger\_source**

This property sets or returns the instrument trigger source. The possible values are:

- **TIM<ER>**: the trigger is sent at regular intervals.
- **EXT<ERNAL>**: the trigger come from the external BNC connector.
- **MAN<UAL>**: the trigger is sent via software or using the trigger button on front panel.

**property waveforms**

This property returns a dict with all the waveform present in the instrument system (Wave. List). It is possible to modify the values, delete them or create new waveforms

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

Bases: ChannelBase

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

**property baseline\_offset**

This property sets or queries the offset level for the specified channel. The offset range setting depends on the amplitude parameter. (dynamic)

**property baseline\_offset\_max**

This property queries the maximum offset voltage level that can be set to the output waveform.

**property baseline\_offset\_min**

This property queries the minimum offset voltage level that can be set to the output waveform.

**property frequency**

This property sets or queries 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**

This property queries the maximum frequency that can be set to the output waveform.

**property frequency\_min**

This property queries the minimum frequency that can be set to the output waveform.

**property load\_impedance**

This property sets 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**

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

**property phase**

This property sets or queries the phase of the output waveform for the specified channel. The value is in degrees.(dynamic)

**property phase\_max**

This property queries the maximum phase that can be set to the output waveform.

**property phase\_min**

This property queries the minimum phase that can be set to the output waveform.

**property shape**

This property sets or queries the shape of the carrier waveform. Allowed choices depends on the chosen 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**

This property sets or queries 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**

This property queries the maximum amplitude voltage level that can be set to the output waveform.

**property voltage\_amplitude\_min**

This property queries the minimum amplitude voltage level that can be set to the output waveform.

**property voltage\_high**

This property sets or queries 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**

This property queries the maximum high voltage level that can be set to the output waveform.

**property voltage\_high\_min**

This property queries the minimum high voltage level that can be set to the output waveform.

**property voltage\_low**

This property sets or queries 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**

This property queries the maximum low voltage level that can be set to the output waveform.

**property voltage\_low\_min**

This property queries the minimum low voltage level that can be set to the output waveform.

**property voltage\_offset**

This property sets or queries the offset level for the specified channel. The offset range setting depends on the amplitude parameter. (dynamic)

**property voltage\_offset\_max**

This property queries the maximum offset voltage level that can be set to the output waveform.

**property voltage\_offset\_min**

This property queries the minimum offset voltage level that can be set to the output waveform.

**property voltage\_unit**

This property sets or queries 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.

## 7.6 Advantest

This section contains specific documentation on the Advantest instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

### 7.6.1 Advantest R3767CG Vector Network Analyzer

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

Bases: [Instrument](#)

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

**property center\_frequency**

Center Frequency in Hz

**property id**

Reads the instrument identification

**property span\_frequency**

Span Frequency in Hz

**property start\_frequency**

Starting frequency in Hz

**property stop\_frequency**

Stopping frequency in Hz

**property trace\_1**

Reads the Data array from trace 1 after formatting

### 7.6.2 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: [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
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 and log them.

**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 ( $T_a < T_w$  and  $T_a < T_d + T_{it}$ ), an execution error will occur and it will not be set:

$T_w$ : Pulse width  $T_d$ : Major delay time  $T_{it}$ : 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 scout 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**

A query command that runs a self-test and reads the result (\*TST?).

#### **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  $\mu$ s. 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 ( $T_w$ ), pulse period ( $T_p$ ), and measure delay time ( $T_d$ ) 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 ( $T_w$ ) and pulse period ( $T_p$ ). 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 ( $T_p$ ) setting. (For pulse sweep measurements, there are no restrictions.)

- $T_p < 2\text{ms}$  : Not measured.
  - $2\text{ms} \leq T_p < 10\text{ms}$  : Measure once every 5 ~ 20ms.
  - $10\text{ms} \leq T_p$  : 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  $m \times N \leq 2048$
      - \*  $m \times N \leq 2048$  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,  $m = 1$  (one-way sweep),  $m = 2$  (round-trip sweep).
      - \* When the OFM command sets the output data output method to 1 or 2  $m \times N \leq 2048$
      - \*  $m \times N \leq 2048$  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 = 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\_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,  $m \times N$  number of repeats  $\times N \leq 2048$ .
      - \*  $m \times N \leq 2048$  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,  $m \times N$  number of repeats  $\times N \leq 2048$ .
      - \*  $m \times N \leq 2048$  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  $N = (\text{stop number} - \text{start number} + 1)$ ,  $m = 1$  (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*) – Address 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=None, *,
                                                                module=None, qualname=None,
                                                                type=None, start=1,
                                                                boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.SampleMode(value, names=None, *,
                                                                module=None, qualname=None,
                                                                type=None, start=1,
                                                                boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.VoltageRange(value, names=None, *,
                                                                    module=None,
                                                                    qualname=None, type=None,
                                                                    start=1, boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.CurrentRange(value, names=None, *,
                                                                    module=None,
                                                                    qualname=None, type=None,
                                                                    start=1, boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.SweepMode(value, names=None, *,
                                                                module=None, qualname=None,
                                                                type=None, start=1,
                                                                boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.OutputType(value, names=None, *,
                                                                module=None, qualname=None,
                                                                type=None, start=1,
                                                                boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.TriggerInputType(value, names=None, *,
                                                                    module=None,
                                                                    qualname=None,
                                                                    type=None, start=1,
                                                                    boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.MeasurementType(value, names=None, *,
                                                                    module=None,
                                                                    qualname=None,
                                                                    type=None, start=1,
                                                                    boundary=None)
```

Bases: IntEnum

```
class pymeasure.instruments.advantest.advantestR624X.SequenceInterruptType(value,
                                                                              names=None,
                                                                              *,
                                                                              module=None,
                                                                              qual-
                                                                              name=None,
                                                                              type=None,
                                                                              start=1,
                                                                              bound-
                                                                              ary=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=None, *, module=None,
                                                         qualname=None, type=None, start=1,
                                                         boundary=None)
```

Bases: IntFlag

bit assignment 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=None, *, module=None,
                                                         qualname=None, type=None, start=1,
                                                         boundary=None)
```

Bases: IntFlag

bit assignment 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=None, *, module=None,
qualname=None, type=None, start=1,
boundary=None)
```

Bases: IntFlag

bit assignment 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 0 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=None, *, module=None,
qualname=None, type=None, start=1,
boundary=None)
```

Bases: IntFlag

bit assignment 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=None,
                                                                              *, mod-
                                                                              ule=None,
                                                                              qual-
                                                                              name=None,
                                                                              type=None,
                                                                              start=1,
                                                                              bound-
                                                                              ary=None)
```

Bases: IntFlag

bit assignment 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

- *Advantest R6245/R6246 DC Voltage/Current Sources/Monitors*
  - *Main Classes*
  - *General Information*
  - *Examples*
    - \* *Initialization of the Instrument*
    - \* *Simple dual channel measurement example*
    - \* *Program example for DC measurement*
    - \* *Program example for DC measurement (with external trigger)*
    - \* *Program example for pulse measurement*
    - \* *Fixed Level Sweep Program Example*

## 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
```

(continues on next page)

(continued from previous page)

```

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
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)
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
                               pulsed_width = 0, # 0 sec pulse width
                               pulsed_period = 0) # 0 sec pulse period
↳between measurements
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
    ↪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
```

(continues on next page)

(continued from previous page)

```
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
```

## 7.7 Agilent

This section contains specific documentation on the Agilent instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

### 7.7.1 Agilent 8257D Signal Generator

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

Bases: *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.

## 7.7.2 Agilent 8722ES Vector Network Analyzer

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

Bases: *Instrument*

Represents the Agilent8722ES Vector Network Analyzer and provides a high-level interface for taking scans of the scattering parameters.

**property averages**

An integer representing the number of averages to take. Note that averaging must be enabled for this to take effect. This property can be set.

**property averaging\_enabled**

A bool that indicates whether or not averaging is enabled. This property can be set.

**property data**

Returns the real and imaginary data from the last scan

**property data\_complex**

Returns the complex power from the last scan

**property data\_log\_magnitude**

Returns the absolute magnitude values in dB from the last scan

**property data\_magnitude**

Returns the absolute magnitude values from the last scan

**property data\_phase**

Returns the phase in degrees from the last scan

**disable\_averaging()**

Disables averaging

**enable\_averaging()**

Enables averaging

**property frequencies**

Returns a list of frequencies from the last scan

**is\_averaging()**

Returns True if averaging is enabled

**log\_magnitude(*real*, *imaginary*)**

Returns the magnitude in dB from a real and imaginary number or numpy arrays

**magnitude**(*real, imaginary*)

Returns the magnitude from a real and imaginary number or numpy arrays

**phase**(*real, imaginary*)

Returns the phase in degrees from a real and imaginary number or numpy arrays

**scan**(*averages=None, blocking=None, timeout=None, delay=None*)

Initiates a scan with the number of averages specified and blocks until the operation is complete.

**scan\_continuous**()

Initiates a continuous scan

**property scan\_points**

Gets the number of scan points

**scan\_single**()

Initiates a single scan

**set\_IF\_bandwidth**(*bandwidth*)

Sets the resolution bandwidth (IF bandwidth)

**set\_averaging**(*averages*)

Sets the number of averages and enables/disables averaging. Should be between 1 and 999

**set\_fixed\_frequency**(*frequency*)

Sets the scan to be of only one frequency in Hz

**property start\_frequency**

A floating point property that represents the start frequency in Hz. This property can be set.

**property stop\_frequency**

A floating point property that represents the stop frequency in Hz. This property can be set.

**property sweep\_time**

A floating point property that represents the sweep time in seconds. This property can be set.

### 7.7.3 Agilent E4408B Spectrum Analyzer

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

Bases: [Instrument](#)

Represents the AgilentE4408B Spectrum Analyzer and provides a high-level interface for taking scans of high-frequency spectrums

**property center\_frequency**

A floating point property that represents the center frequency in Hz. This property can be set.

**property frequencies**

Returns a numpy array of frequencies in Hz that correspond to the current settings of the instrument.

**property frequency\_points**

An integer property that represents the number of frequency points in the sweep. This property can take values from 101 to 8192.

**property frequency\_step**

A floating point property that represents the frequency step in Hz. This property can be set.

**property start\_frequency**

A floating point property that represents the start frequency in Hz. This property can be set.

**property stop\_frequency**

A floating point property that represents the stop frequency in Hz. This property can be set.

**property sweep\_time**

A floating point property that represents the sweep time in seconds. This property can be set.

**trace(number=1)**

Returns a numpy array of the data for a particular trace based on the trace number (1, 2, or 3).

**trace\_df(number=1)**

Returns a pandas DataFrame containing the frequency and peak data for a particular trace, based on the trace number (1, 2, or 3).

## 7.7.4 Agilent E4980 LCR Meter

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

Bases: [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: Series inductance [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

### 7.7.5 Agilent 34410A Multimeter

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

Bases: *Instrument*

Represent the HP/Agilent/Keysight 34410A and related multimeters.

Implemented measurements: voltage\_dc, voltage\_ac, current\_dc, current\_ac, resistance, resistance\_4w

**property current\_ac**

AC current, in Amps

**property current\_dc**

DC current, in Amps

**property resistance**

Resistance, in Ohms

**property resistance\_4w**

Four-wires (remote sensing) resistance, in Ohms

**property voltage\_ac**

AC voltage, in Volts

**property voltage\_dc**

DC voltage, in Volts

### 7.7.6 HP/Agilent/Keysight 34450A Digital Multimeter

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

Bases: *Instrument*

Represent the HP/Agilent/Keysight 34450A and related multimeters.

```
dmm = Agilent34450A("USB0:...")
dmm.reset()
dmm.configure_voltage()
print(dmm.voltage)
dmm.shutdown()
```

**beep()**

Sounds a system beep.

**property capacitance**

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

**property capacitance\_auto\_range**

A boolean property that toggles auto ranging for capacitance.

**property capacitance\_range**

A property that controls the capacitance range in Farads, which can take values 1E-9, 10E-9, 100E-9, 1E-6, 10E-6, 100E-6, 1E-3, 10E-3, as well as “MIN”, “MAX”, or “DEF” (1E-6). Auto-range is disabled when this property is set.

**configure\_capacitance**(*capacitance\_range*='AUTO')

Configures the instrument to measure capacitance.

**Parameters**

**capacitance\_range** – A capacitance in Farads to set the capacitance range, can be 1E-9, 10E-9, 100E-9, 1E-6, 10E-6, 100E-6, 1E-3, 10E-3, as well as “MIN”, “MAX”, “DEF” (1E-6), or “AUTO”.

**configure\_continuity**()

Configures the instrument to measure continuity.

**configure\_current**(*current\_range*='AUTO', *ac*=False, *resolution*='DEF')

Configures the instrument to measure current.

**Parameters**

- **current\_range** – A current in Amps to set the current range. DC values can be 100E-6, 1E-3, 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, “DEF” (100 mA), or “AUTO”. AC values can be 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, “DEF” (100 mA), or “AUTO”.
- **ac** – False for DC current, and True for AC current
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**configure\_diode**()

Configures the instrument to measure diode voltage.

**configure\_frequency**(*measured\_from*='voltage\_ac', *measured\_from\_range*='AUTO', *aperture*='DEF')

Configures the instrument to measure frequency.

**Parameters**

- **measured\_from** – “voltage\_ac” or “current\_ac”
- **measured\_from\_range** – range of measured\_from. AC voltage can have ranges 100E-3, 1, 10, 100, 750, as well as “MIN”, “MAX”, “DEF” (10 V), or “AUTO”. AC current can have ranges 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, “DEF” (100 mA), or “AUTO”.
- **aperture** – Aperture time in Seconds, can be 100 ms, 1 s, as well as “MIN”, “MAX”, or “DEF” (1 s).

**configure\_resistance**(*resistance\_range*='AUTO', *wires*=2, *resolution*='DEF')

Configures the instrument to measure resistance.

**Parameters**

- **resistance\_range** – A resistance in Ohms to set the resistance range, can be 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as “MIN”, “MAX”, “DEF” (1E3), or “AUTO”.
- **wires** – Number of wires used for measurement, can be 2 or 4.
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**configure\_temperature**()

Configures the instrument to measure temperature.

**configure\_voltage**(*voltage\_range*='AUTO', *ac*=False, *resolution*='DEF')

Configures the instrument to measure voltage.

**Parameters**

- **voltage\_range** – A voltage in Volts to set the voltage range. DC values can be 100E-3, 1, 10, 100, 1000, as well as “MIN”, “MAX”, “DEF” (10 V), or “AUTO”. AC values can be 100E-3, 1, 10, 100, 750, as well as “MIN”, “MAX”, “DEF” (10 V), or “AUTO”.
- **ac** – False for DC voltage, True for AC voltage
- **resolution** – Desired resolution, can be 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**property continuity**

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

**property current**

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

**property current\_ac**

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

**property current\_ac\_auto\_range**

A boolean property that toggles auto ranging for AC current.

**property current\_ac\_range**

A property that controls the AC current range in Amps, which can take values 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, or “DEF” (100 mA). Auto-range is disabled when this property is set.

**property current\_ac\_resolution**

An property that controls the resolution in the AC current readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**property current\_auto\_range**

A boolean property that toggles auto ranging for DC current.

**property current\_range**

A property that controls the DC current range in Amps, which can take values 100E-6, 1E-3, 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, or “DEF” (100 mA). Auto-range is disabled when this property is set.

**property current\_resolution**

A property that controls the resolution in the DC current readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, and “DEF” (3.00E-5).

**property diode**

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

**property frequency**

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

**property frequency\_aperture**

A property that controls the frequency aperture in seconds, which sets the integration period and measurement speed. Takes values 100 ms, 1 s, as well as “MIN”, “MAX”, or “DEF” (1 s).

**property frequency\_current\_auto\_range**

Boolean property that toggles auto ranging for AC current in frequency measurements.

**property frequency\_current\_range**

A property that controls the current range in Amps for frequency on AC current measurements, which can take values 10E-3, 100E-3, 1, 10, as well as “MIN”, “MAX”, or “DEF” (100 mA). Auto-range is disabled when this property is set.

**property frequency\_voltage\_auto\_range**

Boolean property that toggles auto ranging for AC voltage in frequency measurements.

**property frequency\_voltage\_range**

A property that controls the voltage range in Volts for frequency on AC voltage measurements, which can take values 100E-3, 1, 10, 100, 750, as well as “MIN”, “MAX”, or “DEF” (10 V). Auto-range is disabled when this property is set.

**property resistance**

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

**property resistance\_4w**

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

**property resistance\_4w\_auto\_range**

A boolean property that toggles auto ranging for 4-wire resistance.

**property resistance\_4w\_range**

A property that controls the 4-wire resistance range in Ohms, which can take values 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as “MIN”, “MAX”, or “DEF” (1E3). Auto-range is disabled when this property is set.

**property resistance\_4w\_resolution**

A property that controls the resolution in the 4-wire resistance readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**property resistance\_auto\_range**

A boolean property that toggles auto ranging for 2-wire resistance.

**property resistance\_range**

A property that controls the 2-wire resistance range in Ohms, which can take values 100, 1E3, 10E3, 100E3, 1E6, 10E6, 100E6, as well as “MIN”, “MAX”, or “DEF” (1E3). Auto-range is disabled when this property is set.

**property resistance\_resolution**

A property that controls the resolution in the 2-wire resistance readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**property temperature**

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

**property voltage**

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

**property voltage\_ac**

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

**property voltage\_ac\_auto\_range**

A boolean property that toggles auto ranging for AC voltage.

**property voltage\_ac\_range**

A property that controls the AC voltage range in Volts, which can take values 100E-3, 1, 10, 100, 750, as well as “MIN”, “MAX”, or “DEF” (10 V). Auto-range is disabled when this property is set.

**property voltage\_ac\_resolution**

A property that controls the resolution in the AC voltage readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

**property voltage\_auto\_range**

A boolean property that toggles auto ranging for DC voltage.

**property voltage\_range**

A property that controls the DC voltage range in Volts, which can take values 100E-3, 1, 10, 100, 1000, as well as “MIN”, “MAX”, or “DEF” (10 V). Auto-range is disabled when this property is set.

**property voltage\_resolution**

A property that controls the resolution in the DC voltage readings, which can take values 3.00E-5, 2.00E-5, 1.50E-6 (5 1/2 digits), as well as “MIN”, “MAX”, or “DEF” (1.50E-6).

## 7.7.7 Agilent 4155/4156 Semiconductor Parameter Analyzer

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

Bases: *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",
```

(continues on next page)

(continued from previous page)

```

        "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
    }
}

```

**property analyzer\_mode**

A string property that controls the instrument operating mode.

- Values: SWEEP, SAMPLING

```
smu.analyzer_mode = "SWEEP"
```

### **configure(*config\_file*)**

Configure the channel setup and sweep using a JSON configuration file.

(JSON is the [JavaScript Object Notation](#))

#### **Parameters**

**config\_file** – JSON file to configure instrument channels.

```
instr.configure('config.json')
```

### **property data\_variables**

Get a string list of data variables for which measured data is available.

This looks for all the variables saved by the [save\(\)](#) and [save\\_var\(\)](#) methods and returns it. This is useful for creation of dataframe headers.

#### **Returns**

List

```
header = instr.data_variables
```

### **property delay\_time**

A floating point property that measurement delay time in seconds, which can take the values from 0 to 65s in 0.1s steps.

```
instr.delay_time = 1 # delay time of 1-sec
```

### **disable\_all()**

Disables all channels in the instrument.

```
instr.disable_all()
```

### **get\_data(*path=None*)**

Get the measurement data from the instrument after completion.

If the measurement period is set to INF in the [measure\(\)](#) method, then the measurement must be stopped using [stop\(\)](#) before getting valid data.

#### **Parameters**

**path** – Path for optional data export to CSV.

#### **Returns**

Pandas Dataframe

```
df = instr.get_data(path='./datafolder/data1.csv')
```

### **property hold\_time**

A floating point property that measurement hold time in seconds, which can take the values from 0 to 655s in 1s steps.

```
instr.hold_time = 2 # hold time of 2-secs.
```

### **property integration\_time**

A string property that controls the integration time.

- Values: SHORT, MEDIUM, LONG

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

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

Performs a single measurement and waits for completion in sweep mode. In sampling mode, the measurement period and number of points can be specified.

**Parameters**

- **period** – Period of sampling measurement from 6E-6 to 1E11 seconds. Default setting is INF.
- **points** – Number of samples to be measured, from 1 to 10001. Default setting is 100.

**save**(*trace\_list*)

Save the voltage or current in the instrument display list

**Parameters**

**trace\_list** – A list of channel variables whose measured data should be saved. A maximum of 8 variables are allowed. If only one variable is being saved, a string can be specified.

```
instr.save(['IC', 'IB', 'VC', 'VB']) #for list of variables
instr.save('IC') #for single variable
```

**save\_var**(*trace\_list*)

Save the voltage or current in the instrument variable list.

This is useful if one or two more variables need to be saved in addition to the 8 variables allowed by [save\(\)](#).

**Parameters**

**trace\_list** – A list of channel variables whose measured data should be saved. A maximum of 2 variables are allowed. If only one variable is being saved, a string can be specified.

```
instr.save_var(['VA', 'VB'])
```

**stop**()

Stops the ongoing measurement

```
instr.stop()
```

**class** pymeasure.instruments.agilent.agilent4156.SMU(*adapter, channel, \*\*kwargs*)

Bases: [Instrument](#)

**property channel\_function**

A string property that controls the SMU<n> channel function.

- Values: VAR1, VAR2, VARD or CONS.

```
instr.smul.channel_function = "VAR1"
```

**property channel\_mode**

A string property that controls the SMU<n> channel mode.

- Values: V, I or COMM

VPULSE AND IPULSE are not yet supported.

```
instr.smul.channel_mode = "V"
```

### property compliance

Sets the *constant* compliance value of SMU<n>.

If the SMU channel is setup as a variable (VAR1, VAR2, VARD) then compliance limits are set by the variable definition.

- Value: Voltage in (-200V, 200V) and current in (-1A, 1A) based on `channel_mode()`.

```
instr.smul.compliance = 0.1
```

### property constant\_value

Set the constant source value of SMU<n>.

You use this command only if `channel_function()` is CONS and also `channel_mode()` should not be COMM.

#### Parameters

**const\_value** – Voltage in (-200V, 200V) and current in (-1A, 1A). Voltage or current depends on if `channel_mode()` is set to V or I.

```
instr.smul.constant_value = 1
```

### property current\_name

Define the current name of the channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.smul.current_name = "Ibase"
```

### property disable

Deletes the settings of SMU<n>.

```
instr.smul.disable()
```

### property series\_resistance

Controls the series resistance of SMU<n>.

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

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

### property voltage\_name

Define the voltage name of the channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.smul.voltage_name = "Vbase"
```

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

Bases: `VARX`

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

**property spacing**

Selects the sweep type of VAR1.

- Values: LINEAR, LOG10, LOG25, LOG50.

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

Bases: [VARX](#)

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

**property points**

Sets the number of sweep steps of VAR2. You use this command only if there is an SMU or VSU whose function (FCTN) is VAR2.

```
instr.var2.points = 10
```

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

Bases: [Instrument](#)

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

**property compliance**

Sets the sweep COMPLIANCE value of VARD.

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

**property offset**

Sets the OFFSET value of VARD. For each step of sweep, the output values of VAR1' are determined by the following equation:  $VARD = VAR1 \times RATIO + OFFSET$  You use this command only if there is an SMU or VSU whose function is VARD.

```
instr.vard.offset = 1
```

**property ratio**

Sets the RATIO of VAR1'. For each step of sweep, the output values of VAR1' are determined by the following equation:  $VAR1' = VAR1 \times RATIO + OFFSET$  You use this command only if there is an SMU or VSU whose function (FCTN) is VAR1'.

```
instr.vard.ratio = 1
```

**class** pymeasure.instruments.agilent.agilent4156.VARX(*adapter*, *var\_name*, *\*\*kwargs*)

Bases: [Instrument](#)

Base class to define sweep variable settings

**property compliance**

Sets the sweep COMPLIANCE value.

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

**property start**

Sets the sweep START value.

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

**property step**

Sets the sweep STEP value.

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

**property stop**

Sets the sweep STOP value.

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

**class** pymeasure.instruments.agilent.agilent4156.VMU(*adapter, channel, \*\*kwargs*)

Bases: *Instrument*

**property channel\_mode**

A string property that controls the VMU<n> channel mode.

- Values: V, DVOL

**property disable**

Disables the settings of VMU<n>.

```
instr.vmu1.disable()
```

**property voltage\_name**

Define the voltage name of the VMU channel.

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.vmu1.voltage_name = "Vnode"
```

**class** pymeasure.instruments.agilent.agilent4156.VSU(*adapter, channel, \*\*kwargs*)

Bases: *Instrument*

**property channel\_function**

A string property that controls the VSU channel function.

- Value: VAR1, VAR2, VARD or CONS.

**property channel\_mode**

Get channel mode of VSU<n>.

**property constant\_value**

Sets the constant source value of VSU<n>.

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

**property disable**

Deletes the settings of VSU<n>.

```
instr.vsu1.disable()
```

**property voltage\_name**

Define the voltage name of the VSU channel

If input is greater than 6 characters long or starts with a number, the name is autocorrected and prepended with 'a'. Event is logged.

```
instr.vsu1.voltage_name = "Ve"
```

### 7.7.8 Agilent 33220A Arbitrary Waveform Generator

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

Bases: *Instrument*

Represents the Agilent 33220A Arbitrary Waveform Generator.

```
# Default channel for the Agilent 33220A
wfg = Agilent33220A("GPIB::10")

wfg.shape = "SINUSOID"          # Sets a sine waveform
wfg.frequency = 4.7e3            # Sets the frequency to 4.7 kHz
wfg.amplitude = 1                # Set amplitude of 1 V
wfg.offset = 0                  # Set the amplitude to 0 V

wfg.burst_state = True           # Enable burst mode
wfg.burst_ncycles = 10          # A burst will consist of 10 cycles
wfg.burst_mode = "TRIGGERED"    # A burst will be applied on a trigger
wfg.trigger_source = "BUS"      # A burst will be triggered on TRG*

wfg.output = True               # Enable output of waveform generator
wfg.trigger()                   # Trigger a burst
wfg.wait_for_trigger()          # Wait until the triggering is finished
wfg.beep()                      # "beep"

print(wfg.check_errors())       # Get the error queue
```

#### property amplitude

A floating point property that controls the voltage amplitude of the output waveform in V, from 10e-3 V to 10 V. Can be set.

#### property amplitude\_unit

A string property that controls the units of the amplitude. Valid values are Vpp (default), Vrms, and dBm. Can be set.

#### beep()

Causes a system beep.

#### property beeper\_state

A boolean property that controls the state of the beeper. Can be set.

#### property burst\_mode

A string property that controls the burst mode. Valid values are: TRIG<GERED>, GAT<ED>. This setting can be set.

#### property burst\_ncycles

An integer property that sets the number of cycles to be output when a burst is triggered. Valid values are 1 to 50000. This can be set.

**property burst\_state**

A boolean property that controls whether the burst mode is on (True) or off (False). Can be set.

**property frequency**

A floating point property that controls the frequency of the output waveform in Hz, from 1e-6 (1 uHz) to 20e+6 (20 MHz), depending on the specified function. Can be set.

**property offset**

A floating point property that controls the voltage offset of the output waveform in V, from 0 V to 4.995 V, depending on the set voltage amplitude (maximum offset =  $(10 - \text{voltage}) / 2$ ). Can be set.

**property output**

A boolean property that turns on (True) or off (False) the output of the function generator. Can be set.

**property pulse\_dutycycle**

A floating point property that controls the duty cycle of a pulse waveform function in percent. Can be set.

**property pulse\_hold**

A string property that controls if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform. Can be set to: WIDT<H> or DCYC<LE>.

**property pulse\_period**

A floating point property that controls the period of a pulse waveform function in seconds, ranging from 200 ns to 2000 s. Can be set and overwrites the frequency for *all* waveforms. If the period is shorter than the pulse width + the edge time, the edge time and pulse width will be adjusted accordingly.

**property pulse\_transition**

A floating point property that controls the the edge time in seconds for both the rising and falling edges. It is defined as the time between 0.1 and 0.9 of the threshold. Valid values are between 5 ns to 100 ns. The transition time has to be smaller than  $0.625 * \text{the pulse width}$ . Can be set.

**property pulse\_width**

A floating point property that controls the width of a pulse waveform function in seconds, ranging from 20 ns to 2000 s, within a set of restrictions depending on the period. Can be set.

**property ramp\_symmetry**

A floating point property that controls the symmetry percentage for the ramp waveform. Can be set.

**property remote\_local\_state**

A string property that controls the remote/local state of the function generator. Valid values are: LOC<AL>, REM<OTE>, RWL<OCK>. This setting can only be set.

**property shape**

A string property that controls the output waveform. Can be set to: SIN<USOID>, SQU<ARE>, RAMP, PULS<E>, NOIS<E>, DC, USER.

**property square\_dutycycle**

A floating point property that controls the duty cycle of a square waveform function in percent. Can be set.

**trigger()**

Send a trigger signal to the function generator.

**property trigger\_source**

A string property that controls the trigger source. Valid values are: IMM<EDIATE> (internal), EXT<ERNAL> (rear input), BUS (via trigger command). This setting can be set.

**property trigger\_state**

A boolean property that controls whether the output is triggered (True) or not (False). Can be set.

**property voltage\_high**

A floating point property that controls the upper voltage of the output waveform in V, from -4.990 V to 5 V (must be higher than low voltage). Can be set.

**property voltage\_low**

A floating point property that controls the lower voltage of the output waveform in V, from -5 V to 4.990 V (must be lower than high voltage). Can be set.

**wait\_for\_trigger**(*timeout=3600, should\_stop=<function Agilent33220A.<lambda>>>*)

Wait until the triggering has finished or timeout is reached.

**Parameters**

- **timeout** – The maximum time the waiting is allowed to take. If timeout is exceeded, a TimeoutError is raised. If timeout is set to zero, no timeout will be used.
- **should\_stop** – Optional function (returning a bool) to allow the waiting to be stopped before its end.

## 7.7.9 Agilent 33500 Function/Arbitrary Waveform Generator Family

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

Bases: [Instrument](#)

Represents the Agilent 33500 Function/Arbitrary Waveform Generator family.

Individual devices are represented by subclasses. User can specify a channel to control, if no channel specified, a default channel is picked based on the device e.g. For Agilent33500B the default channel is channel 1. See reference manual for your device

```
generator = Agilent33500("GPIB::1")

generator.shape = 'SIN'           # Sets default channel output signal shape_
↪to sine
generator.ch_1.shape = 'SIN'      # Sets channel 1 output signal shape to sine
generator.frequency = 1e3         # Sets default channel output frequency to_
↪1 kHz
generator.ch_1.frequency = 1e3    # Sets channel 1 output frequency to 1 kHz
generator.ch_2.amplitude = 5      # Sets channel 2 output amplitude to 5 Vpp
generator.ch_2.output = 'on'      # Enables channel 2 output

generator.ch_1.shape = 'ARB'      # Set channel 1 shape to arbitrary
generator.ch_1.arb_srate = 1e6    # Set channel 1 sample rate to 1MSa/s

generator.ch_1.data_volatile_clear() # Clear channel 1 volatile internal memory
generator.ch_1.data_arb(          # Send data of arbitrary waveform to channel_
↪1
    'test',
    range(-10000, 10000, +20),    # In this case a simple ramp
    data_format='DAC'            # Data format is set to 'DAC'
)
generator.ch_1.arb_file = 'test'  # Select the transmitted waveform 'test'
```

**ch\_1**

**Channel**

*Agilent33500Channel*

**ch\_2**

**Channel**

*Agilent33500Channel*

**property amplitude**

A floating point property that controls the voltage amplitude of the output waveform in V, from 10e-3 V to 10 V. Depends on the output impedance.

**property amplitude\_unit**

A string property that controls the units of the amplitude. Valid values are VPP (default), VRMS, and DBM.

**property arb\_advance**

A string property that selects how the device advances from data point to data point. Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

**property arb\_file**

A string property that selects the arbitrary signal from the volatile memory of the device. String has to match an existing arb signal in volatile memory (set by *data\_arb()*).

**property arb\_filter**

A string property that selects the filter setting for arbitrary signals. Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

**property arb\_srate**

An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1  $\mu$ Sa/s to 250 MSa/s (maximum range, can be lower depending on your device).

**beep()**

Causes a system beep.

**property burst\_mode**

A string property that controls the burst mode. Valid values are: TRIG<GERED>, GAT<ED>.

**property burst\_ncycles**

An integer property that sets the number of cycles to be output when a burst is triggered. Valid values are 1 to 100000. This can be set.

**property burst\_period**

A floating point property that controls the period of subsequent bursts. Has to follow the equation  $\text{burst\_period} > (\text{burst\_ncycles} / \text{frequency}) + 1 \mu\text{s}$ . Valid values are 1  $\mu\text{s}$  to 8000 s.

**property burst\_state**

A boolean property that controls whether the burst mode is on (True) or off (False).

**clear\_display()**

Removes a text message from the display.

**data\_arb(arb\_name, data\_points, data\_format='DAC')**

Uploads an arbitrary trace into the volatile memory of the device.

The data\_points can be given as: comma separated 16 bit DAC values (ranging from -32767 to +32767), as comma separated floating point values (ranging from -1.0 to +1.0) or as a binary data stream. Check

the manual for more information. The storage depends on the device type and ranges from 8 Sa to 16 MSa (maximum).

#### Parameters

- **arb\_name** – The name of the trace in the volatile memory. This is used to access the trace.
- **data\_points** – Individual points of the trace. The format depends on the format parameter. format = 'DAC' (default): Accepts list of integer values ranging from -32767 to +32767. Minimum of 8 a maximum of 65536 points. format = 'float': Accepts list of floating point values ranging from -1.0 to +1.0. Minimum of 8 a maximum of 65536 points. format = 'binary': Accepts a binary stream of 8 bit data.
- **data\_format** – Defines the format of data\_points. Can be 'DAC' (default), 'float' or 'binary'. See documentation on parameter data\_points above.

#### data\_volatile\_clear()

Clear all arbitrary signals from volatile memory.

This should be done if the same name is used continuously to load different arbitrary signals into the memory, since an error will occur if a trace is loaded which already exists in the memory.

#### property display

A string property which is displayed on the front panel of the device.

#### property ext\_trig\_out

A boolean property that controls whether the trigger out signal is active (True) or not (False). This signal is output from the Ext Trig connector on the rear panel in Burst and Wobbel mode.

#### property frequency

A floating point property that controls the frequency of the output waveform in Hz, from 1 uHz to 120 MHz (maximum range, can be lower depending on your device), depending on the specified function.

#### property offset

A floating point property that controls the voltage offset of the output waveform in V, from 0 V to 4.995 V, depending on the set voltage amplitude (maximum offset =  $(V_{\text{max}} - \text{voltage}) / 2$ ).

#### property output

A boolean property that turns on (True, 'on') or off (False, 'off') the output of the function generator.

#### property output\_load

Sets the expected load resistance (should be the load impedance connected to the output. The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm. Valid values are between 1 and 10 kOhm or INF for high impedance. No validator is used since both numeric and string inputs are accepted, thus a value outside the range will not return an error.

#### property phase

A floating point property that controls the phase of the output waveform in degrees, from -360 degrees to 360 degrees. Not available for arbitrary waveforms or noise.

#### phase\_sync()

Synchronize the phase of all channels.

#### property pulse\_dutycycle

A floating point property that controls the duty cycle of a pulse waveform function in percent, from 0% to 100%.

**property pulse\_hold**

A string property that controls if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform. Can be set to: WIDT<H> or DCYC<LE>.

**property pulse\_period**

A floating point property that controls the period of a pulse waveform function in seconds, ranging from 33 ns to 1e6 s. Can be set and overwrites the frequency for *all* waveforms. If the period is shorter than the pulse width + the edge time, the edge time and pulse width will be adjusted accordingly.

**property pulse\_transition**

A floating point property that controls the edge time in seconds for both the rising and falling edges. It is defined as the time between the 10% and 90% thresholds of the edge. Valid values are between 8.4 ns to 1  $\mu$ s.

**property pulse\_width**

A floating point property that controls the width of a pulse waveform function in seconds, ranging from 16 ns to 1 Ms, within a set of restrictions depending on the period.

**property ramp\_symmetry**

A floating point property that controls the symmetry percentage for the ramp waveform, from 0.0% to 100.0%.

**property shape**

A string property that controls the output waveform. Can be set to: SIN<USOID>, SQU<ARE>, TRI<ANGLE>, RAMP, PULS<E>, PRBS, NOIS<E>, ARB, DC.

**property square\_dutycycle**

A floating point property that controls the duty cycle of a square waveform function in percent, from 0.01% to 99.98%. The duty cycle is limited by the frequency and the minimal pulse width of 16 ns. See manual for more details.

**trigger()**

Send a trigger signal to the function generator.

**property trigger\_source**

A string property that controls the trigger source. Valid values are: IMM<EDIATE> (internal), EXT<ERNAL> (rear input), BUS (via trigger command).

**property voltage\_high**

A floating point property that controls the upper voltage of the output waveform in V, from -4.999 V to 5 V (must be higher than low voltage by at least 1 mV).

**property voltage\_low**

A floating point property that controls the lower voltage of the output waveform in V, from -5 V to 4.999 V (must be lower than high voltage by at least 1 mV).

**wait\_for\_trigger**(*timeout=3600, should\_stop=<function Agilent33500.<lambda>>*)

Wait until the triggering has finished or timeout is reached.

**Parameters**

- **timeout** – The maximum time the waiting is allowed to take. If timeout is exceeded, a `TimeoutError` is raised. If timeout is set to zero, no timeout will be used.
- **should\_stop** – Optional function (returning a bool) to allow the waiting to be stopped before its end.

### 7.7.10 Agilent 33521A Function/Arbitrary Waveform Generator

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

Bases: [`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  $\mu$ Sa/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**

A floating point property that controls the voltage amplitude of the output waveform in V, from 10e-3 V to 10 V. Depends on the output impedance.

**property amplitude\_unit**

A string property that controls the units of the amplitude. Valid values are VPP (default), VRMS, and DBM.

**property arb\_advance**

A string property that selects how the device advances from data point to data point. Can be set to 'TRIG<GER>' or 'SRAT<E>' (default).

**property arb\_file**

A string property that selects the arbitrary signal from the volatile memory of the device. String has to match an existing arb signal in volatile memory (set by [`data\_arb\(\)`](#)).

**property arb\_filter**

A string property that selects the filter setting for arbitrary signals. Can be set to 'NORM<AL>', 'STEP' and 'OFF'.

**property arb\_srate**

An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1  $\mu$ Sa/s to 250 MSa/s (maximum range, can be lower depending on your device).

**property burst\_mode**

A string property that controls the burst mode. Valid values are: TRIG<GERED>, GAT<ED>.

**property burst\_ncycles**

An integer property that sets the number of cycles to be output when a burst is triggered. Valid values are 1 to 100000. This can be set.

**property burst\_period**

A floating point property that controls the period of subsequent bursts. Has to follow the equation  $\text{burst\_period} > (\text{burst\_ncycles} / \text{frequency}) + 1 \mu\text{s}$ . Valid values are 1  $\mu\text{s}$  to 8000 s.

**property burst\_state**

A boolean property that controls whether the burst mode is on (True) or off (False).

**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**

A floating point property that controls the frequency of the output waveform in Hz, from 1 uHz to 120 MHz (maximum range, can be lower depending on your device), depending on the specified function.

**property offset**

A floating point property that controls the voltage offset of the output waveform in V, from 0 V to 4.995 V, depending on the set voltage amplitude (maximum offset =  $(V_{\text{max}} - \text{voltage}) / 2$ ).

**property output**

A boolean property that turns on (True, 'on') or off (False, 'off') the output of the function generator.

**property output\_load**

Sets the expected load resistance (should be the load impedance connected to the output. The output impedance is always 50 Ohm, this setting can be used to correct the displayed voltage for loads unmatched to 50 Ohm. Valid values are between 1 and 10 kOhm or INF for high impedance. No validator is used since both numeric and string inputs are accepted, thus a value outside the range will not return an error.

**property phase**

A floating point property that controls the phase of the output waveform in degrees, from -360 degrees to 360 degrees. Not available for arbitrary waveforms or noise.

**property pulse\_dutycycle**

A floating point property that controls the duty cycle of a pulse waveform function in percent, from 0% to 100%.

**property pulse\_hold**

A string property that controls if either the pulse width or the duty cycle is retained when changing the period or frequency of the waveform. Can be set to: WIDT<H> or DCYC<LE>.

**property pulse\_period**

A floating point property that controls the period of a pulse waveform function in seconds, ranging from 33 ns to 1 Ms. Can be set and overwrites the frequency for *all* waveforms. If the period is shorter than the pulse width + the edge time, the edge time and pulse width will be adjusted accordingly.

**property pulse\_transition**

A floating point property that controls the edge time in seconds for both the rising and falling edges. It is defined as the time between the 10% and 90% thresholds of the edge. Valid values are between 8.4 ns to 1  $\mu$ s.

**property pulse\_width**

A floating point property that controls the width of a pulse waveform function in seconds, ranging from 16 ns to 1e6 s, within a set of restrictions depending on the period.

**property ramp\_symmetry**

A floating point property that controls the symmetry percentage for the ramp waveform, from 0.0% to 100.0%.

**property shape**

A string property that controls the output waveform. Can be set to: SIN<USOID>, SQU<ARE>, TRI<ANGLE>, RAMP, PULS<E>, PRBS, NOIS<E>, ARB, DC.

**property square\_dutycycle**

A floating point property that controls the duty cycle of a square waveform function in percent, from 0.01% to 99.98%. The duty cycle is limited by the frequency and the minimal pulse width of 16 ns. See manual for more details.

**property voltage\_high**

A floating point property that controls the upper voltage of the output waveform in V, from -4.999 V to 5 V (must be higher than low voltage by at least 1 mV).

**property voltage\_low**

A floating point property that controls the lower voltage of the output waveform in V, from -5 V to 4.999 V (must be lower than high voltage by at least 1 mV).

## 7.7.11 Agilent B1500 Semiconductor Parameter Analyzer

### Contents

- *Agilent B1500 Semiconductor Parameter Analyzer*
  - *General Information*
    - \* *Command Translation*
  - *Examples*
    - \* *Initialization of the Instrument*
    - \* *IV measurement with 4 SMUs*
    - \* *Sampling measurement with 4 SMUs*
  - *Main Classes*
  - *Supporting Classes*
    - \* *Enumerations*

### 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	<code>SMU.adc_type()</code>
AB	<code>abort()</code>
AIT	<code>adc_setup()</code>
AV	<code>adc_averaging()</code>
AZ	<code>adc_auto_zero</code>
BC	<code>clear_buffer()</code>
CL	<code>SMU.disable()</code>
CM	<code>auto_calibration</code>
CMM	<code>SMU.meas_op_mode()</code>
CN	<code>SMU.enable()</code>
DI	<code>SMU.force()</code> mode: 'CURRENT'
DV	<code>SMU.force()</code> mode: 'VOLTAGE'
DZ	<code>force_gnd()</code> , <code>SMU.force_gnd()</code>
ERRX?	<code>check_errors()</code>
FL	<code>SMU.filter</code>

continues on next page

Table 1 – continued from previous page

Command	Property/Method
FMT	<code>data_format()</code>
*IDN?	<code>id()</code>
*LRN?	<code>query_learn()</code> , multiple methods to read/format settings directly
MI	<code>SMU.sampling_source()</code> mode: 'CURRENT'
ML	<code>sampling_mode</code>
MM	<code>meas_mode()</code>
MSC	<code>sampling_auto_abort()</code>
MT	<code>sampling_timing()</code>
MV	<code>SMU.sampling_source()</code> mode: 'VOLTAGE'
*OPC?	<code>check_idle()</code>
PA	<code>pause()</code>
PAD	<code>parallel_meas</code>
RI	<code>meas_range_current</code>
RM	<code>SMU.meas_range_current_auto()</code>
*RST	<code>reset()</code>
RV	<code>meas_range_voltage</code>
SSR	<code>series_resistor</code>
TSC	<code>time_stamp</code>
TSR	<code>clear_timer()</code>
UNT?	<code>query_modules()</code>
WAT	<code>wait_time()</code>
WI	<code>SMU.staircase_sweep_source()</code> mode: 'CURRENT'
WM	<code>sweep_auto_abort()</code>
WSI	<code>SMU.synchronous_sweep_source()</code> mode: 'CURRENT'
WSV	<code>SMU.synchronous_sweep_source()</code> mode: 'VOLTAGE'
WT	<code>sweep_timing()</code>
WV	<code>SMU.staircase_sweep_source()</code> mode: 'VOLTAGE'
XE	<code>send_trigger()</code>

## 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=6000000)
# 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-resolution ADC
    smu.meas_range_current = '1 nA'
    smu.meas_op_mode = 'COMPLIANCE_SIDE' # other choices: Current, Voltage, FORCE_SIDE,
↳ COMPLIANCE_AND_FORCE_SIDE

# General Instrument Settings
# b1500.adc_averaging = 1
# b1500.adc_auto_zero = True
b1500.adc_setup('HRADC', 'AUTO', 6)
#b1500.adc_setup('HRADC', 'PLC', 1)

#Sweep Settings
b1500.sweep_timing(0, 5, step_delay=0.1) #hold, delay
b1500.sweep_auto_abort(False, post='STOP') #disable auto abort, set post measurement
↳ output condition to stop value of sweep
# Sweep Source
nop = 11
b1500.smu1.staircase_sweep_source('VOLTAGE', 'LINEAR_DOUBLE', 'Auto Ranging', 0, 1, nop, 0.
↳ 001) #type, mode, range, start, stop, steps, compliance
# Synchronous Sweep Source
b1500.smu2.synchronous_sweep_source('VOLTAGE', 'Auto Ranging', 0, 1, 0.001) #type, range,
↳ start, stop, comp
# Constant Output (could also be done using synchronous sweep source with start=stop,
↳ but then the output is not ramped up)
b1500.smu3.ramp_source('VOLTAGE', 'Auto Ranging', -1, stepsize=0.1, pause=20e-3) #output
↳ starts immediately! (compared to sweeps)
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)
```

(continues on next page)

(continued from previous page)

```

    # process live data for plotting etc.
    # data format for every channel (status code, channel name e.g. 'SMU1', data name e.g
    ↪ 'Current Measurement (A)', value)
    meas.append(read_data)

#sweep constant sources back to 0V
b1500.smu3.ramp_source('VOLTAGE', 'Auto Ranging', 0, stepsize=0.1, pause=20e-3)
b1500.smu4.ramp_source('VOLTAGE', 'Auto Ranging', 0, stepsize=0.1, pause=20e-3)

```

## Sampling measurement with 4 SMUs

```

# choose measurement mode
b1500.meas_mode('SAMPLING', *b1500.smu_references) #order in smu_references determines
↪ order of measurement
number_of_channels = len(b1500.smu_references)

# settings for individual SMUs
for smu in b1500.smu_references:
    smu.enable() #enable SMU
    smu.adc_type = 'HSADC' #set ADC to high-speed ADC
    smu.meas_range_current = '1 nA'
    smu.meas_op_mode = 'COMPLIANCE_SIDE' # other choices: Current, Voltage, FORCE_SIDE,
    ↪ COMPLIANCE_AND_FORCE_SIDE

b1500.sampling_mode = 'LINEAR'
# b1500.adc_averaging = 1
# b1500.adc_auto_zero = True
b1500.adc_setup('HSADC', 'AUTO', 1)
#b1500.adc_setup('HSADC', 'PLC', 1)
nop=11
b1500.sampling_timing(2, 0.005, nop) #MT: bias hold time, sampling interval, number of
↪ points
b1500.sampling_auto_abort(False, post='BIAS') #MSC: BASE/BIAS
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=[]

```

(continues on next page)

(continued from previous page)

```

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: *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. Halfs the integration time, if off. (AZ)

**Type**

bool

**property time\_stamp**

Enable/Disable Time Stamp function. (TSC)

**Type**

bool

**query\_time\_stamp\_setting**()

Read time stamp settings (60) from the instrument.

**wait\_time**(*wait\_type*, *N*, *offset*=0)

Configure wait time. (WAT)

**Parameters**

- **wait\_type** (*WaitTimeType*) – Wait time type
- **N** (*float*) – Coefficient for initial wait time, default: 1
- **offset** (*int*, optional) – Offset for wait time, defaults to 0

**query\_staircase\_sweep\_settings**()

Reads Staircase Sweep Measurement settings (33) from the instrument.

**sweep\_timing**(*hold*, *delay*, *step\_delay*=0, *step\_trigger\_delay*=0, *measurement\_trigger\_delay*=0)

Sets Hold Time, Delay Time and Step Delay Time for staircase or multi channel sweep measurement. (WT)  
If not set, all parameters are 0.

**Parameters**

- **hold** (*float*) – Hold time
- **delay** (*float*) – Delay time
- **step\_delay** (*float*, optional) – Step delay time, defaults to 0
- **step\_trigger\_delay** (*float*, optional) – Trigger delay time, defaults to 0
- **measurement\_trigger\_delay** (*float*, optional) – Measurement trigger delay time, defaults to 0

**sweep\_auto\_abort**(*abort*, *post*='START')

Enables/Disables the automatic abort function. Also sets the post measurement condition. (WM)

**Parameters**

- **abort** (*bool*) – Enable/Disable automatic abort

- **post** (*StaircaseSweepPostOutput*, optional) – Output after measurement, defaults to ‘Start’

**query\_sampling\_settings()**

Reads Sampling Measurement settings (47) from the instrument.

**property sampling\_mode**

Set linear or logarithmic sampling mode. (ML)

**Type**

*SamplingMode*

**sampling\_timing**(*hold\_bias*, *interval*, *number*, *hold\_base*=0)

Sets Timing Parameters for the Sampling Measurement (MT)

**Parameters**

- **hold\_bias** (*float*) – Bias hold time
- **interval** (*float*) – Sampling interval
- **number** (*int*) – Number of Samples
- **hold\_base** (*float*, *optional*) – Base hold time, defaults to 0

**sampling\_auto\_abort**(*abort*, *post*='Bias')

Enables/Disables the automatic abort function. Also sets the post measurement condition. (MSC)

**Parameters**

- **abort** (*bool*) – Enable/Disable automatic abort
- **post** (*SamplingPostOutput*, optional) – Output after measurement, defaults to ‘Bias’

**read\_data**(*number\_of\_points*)

Reads all data from buffer and returns Pandas DataFrame. Specify number of measurement points for correct splitting of the data list.

**Parameters**

**number\_of\_points** (*int*) – Number of measurement points

**Returns**

Measurement Data

**Return type**

pd.DataFrame

**read\_channels**(*nchannels*)

Reads data for 1 measurement point from the buffer. Specify number of measurement channels + sweep sources (depending on data output setting).

**Parameters**

**nchannels** (*int*) – Number of channels which return data

**Returns**

Measurement data

**Return type**

tuple

**query\_series\_resistor()**

Read series resistor status (53) for all SMUs.

**query\_meas\_range\_current\_auto()**

Read auto ranging mode status (54) for all SMUs.

**query\_meas\_op\_mode()**

Read SMU measurement operation mode (46) for all SMUs.

**query\_meas\_ranges()**

Read measruement ranging status (32) for all SMUs.

**class** pymeasure.instruments.agilent.agilentB1500.SMU(*parent, channel, smu\_type, name, \*\*kwargs*)

Bases: object

Provides specific methods for the SMUs of the Agilent B1500 mainframe

**Parameters**

- **parent** (*AgilentB1500*) – Instance of the B1500 mainframe class
- **channel** (*int*) – Channel number of the SMU
- **smu\_type** (*str*) – Type of the SMU
- **name** (*str*) – Name of the SMU

**write**(*string*)

Wraps *Instrument.write()* method of B1500.

**ask**(*string*)

Wraps ask() method of B1500.

**query\_learn**(*query\_type, command*)

Wraps *query\_learn()* method of B1500.

**check\_errors()**

Wraps *check\_errors()* method of B1500.

**property status**

Query status of the SMU.

**enable()**

Enable Source/Measurement Channel (CN)

**disable()**

Disable Source/Measurement Channel (CL)

**force\_gnd()**

Force 0V immediately. Current Settings can be restored with RZ (not implemented). (DZ)

**property filter**

Enables/Disables SMU Filter. (FL)

**Type**

bool

**property series\_resistor**

Enables/Disables 1MOhm series resistor. (SSR)

**Type**

bool

**property meas\_op\_mode**

Set SMU measurement operation mode. (CMM)

**Type**

*MeasOpMode*

**property adc\_type**

ADC type of individual measurement channel. (AAD)

**Type**

*ADCType*

**force**(*source\_type*, *source\_range*, *output*, *comp*="", *comp\_polarity*="", *comp\_range*="")

Applies DC Current or Voltage from SMU immediately. (DI, DV)

**Parameters**

- **source\_type** (*str*) – Source type ('Voltage', 'Current')
- **source\_range** (*int* or *str*) – Output range index or name
- **output** – Source output value in A or V
- **comp** (*float*, *optional*) – Compliance value, defaults to previous setting
- **comp\_polarity** (*CompliancePolarity*) – Compliance polarity, 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 polarity, 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 indices)
- **ranges** (*dict*) – All range names {Name: Indices}
- **fixed\_ranges** – add fixed ranges (negative indices); 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=None, *,
                                                                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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *,
                                                             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=None, *,
                                                                      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=None, *,
                                                                              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=None, *,
                                                                    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=None, *,
                                                                module=None, qualname=None,
                                                                type=None, start=1,
                                                                boundary=None)
```

Bases: *CustomIntEnum*

Wait time type

**SMU\_SOURCE** = 1

**SMU\_MEASUREMENT** = 2

**CMU\_MEASUREMENT** = 3

## 7.8 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](#)*.

### 7.8.1 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 communication protocol of the device 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=0*, *\*\*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.

## 7.9 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](#).

### 7.9.1 Ametek 7270 DSP Lockin Amplifier

```
class pymeasure.instruments.ametek.Ametek7270(adapter, name='Ametek DSP 7270',
                                              read_termination='\x00', write_termination='\x00',
                                              **kwargs)
```

Bases: [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**

Reads the input value of ADC1 in Volts

**property adc2**

Reads the input value of ADC2 in Volts

**property adc3**

Reads the input value of ADC3 in Volts

**property adc4**

Reads the input value of ADC4 in Volts

**ask(command, query\_delay=0)**

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.

**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**

A floating point property that represents the output value on DAC1 in Volts. This property can be set.

**property dac2**

A floating point property that represents the output value on DAC2 in Volts. This property can be set.

**property dac3**

A floating point property that represents the output value on DAC3 in Volts. This property can be set.

**property dac4**

A floating point property that represents the output value on DAC4 in Volts. This property can be set.

**property frequency**

A floating point property that represents the lock-in frequency in Hz. This property can be set.

**property harmonic**

An integer property that represents the reference harmonic mode control, taking values from 1 to 127. This property can be set.

**property id**

Get the instrument ID and firmware version

**property mag**

Reads the magnitude in Volts

**property phase**

A floating point property that represents the reference harmonic phase in degrees. This property can be set.

**property sensitivity**

A floating point property that controls the sensitivity range in Volts, which can take discrete values from 2 nV to 1 V. This property can be set. (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**

A integer property that controls the filter slope in dB/octave, which can take the values 6, 12, 18, or 24 dB/octave. This property can be set.

**property theta**

Reads the signal phase in degrees

**property time\_constant**

A floating point property that controls the time constant in seconds, which takes values from 10 microseconds to 100,000 seconds. This property can be set.

**property voltage**

A floating point property that represents the voltage in Volts. This property can be set.

**property x**

Reads the X value in Volts

**property x1**

Reads the first harmonic X value in Volts

**property x2**

Reads the second harmonic X value in Volts

**property xy**

Reads both the X and Y values in Volts

**property y**

Reads the Y value in Volts

**property y1**

Reads the first harmonic Y value in Volts

**property y2**

Reads the second harmonic Y value in Volts

## 7.10 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*.

### 7.10.1 AMI 430 Power Supply

**class** pymeasure.instruments.ami.AMI430(*adapter*, *name*='AMI superconducting magnet power supply.',  
\*\**kwargs*)

Bases: *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
```

(continues on next page)

(continued from previous page)

```

magnet.pause                # Pauses the ramping
magnet.status               # Returns the status of the magnet

magnet.ramp_to_current(5)   # Ramps the current to 5 A

magnet.shutdown()           # Ramps the current to zero and disables_
↪ output

```

**property coilconst**

A floating point property that sets the coil constant in kGauss/A.

**disable\_persistent\_switch()**

Disables the persistent switch.

**enable\_persistent\_switch()**

Enables the persistent switch.

**property field**

Reads the field in kGauss of the magnet.

**has\_persistent\_switch\_enabled()**

Returns a boolean if the persistent switch is enabled.

**property magnet\_current**

Reads the current in Amps of the magnet.

**pause()**

Pauses the ramping of the magnetic field.

**ramp()**

Initiates the ramping of the magnetic field to set current/field with ramping rate previously set.

**property ramp\_rate\_current**

A floating point property that sets the current ramping rate in A/s.

**property ramp\_rate\_field**

A floating point property that sets the field ramping rate in kGauss/s.

**ramp\_to\_current(current, rate)**

Heats up the persistent switch and ramps the current with set ramp rate.

**ramp\_to\_field(field, rate)**

Heats up the persistent switch and ramps the current with set ramp rate.

**shutdown(ramp\_rate=0.0357)**

Turns on the persistent switch, ramps down the current to zero, and turns off the persistent switch.

**property state**

Reads the field in kGauss of the magnet.

**property supply\_current**

Reads the current in Amps of the power supply.

**property target\_current**

A floating point property that sets the target current in A for the magnet.

**property target\_field**

A floating point property that sets the target field in kGauss for the magnet.

**property voltage\_limit**

A floating point property that sets the voltage limit for charging/discharging the magnet.

**wait\_for\_holding**(*should\_stop=<function AMI430.<lambda>>, timeout=800, interval=0.1*)

**zero()**

Initiates the ramping of the magnetic field to zero current/field with ramping rate previously set.

## 7.11 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*.

### 7.11.1 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.

## 7.12 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](#).

### 7.12.1 Anapico APSIN12G Signal Generator

```
class pymeasure.instruments.anapico.APSIN12G(adapter, name='Anapico APSIN12G Signal Generator',
                                             **kwargs)
```

Bases: [\*Instrument\*](#)

Represents the Anapico APSIN12G Signal Generator with option 9K, HP and GPIB.

**property blanking**

A string property that represents the blanking of output power when frequency is changed. ON makes the output to be blanked (off) while changing frequency. This property can be set.

**disable\_rf()**

Disables the RF output.

**enable\_rf()**

Enables the RF output.

**property frequency**

A floating point property that represents the output frequency in Hz. This property can be set.

**property power**

A floating point property that represents the output power in dBm. This property can be set.

**property reference\_output**

A string property that represents the 10MHz reference output from the synth. This property can be set.

## 7.13 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\*](#).

### 7.13.1 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**

Perform a single capacitance, loss measurement and return the values in units of pF and nS. The used measurement voltage is returned as third value.

**property config**

Read out configuration

**trigger()**

Triggers a new measurement without blocking and waiting for the return value.

**triggered\_caplossvolt()**

reads the measurement value after the device was triggered by the trigger function.

**property `vhighest`**

maximum RMS value of the used measurement voltage. Values of up to 15 V are allowed. The device will select the best suiting range below the given value.

### 7.13.2 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`**

Perform a single capacitance, loss measurement and return the values in units of pF and nS. The used measurement voltage is returned as third value.

**`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`**

Read out configuration

**property `frequency`**

test frequency used for the measurements. Allowed are values between 50 and 20000 Hz. The device selects the closest possible frequency to the given value.

**property id**

Reads the instrument identification

**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**

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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write(command, \*\*kwargs)**

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command*, *values*, \**args*, \*\**kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (\**args*,) – Further arguments to hand to the Adapter.

**write\_bytes**(*content*, \*\**kwargs*)

Write the bytes *content* to the instrument.

## 7.14 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](#).

### 7.14.1 Anritsu MG3692C Signal Generator

**class** pymeasure.instruments.anritsu.**AnritsuMG3692C**(*adapter*, *name*='Anritsu MG3692C Signal Generator', \*\**kwargs*)

Bases: [Instrument](#)

Represents the Anritsu MG3692C Signal Generator

**disable**()

Disables the signal output.

**enable**()

Enables the signal output.

**property frequency**

A floating point property that represents the output frequency in Hz. This property can be set.

**property output**

A boolean property that represents the signal output state. This property can be set to control the output.

**property power**

A floating point property that represents the output power in dBm. This property can be set.

**shutdown**()

Shuts down the instrument, putting it in a safe state.

### 7.14.2 Anritsu MS9710C Optical Spectrum Analyzer

**class** pymeasure.instruments.anritsu.**AnritsuMS9710C**(*adapter*, *name*='Anritsu MS9710C Optical Spectrum Analyzer', \*\**kwargs*)

Bases: [Instrument](#)

Anritsu MS9710C Optical Spectrum Analyzer.

**property analysis**

Analysis Control

**property analysis\_result**

Read back analysis result from current scan.

**property average\_point**

Number of averages to take on each point (2-1000), or OFF

**property average\_sweep**

Number of averages to make on a sweep (2-1000) or OFF

**center\_at\_peak(\*\*kwargs)**

Center the spectrum at the measured peak.

**property data\_memory\_a\_condition**

Returns the data condition of data memory register A. Starting wavelength, and a sampling point (l1, l2, n).

**property data\_memory\_a\_size**

Returns the number of points sampled in data memory register A.

**property data\_memory\_a\_values**

Reads the binary data from memory register A.

**property data\_memory\_b\_condition**

Returns the data condition of data memory register B. Starting wavelength, and a sampling point (l1, l2, n).

**property data\_memory\_b\_size**

Returns the number of points sampled in data memory register B.

**property data\_memory\_b\_values**

Reads the binary data from memory register B.

**property data\_memory\_select**

Memory Data Select.

**property dip\_search**

Dip Search Mode

**property ese2**

Extended Event Status Enable Register 2

**property esr2**

Extended Event Status Register 2

**property level\_lin**

Level Linear Scale (/div)

**property level\_log**

Level Log Scale (/div)

**property level\_opt\_attn**

Optical Attenuation Status (ON/OFF)

**property level\_scale**

Current Level Scale

**property measure\_mode**

Returns the current Measure Mode the OSA is in.

**measure\_peak()**

Measure the peak and return the trace marker.

**property peak\_search**

Peak Search Mode

**read\_memory(slot='A')**

Read the scan saved in a memory slot.

**property resolution**

Resolution (nm)

**property resolution\_actual**

Resolution Actual (ON/OFF)

**property resolution\_vbw**

Video Bandwidth Resolution

**property sampling\_points**

Number of sampling points

**single\_sweep(\*\*kwargs)**

Perform a single sweep and wait for completion.

**property trace\_marker**

Sets the trace marker with a wavelength. Returns the trace wavelength and power.

**property trace\_marker\_center**

Trace Marker at Center. Set to 1 or True to initiate command

**wait(n=3, delay=1)**

Query OPC Command and waits for appropriate response.

**wait\_for\_sweep(n=20, delay=0.5)**

Wait for a sweep to stop.

This is performed by checking bit 1 of the ESR2.

**property wavelength\_center**

Center Wavelength of Spectrum Scan in nm.

**property wavelength\_marker\_value**

Wavelength Marker Value (wavelength or freq.?)

**property wavelength\_span**

Wavelength Span of Spectrum Scan in nm.

**property wavelength\_start**

Wavelength Start of Spectrum Scan in nm.

**property wavelength\_stop**

Wavelength Stop of Spectrum Scan in nm.

**property wavelength\_value\_in**

Wavelength value in Vacuum or Air

**property wavelengths**

Return a numpy array of the current wavelengths of scans.

### 7.14.3 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**

Nr. of averages to make on a sweep (1-1000), with 1 being a single (non-averaged) sweep

**property data\_memory\_select**

Memory Data Select.

**repeat\_sweep**(*n=20, delay=0.5*)

Perform a single sweep and wait for completion.

**property resolution**

Resolution (nm)

**property resolution\_vbw**

Video Bandwidth Resolution

**property sampling\_points**

Number of sampling points

### 7.14.4 Anritsu MS2090A Handheld Spectrum Analyzer

```
class pymeasure.instruments.anritsu.AnritsuMS2090A(adapter, name='Anritsu MS2090A Handheld  
Spectrum Analyzer', **kwargs)
```

Bases: [\*Instrument\*](#)

Anritsu MS2090A Handheld Spectrum Analyzer.

**abort**()

Initiate a sweep/measurement.

**property active\_state**

The “set” state indicates that the instrument is used by someone.

**property external\_current**

This command queries the actual bias current in A

**property fetch\_control**

Returns the Control Channel measurement in json format.

**property fetch\_density**

Returns the most recent channel density measurement

**property fetch\_eirpower**

Returns the current EIRP, Max EIRP, Horizontal EIRP, Vertical and Sum EIRP results in dBm.

**property fetch\_eirpower\_data**

This command returns the current EIRP measurement result in dBm.

**property fetch\_eirpower\_max**

This command returns the Max EIRP measurement result in dBm.

**property fetch\_emf**

Return the current EMF measurement data. JSON format.

**property fetch\_emf\_meter**

Return the live EMF measurement data. JSON format.

**property fetch\_emf\_meter\_sample**

Return the EMF measurement data for a specified sample number. JSON format.

**property fetch\_interference\_power**

Fetch Interference Finder Integrated Power.

**property fetch\_mimo\_antenas**

Returns the sync power measurement in json format.

**property fetch\_ocupied\_bw**

Returns the different set of measurement information depending on the suffix.

**property fetch\_ota\_mapping**

Returns the most recent Coverage Mapping measurement result.

**property fetch\_pan**

Return the current Pulse Analyzer measurement data. JSON format

**property fetch\_pbch\_constellation**

Get the latest Physical Broadcast Channel constellation hitmap

**property fetch\_pci**

Returns PCI measurements

**property fetch\_pdsch**

Returns the Data Channel Measurements in JSON format.

**property fetch\_pdsch\_constellation**

Get the latest Physical Downlink Shared Channel constellation

**property fetch\_peak**

Returns a pair of peak amplitude in current sweep.

**property fetch\_power**

Returns the most recent channel power measurement.

**property fetch\_rrm**

Returns the Radio Resource Management in JSON format.

**property fetch\_scan**

Returns the cell scanner measurements in JSON format

**property fetch\_semask**

This command returns the current Spectral Emission Mask measurement result.

**property fetch\_ssb**

Returns the SSB measurement

**property fetch\_sync\_evm**

Returns the Sync EVM measurement in JSON format.

**property fetch\_sync\_power**

Returns the sync power measurements in JSON format

**property fetch\_tae**

Returns the Time Alignment Error in JSON format.

**property frequency\_center**

Sets the center frequency in Hz

**property frequency\_offset**

Sets the frequency offset in Hz

**property frequency\_span**

Sets the frequency span in Hz

**property frequency\_span\_full**

Sets the frequency span to full span

**property frequency\_span\_last**

Sets the frequency span to the previous span value.

**property frequency\_start**

Sets the start frequency in Hz

**property frequency\_step**

Set or query the step size to gradually increase or decrease frequency values in Hz

**property frequency\_stop**

Sets the start frequency in Hz

**property gps**

Returns the timestamp, latitude, and longitude of the device.

**property gps\_all**

Returns the fix timestamp, latitude, longitude, altitude and information on the sat used.

**property gps\_full**

Returns the timestamp, latitude, longitude, altitude, and satellite count of the device.

**property gps\_last**

Returns the timestamp, latitude, longitude, and altitude of the last fixed GPS result.

**init\_all\_sweep()**

Initiate all sweep/measurement.

**property init\_continuous**

Specified whether the sweep/measurement is triggered continuously

**property init\_spa\_self**

Perform a self-test and return the results.

**init\_sweep()**

Initiate a sweep/measurement.

**property meas\_acpower**

Sets the active measurement to adjacent channel power ratio, sets the default measurement parameters, triggers a new measurement and returns the main channel power, lower adjacent, upper adjacent, lower alternate and upper alternate channel power results.

**property meas\_emf\_meter\_clear\_all**

Clear the EMF measurement data of all samples. Sampling state will be turned off if it was on.

**property meas\_emf\_meter\_clear\_sample**

Clear the EMF measurement data for a specified sample number. Sampling state will be turned off if the specified sample is currently active.

**property meas\_emf\_meter\_sample**

Start or Stop applying the measurement results to the currently selected sample

**property meas\_int\_power**

Sets the active measurement to interference finder, sets the default measurement parameters, triggers a new measurement and returns integrated power as the result. It is a combination of the commands :CONFigure:INTerference; :READ:INTerference:POWer?

**property meas\_iq\_capture**

This set command is used to start the IQ capture measurement.

**property meas\_iq\_capture\_fail**

Sets or queries whether the instrument will automatically save an IQ capture when losing sync

**property meas\_ota\_mapp**

Sets the active measurement to OTA Coverage Mapping, sets the default measurement parameters, triggers a new measurement, and returns the measured values.

**property meas\_ota\_run**

Turn on/off OTA Coverage Mapping Data Collection. The instrument must be in Coverage Mapping measurement for the command to be effective

**property meas\_power**

Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns channel power as the result. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower:CHPower?

**property meas\_power\_all**

Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns the channel power and channel power density results. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower?

**property power\_density**

Sets the active measurement to channel power, sets the default measurement parameters, triggers a new measurement and returns channel power density as the result. It is a combination of the commands :CONFigure:CHPower; :READ:CHPower:DENSity?

**property preamp**

Sets the state of the preamp. Note that this may cause a change in the reference level and/or attenuation.

**property sense\_mode**

Set the operational mode of the Spa app.

**property view\_sense\_modes**

Returns a list of available modes for the Spa application. The response is a comma-separated list of mode names. See command [:SENSe]:MODE for the mode name specification.

### 7.14.5 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: [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.

## 7.15 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](#).

### 7.15.1 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** – delay between sending and reading (default 0.05 sec)
- **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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**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 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 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 continous 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 continous 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.

## 7.16 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](#).

### 7.16.1 BK Precision 9130B DC Power Supply

```
class pymeasure.instruments.bkprecision.BKPrecision9130B(adapter, name='BK Precision 9130B  
Source', **kwargs)
```

Bases: [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)

## 7.17 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](#).

### 7.17.1 Danfysik 8500 Power Supply

```
class pymeasure.instruments.danfysik.Danfysik8500(adapter, name='Danfysik 8500 Current Supply',
                                                **kwargs)
```

Bases: [Instrument](#)

Represents the Danfysik 8500 Electromagnet Current Supply and provides a high-level interface for interacting with the instrument

To allow user access to the Prolific Technology PL2303 Serial port adapter in Linux, create the file: `/etc/udev/rules.d/50-danfysik.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="067b",ATTRS{idProduct}=="2303",MODE="0666",
↳SYMLINK+="danfysik"
```

Then reload the udev rules with:

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

The device will be accessible through the port `/dev/danfysik`.

**add\_ramp\_step**(*current*)

Adds a current step to the ramp set.

**Parameters**

**current** – A current in Amps

**clear\_ramp\_set**()

Clears the ramp set.

**clear\_sequence**(*stack*)

Clears the sequence by the stack number.

**Parameters**

**stack** – A stack number between 0-15

**property current**

The actual current in Amps. This property can be set through [current\\_ppm](#).

**property current\_ppm**

The current in parts per million. This property can be set.

**property current\_setpoint**

The setpoint for the current, which can deviate from the actual current ([current](#)) while the supply is in the process of setting the value.

**disable**()

Disables the flow of current.

**enable()**

Enables the flow of current.

**property id**

Reads the identification information.

**is\_current\_stable()**

Returns True if the current is within 0.02 A of the setpoint value.

**is\_enabled()**

Returns True if the current supply is enabled.

**is\_ready()**

Returns True if the instrument is in the ready state.

**is\_sequence\_running(stack)**

Returns True if a sequence is running with a given stack number

**Parameters**

**stack** – A stack number between 0-15

**local()**

Sets the instrument in local mode, where the front panel can be used.

**property polarity**

The polarity of the current supply, being either -1 or 1. This property can be set by 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**

The slew rate of the current sweep.

**start\_ramp**()

Starts the current ramp.

**start\_sequence**(*stack*)

Starts a sequence by the stack number.

#### Parameters

**stack** – A stack number between 0-15

**property status**

A list of human-readable strings that contain the instrument status information, based on [status\\_hex](#).

**property status\_hex**

The status in hexadecimal. This value is parsed in [status](#) into a human-readable list.

**stop\_ramp**()

Stops the current ramp.

**stop\_sequence**()

Stops the currently running sequence.

**sync\_sequence**(*stack*, *delay*=0)

Arms the ramp sequence to be triggered by a hardware input to pin P33 1&2 (10 to 24 V) or a TS command. If a delay is provided, the sequence will start after the delay.

#### Parameters

- **stack** – A stack number between 0-15
- **delay** – A delay time in seconds

**wait\_for\_current**(*has\_aborted*=<function Danfysik8500.<lambda>>, *delay*=0.01)

Blocks the process until the current has stabilized. A provided function *has\_aborted* can be supplied, which is checked after each delay time (in seconds) in addition to the stability check. This allows an abort feature to be integrated.

#### Parameters

- **has\_aborted** – A function that returns True if the process should stop waiting
- **delay** – The delay time in seconds between each check for stability

**wait\_for\_ready**(*has\_aborted*=<function Danfysik8500.<lambda>>, *delay*=0.01)

Blocks the process until the instrument is ready. A provided function *has\_aborted* can be supplied, which is checked after each delay time (in seconds) in addition to the readiness check. This allows an abort feature to be integrated.

#### Parameters

- **has\_aborted** – A function that returns True if the process should stop waiting
- **delay** – The delay time in seconds between each check for readiness

## 7.18 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*.

### 7.18.1 Delta Elektronika SM7045D Power source

```
class pymeasure.instruments.deltalelektronika.SM7045D(adapter, name='Delta Elektronika SM 70-45 D', **kwargs)
```

Bases: *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.

## 7.19 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](#).

### 7.19.1 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.

## 7.20 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](#).

### 7.20.1 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)
```

(continues on next page)

(continued from previous page)

```

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=None, *, 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 determined 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

## 7.21 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](#).

### 7.21.1 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`.

## 7.22 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](#).

### 7.22.1 F.W. Bell 5080 Handheld Gaussmeter

**class** pymeasure.instruments.fwbell.FWBell5080(*adapter*, *name*='F.W. Bell 5080 Handheld Gaussmeter', *\*\*kwargs*)

Bases: [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
```

(continues on next page)

(continued from previous page)

```

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 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 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 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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write(command, \*\*kwargs)**

Write a string command to the instrument appending `write_termination`.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command*, *values*, \**args*, \*\**kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (\**args*,) – Further arguments to hand to the Adapter.

**write\_bytes**(*content*, \*\**kwargs*)

Write the bytes *content* to the instrument.

## 7.23 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](#).

### 7.23.1 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.

## 7.24 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](#).

### 7.24.1 HCP TC038 crystal oven

```
class pymeasure.instruments.hcp.TC038(adapter, name='TC038', address=1, timeout=1000,
                                       includeSCPI=False, **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.

### 7.24.2 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).

## 7.25 Hewlett Packard

This section contains specific documentation on the Hewlett Packard instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

### 7.25.1 HP 33120A Arbitrary Waveform Generator

```
class pymeasure.instruments.hp.HP33120A(adapter, name='Hewlett Packard 33120A Function Generator',
                                         **kwargs)
```

Bases: [\*Instrument\*](#)

Represents the Hewlett Packard 33120A Arbitrary Waveform Generator and provides a high-level interface for interacting with the instrument.

**property amplitude**

A floating point property that controls the voltage amplitude of the output signal. The default units are in peak-to-peak Volts, but can be controlled by [\*amplitude\\_units\*](#). The allowed range depends on the waveform shape and can be queried with [\*max\\_amplitude\*](#) and [\*min\\_amplitude\*](#).

**property amplitude\_units**

A string property that controls the units of the amplitude, which can take the values Vpp, Vrms, dBm, and default.

**beep()**

Causes a system beep.

**property frequency**

A floating point property that controls the frequency of the output in Hz. The allowed range depends on the waveform shape and can be queried with *max\_frequency* and *min\_frequency*.

**property max\_amplitude**

Reads the maximum *amplitude* in Volts for the given shape

**property max\_frequency**

Reads the maximum *frequency* in Hz for the given shape

**property max\_offset**

Reads the maximum *offset* in Volts for the given shape

**property min\_amplitude**

Reads the minimum *amplitude* in Volts for the given shape

**property min\_frequency**

Reads the minimum *frequency* in Hz for the given shape

**property min\_offset**

Reads the minimum *offset* in Volts for the given shape

**property offset**

A floating point property that controls the amplitude voltage offset in Volts. The allowed range depends on the waveform shape and can be queried with *max\_offset* and *min\_offset*.

**property shape**

A string property that controls the shape of the wave, which can take the values: sinusoid, square, triangle, ramp, noise, dc, and user.

## 7.25.2 HP 34401A Multimeter

```
class pymeasure.instruments.hp.HP34401A(adapter, name='HP 34401A', **kwargs)
```

Bases: *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 by 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 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.

### 7.25.3 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=None, *, 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 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

## 7.25.4 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=None, *, module=None, qualname=None, type=None, start=1, boundary=None)`

Bases: `IntFlag`

Enum element for error bit decoding

**property** `SRQ_mask`

Return current SRQ mask, this property can be set,  
bit assignment 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 property 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

Possible 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.

## 7.25.5 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=None*, *\**, *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=None*, *\**, *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`

A floating point value that controls the amplitude of the output in V. The allowed amplitude range generally is 10 mV to 16 V, but it is also limited by the current offset.

**ask**(*command*, *num\_bytes=None*)

Write a command to the instrument, read the response, and return the response as ASCII text.

**Parameters**

- **command** – The command to send to the instrument.
- **num\_bytes** – The number of bytes to read from the instrument. If not specified, the number of bytes is automatically determined by the command.

**property** `autovernier_enabled`

A boolean property that controls whether the autovernier is enabled.

**property** `burst_number`

An integer value that controls the number of periods generated in a burst. The allowed range is 1 to 1999. It is only valid for units with Option 001 in one of the burst modes.

**check\_errors()**

Check for errors in the 8116A.

**Returns**

list of error entries or empty list if no error occurred.

**property complement\_enabled**

A boolean property that controls whether the complement of the signal is generated.

**property complete**

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 control\_mode**

A string property that controls the control mode of the instrument. Possible values are 'off', 'FM', 'AM', 'PWM', 'VCO'.

**property duty\_cycle**

An integer value that controls the duty cycle of the output in percent. The allowed range generally is 10 % to 90 %, but it also depends on the current frequency. It is valid for all shapes except 'pulse', where *pulse\_width* is used instead.

**property frequency**

A floating point value that controls the frequency of the output in Hz. The allowed frequency range is 1 mHz to 52.5 MHz.

**property haversine\_enabled**

A boolean property that controls whether a haversine/havertriangle signal is generated when in 'triggered', 'internal\_burst' or 'external\_burst' operating mode.

**property high\_level**

A floating point value that controls the high level of the output in V. The allowed high level range generally is -7.9 V to 8 V, but it must be at least 10 mV greater than the low level.

**property limit\_enabled**

A boolean property that controls whether parameter limiting is enabled.

**property low\_level**

A floating point value that controls the low level of the output in V. The allowed low level range generally is -8 V to 7.9 V, but it must be at least 10 mV less than the high level.

**property offset**

A floating point value that controls the offset of the output in V. The allowed offset range generally is -7.95 V to 7.95 V, but it is also limited by the amplitude.

**property operating\_mode**

A string property that controls the operating mode of the instrument. Possible values (without Option 001) are: 'normal', 'triggered', 'gate', 'external\_width'. With Option 001, 'internal\_sweep', 'external\_sweep', 'external\_width', 'external\_pulse' are also available.

**property options**

Return the device options installed. The only possible option is 001.

**property output\_enabled**

A boolean property that controls whether the output is enabled.

**property pulse\_width**

A floating point value that controls the pulse width. The allowed pulse width range is 8 ns to 999 ms. The pulse width may not be larger than the period.

**property repetition\_rate**

A floating point value that controls the repetition rate (= the time between bursts) in 'internal\_burst' mode. The allowed range is 20 ns to 999 ms.

**reset()**

Initiate a reset (like a power-on reset) of the 8116A.

**property shape**

A string property that controls the shape of the output waveform. Possible values are: 'dc', 'sine', 'triangle', 'square', 'pulse'.

**shutdown()**

Gracefully close the connection to the 8116A.

**start\_autovernier(control, digit, direction, start\_value=None)**

Start the autovernier on the specified control.

**Parameters**

- **control** – The control to change, pass as `HP8116A.some_control`. Allowed controls are frequency, amplitude, offset, duty\_cycle, and pulse\_width
- **digit** – The digit to change, type: [`HP8116A.Digit`](#).
- **direction** – The direction in which to change the control, type: [`HP8116A.Direction`](#).
- **start\_value** – An optional value to start the autovernier at. If not specified, the current value of the control is used.

**property status**

Returns the status byte of the 8116A as an IntFlag-type enum.

**property sweep\_marker\_frequency**

A floating point value that controls the frequency marker in both sweep modes. At this frequency, the marker output switches from low to high. The allowed range is 1 mHz to 52.5 MHz.

**property sweep\_start**

A floating point value that controls the start frequency in both sweep modes. The allowed range is 1 mHz to 52.5 MHz.

**property sweep\_stop**

A floating point value that controls the stop frequency in both sweep modes. The allowed range is 1 mHz to 52.5 MHz.

**property sweep\_time**

A floating point value that controls the sweep time per decade in both sweep modes. The sweep time is selectable in a 1-2-5 sequence between 10 ms and 500 s.

**property trigger\_slope**

A string property that controls the slope the trigger triggers on. Possible values are: 'off', 'positive', 'negative'.

**write(command)**

Write a command to the instrument and wait until the 8116A has interpreted it.

## 7.25.6 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*
- *Demodulation*
- *Frequency*
- *Resolution Bandwidth*
- *Video*
- *FFT & Measurements*
- *Trace*
- *Marker*
- *Diagnostic Values*
- *Sweep*
- *Normalization*
- *Open/Short Calibration (Reflection)*
- *Thru Calibration*

### 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
```

(continues on next page)

(continued from previous page)

```
# 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)
```

(continues on next page)

(continued from previous page)

```

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 returns 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 or 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, str

Takes a value with the unit of [amplitude\\_unit](#) or ‘ON’ / ‘OFF’

```
instr.display_line = 'ON'
instr.display_line = -10

if instr.detector_mode == 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
```

(continues on next page)

(continued from previous page)

```
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 measurement 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)
instr.preset()
```

(continues on next page)

(continued from previous page)

```
# 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 overrange, 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_d1` overrides AMB.

Type: bool

**Warning:** The displayed amplitude of each trace element falls in one of 600 data points. There are 10 points of overrange, 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
```

(continues on next page)

(continued from previous page)

```
# 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()

# 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 typically 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 = "ON"
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 on and off and adjusts the output power.

Type: str, float

---

**Note:** Only available with an HP 8560A Option 002.

---

#### **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 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 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\_mode

Control the mixer mode. Select either the internal mixer or supply an external mixer. Takes enum 'Mixer-Mode' 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 Range (GHz)	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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *, 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=None, *,
                                                                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=None, *, 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=None, *, module=None,
                                                         qualname=None, type=None, start=1,
                                                         boundary=None)
```

Bases: StrEnum

Enumeration.

**SpectrumAnalyzer = 'SA'**

Stimulus Response

**StimulusResponse = 'SR'**

Spectrum Analyzeze

```
class pymeasure.instruments.hp.hp856Xx.SweepOut(value, names=None, *, 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=None, *, 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=None, *, 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.

## 7.25.7 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*=None, \*, *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 regarding 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

## 7.25.8 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()**

Initiate trigger via low-level GPIB-command (aka GET - group execute trigger)

**reset()**

Initiates 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.

### 7.25.9 HP System Power Supplies HP663XA

Currently supported models are:

Model	Voltage	Current	Power
6632A	0..20 V	0..5.0 A	100 W
6633A	0..50 V	0..2.5 A	100 W
6634A	0..100 V	0..1.0 A	100 W

**Note:**

- The multi-channel system power supplies HP 6621A, 6622A, 6623A, 6624A, 6625A, 6626A, 6627A & 6628A share some of the command syntax and could probably be incorporated in this implementation
- The B-version of these models (6632B, 6633B & 6634B) are SPCI-compliant and could be implemented in a similiar manner

**class** `pymeasure.instruments.hp.HP6632A`(*adapter*, *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*=None, \*, *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=None, *, 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 reprogramming 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 bool 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 float 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 property 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.

## 7.26 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](#).

### 7.26.1 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=None, \*, 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.

## 7.27 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](#).

### 7.27.1 Keithley 2000 Multimeter

```
class pymeasure.instruments.keithley.Keithley2000(adapter, name='Keithley 2000 Multimeter',  
                                                  **kwargs)
```

Bases: KeithleyBuffer, [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**

A string property that enables or disables the system status beeper, which can take the values: `enabled` and `disabled`.

**property buffer\_data**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**check\_errors()**

Read all errors from the instrument 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.

**config\_buffer**(*points=64, delay=0*)

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**property current**

Reads a DC or AC current measurement in Amps, based on the active *mode*.

**property current\_ac\_bandwidth**

A floating point property that sets the AC current detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

**property current\_ac\_digits**

An integer property that controls the number of digits in the AC current readings, which can take values from 4 to 7.

**property current\_ac\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the AC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property current\_ac\_range**

A floating point property that controls the AC current range in Amps, which can take values from 0 to 3.1 A. Auto-range is disabled when this property is set.

**property current\_ac\_reference**

A floating point property that controls the AC current reference value in Amps, which can take values from -3.1 to 3.1 A.

**property current\_digits**

An integer property that controls the number of digits in the DC current readings, which can take values from 4 to 7.

**property current\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property current\_range**

A floating point property that controls the DC current range in Amps, which can take values from 0 to 3.1 A. Auto-range is disabled when this property is set.

**property current\_reference**

A floating point property that controls the DC current reference value in Amps, which can take values from -3.1 to 3.1 A.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**disable\_filter**(*mode=None*)

Disables the averaging filter for the active mode, or can set another mode by its name.

**Parameters**

- mode** – A valid *mode* name, or None for the active mode

**disable\_reference**(*mode=None*)

Disables the reference for the active mode, or can set another mode by its name.

**Parameters**

**mode** – A valid *mode* name, or None for the active mode

**enable\_filter**(*mode=None, type='repeat', count=1*)

Enables the averaging filter for the active mode, or can set another mode by its name.

**Parameters**

- **mode** – A valid *mode* name, or None for the active mode
- **type** – The type of averaging filter, either ‘repeat’ or ‘moving’.
- **count** – A number of averages, which can take values from 1 to 100

**enable\_reference**(*mode=None*)

Enables the reference for the active mode, or can set another mode by its name.

**Parameters**

**mode** – A valid *mode* name, or None for the active mode

**property frequency**

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

**property frequency\_aperature**

A floating point property that controls the frequency aperature in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

**property frequency\_digits**

An integer property that controls the number of digits in the frequency readings, which can take values from 4 to 7.

**property frequency\_reference**

A floating point property that controls the frequency reference value in Hz, which can take values from 0 to 15 MHz.

**property frequency\_threshold**

A floating point property that controls the voltage signal threshold level in Volts for the frequency measurement, which can take values from 0 to 1010 V.

**property id**

Get the identification of the instrument.

**is\_buffer\_full**()

Returns True if the buffer is full of measurements.

**local**()

Returns control to the instrument panel, and enables the panel if disabled.

**measure\_continuity**()

Configures the instrument to perform continuity testing.

**measure\_current**(*max\_current=0.01, ac=False*)

Configures the instrument to measure current, based on a maximum current to set the range, and a boolean flag to determine if DC or AC is required.

**Parameters**

- **max\_current** – A current in Volts to set the current range

- **ac** – False for DC current, and True for AC current

**measure\_diode()**

Configures the instrument to perform diode testing.

**measure\_frequency()**

Configures the instrument to measure the frequency.

**measure\_period()**

Configures the instrument to measure the period.

**measure\_resistance(max\_resistance=10000000.0, wires=2)**

Configures the instrument to measure voltage, based on a maximum voltage to set the range, and a boolean flag to determine if DC or AC is required.

**Parameters**

- **max\_voltage** – A voltage in Volts to set the voltage range
- **ac** – False for DC voltage, and True for AC voltage

**measure\_temperature()**

Configures the instrument to measure the temperature.

**measure\_voltage(max\_voltage=1, ac=False)**

Configures the instrument to measure voltage, based on a maximum voltage to set the range, and a boolean flag to determine if DC or AC is required.

**Parameters**

- **max\_voltage** – A voltage in Volts to set the voltage range
- **ac** – False for DC voltage, and True for AC voltage

**property mode**

A string property that controls the configuration mode for measurements, which can take the values: `current (DC)`, `current ac`, `voltage (DC)`, `voltage ac`, `resistance (2-wire)`, `resistance 4W (4-wire)`, `period`, `temperature`, `diode`, and `frequency`.

**property options**

Get the device options installed.

**property period**

Reads a period measurement in seconds, based on the active *mode*.

**property period\_aperature**

A floating point property that controls the period aperature in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

**property period\_digits**

An integer property that controls the number of digits in the period readings, which can take values from 4 to 7.

**property period\_reference**

A floating point property that controls the period reference value in seconds, which can take values from 0 to 1 s.

**property period\_threshold**

A floating point property that controls the voltage signal threshold level in Volts for the period measurement, which can take values from 0 to 1010 V.

**read(\*\*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()**

Resets the buffer.

**property resistance**

Reads a resistance measurement in Ohms for both 2-wire and 4-wire configurations, based on the active *mode*.

**property resistance\_4W\_digits**

An integer property that controls the number of digits in the 4-wire resistance readings, which can take values from 4 to 7.

**property resistance\_4W\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the 4-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property resistance\_4W\_range**

A floating point property that controls the 4-wire resistance range in Ohms, which can take values from 0 to 120 MOhms. Auto-range is disabled when this property is set.

**property resistance\_4W\_reference**

A floating point property that controls the 4-wire resistance reference value in Ohms, which can take values from 0 to 120 MOhms.

**property resistance\_digits**

An integer property that controls the number of digits in the 2-wire resistance readings, which can take values from 4 to 7.

**property resistance\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property resistance\_range**

A floating point property that controls the 2-wire resistance range in Ohms, which can take values from 0 to 120 MOhms. Auto-range is disabled when this property is set.

**property resistance\_reference**

A floating point property that controls the 2-wire resistance reference value in Ohms, which can take values from 0 to 120 MOhms.

**shutdown()**

Brings the instrument to a safe and stable state

**start\_buffer()**

Starts the buffer.

**property status**

Get the status byte and Master Summary Status bit.

**stop\_buffer()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**property temperature**

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

**property temperature\_digits**

An integer property that controls the number of digits in the temperature readings, which can take values from 4 to 7.

**property temperature\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the temperature measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property temperature\_reference**

A floating point property that controls the temperature reference value in Celsius, which can take values from -200 to 1372 C.

**property trigger\_count**

An integer property that controls the trigger count, which can take values from 1 to 9,999.

**property trigger\_delay**

A floating point property that controls the trigger delay in seconds, which can take values from 1 to 9,999,999.999 s.

**property voltage**

Reads a DC or AC voltage measurement in Volts, based on the active *mode*.

**property voltage\_ac\_bandwidth**

A floating point property that sets the AC voltage detector bandwidth in Hz, which can take the values 3, 30, and 300 Hz.

**property voltage\_ac\_digits**

An integer property that controls the number of digits in the AC voltage readings, which can take values from 4 to 7.

**property voltage\_ac\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the AC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property voltage\_ac\_range**

A floating point property that controls the AC voltage range in Volts, which can take values from 0 to 757.5 V. Auto-range is disabled when this property is set.

**property voltage\_ac\_reference**

A floating point property that controls the AC voltage reference value in Volts, which can take values from -757.5 to 757.5 Volts.

**property voltage\_digits**

An integer property that controls the number of digits in the DC voltage readings, which can take values from 4 to 7.

**property voltage\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property voltage\_range**

A floating point property that controls the DC voltage range in Volts, which can take values from 0 to 1010 V. Auto-range is disabled when this property is set.

**property voltage\_reference**

A floating point property that controls the DC voltage reference value in Volts, which can take values from -1010 to 1010 V.

**wait\_for(query\_delay=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer(should\_stop=<function KeithleyBuffer.<lambda>>, timeout=60, interval=0.1)**

Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**write(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.

## 7.27.2 Keithley 2260B DC Power Supply

**class** pymeasure.instruments.keithley.**Keithley2260B**(*adapter*, *name*='Keithley 2260B DC Power Supply', *read\_termination*='\n', \*\**kwargs*)

Bases: [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

Simultaneous control of voltage (volts) and current (amps). Values need to be supplied as tuple of (voltage, current). Depending on whether the instrument is in constant current or constant voltage mode, the values achieved by the instrument will differ from the ones set.

#### check\_errors()

Logs 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()**

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 current**

Reads the current (in Ampere) the dc power supply is putting out.

**property current\_limit**

A floating point property that controls the source current in amps. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual current achieved.

**property error**

Returns a tuple of an error code and message from a single error.

**property id**

Get the identification of the instrument.

**property options**

Get the device options installed.

**property output\_enabled**

A boolean property that controls whether the source is enabled, takes values True or False.

**property power**

Reads the power (in Watt) the dc power supply is putting out.

**read(\*\*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()**

Disable output, call parent function

**property status**

Get the status byte and Master Summary Status bit.

**property voltage**

Reads the voltage (in Volt) the dc power supply is putting out.

**property voltage\_setpoint**

A floating point property that controls the source voltage in volts. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual voltage achieved.

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*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.

### 7.27.3 Keithley 2306 Dual Channel Battery/Charger Simulator

```
class pymeasure.instruments.keithley.Keithley2306(adapter, name='Keithley 2306', **kwargs)
```

Bases: *Instrument*

Represents the Keithley 2306 Dual Channel Battery/Charger Simulator.

**property both\_channels\_enabled**

A boolean setting that controls whether both channel outputs are enabled, takes values of True or False.

**ch**(*channel\_number*)

Get a channel from this instrument.

**Param**

*channel\_number*: int: the number of the channel to be selected

**Type**

Keithley2306Channel

**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 display\_brightness**

A floating point property that controls the display brightness, takes values between 0.0 and 1.0. A blank display is 0.0, 1/4 brightness is for values less or equal to 0.25, otherwise 1/2 brightness for values less than or equal to 0.5, otherwise 3/4 brightness for values less than or equal to 0.75, otherwise full brightness.

**property display\_channel**

An integer property that controls the display channel, takes values 1 or 2.

**property display\_enabled**

A boolean property that controls whether the display is enabled, takes values True or False.

**property display\_text\_data**

A string property that control text to be displayed, takes strings up to 32 characters.

**property display\_text\_enabled**

A boolean property that controls whether display text is enabled, takes values True or False.

**property id**

Get the identification of the instrument.

**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()**

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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write(command, \*\*kwargs)**

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values(command, values, \*args, \*\*kwargs)**

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

## 7.27.4 Keithley 2400 SourceMeter

**class** pymeasure.instruments.keithley.**Keithley2400**(*adapter*, *name*='Keithley 2400 SourceMeter', *\*\*kwargs*)

Bases: KeithleyBuffer, *Instrument*

Represents the Keithley 2400 SourceMeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2400("GPIB::1")

keithley.apply_current()           # Sets up to source current
keithley.source_current_range = 10e-3 # Sets the source current range to 10 mA
keithley.compliance_voltage = 10    # Sets the compliance voltage to 10 V
keithley.source_current = 0         # Sets the source current to 0 mA
keithley.enable_source()           # Enables the source output

keithley.measure_voltage()         # Sets up to measure voltage

keithley.ramp_to_current(5e-3)     # Ramps the current to 5 mA
print(keithley.voltage)             # Prints the voltage in Volts

keithley.shutdown()                # Ramps the current to 0 mA and disables_
↪ output
```

**apply\_current**(*current\_range*=None, *compliance\_voltage*=0.1)

Configures the instrument to apply a source current, and uses an auto range unless a current range is specified. The compliance voltage is also set.

### Parameters

- **compliance\_voltage** – A float in the correct range for a *compliance\_voltage*
- **current\_range** – A *current\_range* value or None

**apply\_voltage**(*voltage\_range*=None, *compliance\_current*=0.1)

Configures the instrument to apply a source voltage, and uses an auto range unless a voltage range is specified. The compliance current is also set.

### Parameters

- **compliance\_current** – A float in the correct range for a *compliance\_current*
- **voltage\_range** – A *voltage\_range* value or None

**property auto\_output\_off**

A boolean property that enables or disables the auto output-off. Valid values are True (output off after measurement) and False (output stays on after measurement).

**auto\_range\_source**()

Configures the source to use an automatic range.

**property auto\_zero**

A property that controls the auto zero option. Valid values are True (enabled) and False (disabled) and 'ONCE' (force immediate).

**beep**(*frequency, duration*)

Sounds a system beep.

**Parameters**

- **frequency** – A frequency in Hz between 65 Hz and 2 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

**property buffer\_data**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**check\_errors()**

Logs any system errors reported by the instrument.

**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 compliance\_current**

A floating point property that controls the compliance current in Amps.

**property compliance\_voltage**

A floating point property that controls the compliance voltage in Volts.

**config\_buffer**(*points=64, delay=0*)

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**property current**

Reads the current in Amps, if configured for this reading.

**property current\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property current\_range**

A floating point property that controls the measurement current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**disable\_output\_trigger()**

Disables the output trigger for the Trigger layer

**disable\_source()**

Disables the source of current or voltage depending on the configuration of the instrument.

**property display\_enabled**

A boolean property that controls whether or not the display of the sourcemeter is enabled. Valid values are True and False.

**enable\_source()**

Enables the source of current or voltage depending on the configuration of the instrument.

**property error**

Returns a tuple of an error code and message from a single error.

**property filter\_count**

A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

**property filter\_state**

A string property that controls if the filter is active.

**property filter\_type**

A String property that controls the filter's type. REP : Repeating filter MOV : Moving filter

**property id**

Get the identification of the instrument.

**is\_buffer\_full()**

Returns True if the buffer is full of measurements.

**property line\_frequency**

An integer property that controls the line frequency in Hertz. Valid values are 50 and 60.

**property line\_frequency\_auto**

A boolean property that enables or disables auto line frequency. Valid values are True and False.

**property max\_current**

Returns the maximum current from the buffer

**property max\_resistance**

Returns the maximum resistance from the buffer

**property max\_voltage**

Returns the maximum voltage from the buffer

**property maximums**

Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list.

**property mean\_current**

Returns the mean current from the buffer

**property mean\_resistance**

Returns the mean resistance from the buffer

**property mean\_voltage**

Returns the mean voltage from the buffer

**property means**

Returns the calculated means (averages) for voltage, current, and resistance from the buffer data as a list.

**property measure\_concurrent\_functions**

A boolean property that enables or disables the ability to measure more than one function simultaneously. When disabled, volts function is enabled. Valid values are True and False.

**measure\_current**(*nplc=1, current=0.000105, auto\_range=True*)

Configures the measurement of current.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -1.05 A to 1.05 A
- **auto\_range** – Enables auto\_range if True, else uses the set current

**measure\_resistance**(*nplc=1, resistance=210000.0, auto\_range=True*)

Configures the measurement of resistance.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **resistance** – Upper limit of resistance in Ohms, from -210 MOhms to 210 MOhms
- **auto\_range** – Enables auto\_range if True, else uses the set resistance

**measure\_voltage**(*nplc=1, voltage=21.0, auto\_range=True*)

Configures the measurement of voltage.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -210 V to 210 V
- **auto\_range** – Enables auto\_range if True, else uses the set voltage

**property min\_current**

Returns the minimum current from the buffer

**property min\_resistance**

Returns the minimum resistance from the buffer

**property min\_voltage**

Returns the minimum voltage from the buffer

**property minimums**

Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

**property options**

Get the device options installed.

**property output\_off\_state**

Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

**output\_trigger\_on\_external**(*line=1, after='DEL'*)

Configures the output trigger on the specified trigger link line number, with the option of supplying the part of the measurement after which the trigger should be generated (default to delay, which is right before the measurement)

**Parameters**

- **line** – A trigger line from 1 to 4
- **after** – An event string that determines when to trigger

**ramp\_to\_current**(*target\_current, steps=30, pause=0.02*)

Ramps to a target current from the set current value over a certain number of linear steps, each separated by a pause duration.

**Parameters**

- **target\_current** – A current in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

**ramp\_to\_voltage**(*target\_voltage, steps=30, pause=0.02*)

Ramps to a target voltage from the set voltage value over a certain number of linear steps, each separated by a pause duration.

**Parameters**

- **target\_voltage** – A voltage in Amps
- **steps** – An integer number of steps
- **pause** – A pause duration in seconds to wait between steps

**read**(*\*\*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()**

Resets the buffer.

**property resistance**

Reads the resistance in Ohms, if configured for this reading.

**property resistance\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property resistance\_range**

A floating point property that controls the resistance range in Ohms, which can take values from 0 to 210 MOhms. Auto-range is disabled when this property is set.

**sample\_continuously()**

Causes the instrument to continuously read samples and turns off any buffer or output triggering

**set\_timed\_arm(interval)**

Sets up the measurement to be taken with the internal trigger at a variable sampling rate defined by the interval in seconds between sampling points

**set\_trigger\_counts(arm, trigger)**

Sets the number of counts for both the sweeps (arm) and the points in those sweeps (trigger), where the total number of points can not exceed 2500

**shutdown()**

Ensures that the current or voltage is turned to zero and disables the output.

**property source\_current**

A floating point property that controls the source current in Amps.

**property source\_current\_range**

A floating point property that controls the source current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

**property source\_delay**

A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

**property source\_delay\_auto**

A boolean property that enables or disables auto delay. Valid values are True and False.

**property source\_enabled**

A boolean property that controls whether the source is enabled, takes values True or False. The convenience methods [enable\\_source\(\)](#) and [disable\\_source\(\)](#) can also be used.

**property source\_mode**

A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods [`apply\_current\(\)`](#) and [`apply\_voltage\(\)`](#) can also be used.

**property source\_voltage**

A floating point property that controls the source voltage in Volts.

**property source\_voltage\_range**

A floating point property that controls the source voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

**property standard\_devs**

Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

**start\_buffer()**

Starts the buffer.

**status()**

Get the status byte and Master Summary Status bit.

**property std\_current**

Returns the current standard deviation from the buffer

**property std\_resistance**

Returns the resistance standard deviation from the buffer

**property std\_voltage**

Returns the voltage standard deviation from the buffer

**stop\_buffer()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**triad(*base\_frequency*, *duration*)**

Sounds a musical triad using the system beep.

**Parameters**

- **base\_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

**trigger()**

Executes a bus trigger, which can be used when [`trigger\_on\_bus\(\)`](#) is configured.

**property trigger\_count**

An integer property that controls the trigger count, which can take values from 1 to 9,999.

**property trigger\_delay**

A floating point property that controls the trigger delay in seconds, which can take values from 0 to 999.9999 s.

**trigger\_immediately()**

Configures measurements to be taken with the internal trigger at the maximum sampling rate.

**trigger\_on\_bus()**

Configures the trigger to detect events based on the bus trigger, which can be activated by [`trigger\(\)`](#).

**trigger\_on\_external**(*line=1*)

Configures the measurement trigger to be taken from a specific line of an external trigger

**Parameters**

**line** – A trigger line from 1 to 4

**use\_front\_terminals**()

Enables the front terminals for measurement, and disables the rear terminals.

**use\_rear\_terminals**()

Enables the rear terminals for measurement, and disables the front terminals.

**property voltage**

Reads the voltage in Volts, if configured for this reading.

**property voltage\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property voltage\_range**

A floating point property that controls the measurement voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer**(*should\_stop=<function KeithleyBuffer.<lambda>>, timeout=60, interval=0.1*)

Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**property wires**

An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

**write**(*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.

### 7.27.5 Keithley 2450 SourceMeter

**class** pymeasure.instruments.keithley.**Keithley2450**(*adapter*, *name*='Keithley 2450 SourceMeter', **\*\*kwargs**)

Bases: KeithleyBuffer, *Instrument*

Represents the Keithley 2450 SourceMeter and provides a high-level interface for interacting with the instrument.

```
keithley = Keithley2450("GPIB::1")

keithley.apply_current()           # Sets up to source current
keithley.source_current_range = 10e-3 # Sets the source current range to 10 mA
keithley.compliance_voltage = 10    # Sets the compliance voltage to 10 V
keithley.source_current = 0         # Sets the source current to 0 mA
keithley.enable_source()           # Enables the source output

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**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**check\_errors()**

Logs any system errors reported by the instrument.

**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 compliance\_current**

A floating point property that controls the compliance current in Amps.

**property compliance\_voltage**

A floating point property that controls the compliance voltage in Volts.

**config\_buffer(*points=64, delay=0*)**

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**property current**

Reads the current in Amps, if configured for this reading.

**property current\_filter\_count**

A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

**property current\_filter\_state**

A string property that controls if the filter is active.

**property current\_filter\_type**

A String property that controls the filter's type for the current. REP : Repeating filter MOV : Moving filter

**property current\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property current\_output\_off\_state**

Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

**property current\_range**

A floating point property that controls the measurement current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**disable\_source()**

Disables the source of current or voltage depending on the configuration of the instrument.

**enable\_source()**

Enables the source of current or voltage depending on the configuration of the instrument.

**property error**

Returns a tuple of an error code and message from a single error.

**property id**

Get the identification of the instrument.

**is\_buffer\_full()**

Returns True if the buffer is full of measurements.

**property max\_current**

Returns the maximum current from the buffer

**property max\_resistance**

Returns the maximum resistance from the buffer

**property max\_voltage**

Returns the maximum voltage from the buffer

**property maximums**

Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list.

**property mean\_current**

Returns the mean current from the buffer

**property mean\_resistance**

Returns the mean resistance from the buffer

**property mean\_voltage**

Returns the mean voltage from the buffer

**property means**

Reads the calculated means (averages) for voltage, current, and resistance from the buffer data as a list.

**measure\_current**(*nplc=1, current=0.000105, auto\_range=True*)

Configures the measurement of current.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -1.05 A to 1.05 A
- **auto\_range** – Enables auto\_range if True, else uses the set current

**measure\_resistance**(*nplc=1, resistance=210000.0, auto\_range=True*)

Configures the measurement of resistance.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **resistance** – Upper limit of resistance in Ohms, from -210 MOhms to 210 MOhms
- **auto\_range** – Enables auto\_range if True, else uses the set resistance

**measure\_voltage**(*nplc=1, voltage=21.0, auto\_range=True*)

Configures the measurement of voltage.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -210 V to 210 V
- **auto\_range** – Enables auto\_range if True, else uses the set voltage

**property min\_current**

Returns the minimum current from the buffer

**property min\_resistance**

Returns the minimum resistance from the buffer

**property min\_voltage**

Returns the minimum voltage from the buffer

**property minimums**

Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

**property options**

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**()

Resets the buffer.

**property resistance**

Reads the resistance in Ohms, if configured for this reading.

**property resistance\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property resistance\_range**

A floating point property that controls the resistance range in Ohms, which can take values from 0 to 210 MOhms. Auto-range is disabled when this property is set.

**shutdown**()

Ensures that the current or voltage is turned to zero and disables the output.

**property source\_current**

A floating point property that controls the source current in Amps.

**property source\_current\_delay**

A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

**property source\_current\_delay\_auto**

A boolean property that enables or disables auto delay. Valid values are True and False.

**property source\_current\_range**

A floating point property that controls the source current range in Amps, which can take values between -1.05 and +1.05 A. Auto-range is disabled when this property is set.

**property source\_enabled**

Reads a boolean value that is True if the source is enabled.

**property source\_mode**

A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods [`apply\_current\(\)`](#) and [`apply\_voltage\(\)`](#) can also be used.

**property source\_voltage**

A floating point property that controls the source voltage in Volts.

**property source\_voltage\_delay**

A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 0 [seconds] and 999.9999 [seconds].

**property source\_voltage\_delay\_auto**

A boolean property that enables or disables auto delay. Valid values are True and False.

**property source\_voltage\_range**

A floating point property that controls the source voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

**property standard\_devs**

Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

**start\_buffer()**

Starts the buffer.

**property status**

Get the status byte and Master Summary Status bit.

**property std\_current**

Returns the current standard deviation from the buffer

**property std\_resistance**

Returns the resistance standard deviation from the buffer

**property std\_voltage**

Returns the voltage standard deviation from the buffer

**stop\_buffer()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**triad**(*base\_frequency*, *duration*)

Sounds a musical triad using the system beep.

**Parameters**

- **base\_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

**trigger**()

Executes a bus trigger.

**use\_front\_terminals**()

Enables the front terminals for measurement, and disables the rear terminals.

**use\_rear\_terminals**()

Enables the rear terminals for measurement, and disables the front terminals.

**property voltage**

Reads the voltage in Volts, if configured for this reading.

**property voltage\_filter\_count**

A integer property that controls the number of readings that are acquired and stored in the filter buffer for the averaging

**property voltage\_filter\_type**

A String property that controls the filter's type for the current. REP : Repeating filter MOV : Moving filter

**property voltage\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property voltage\_output\_off\_state**

Select the output-off state of the SourceMeter. HIMP : output relay is open, disconnects external circuitry. NORM : V-Source is selected and set to 0V, Compliance is set to 0.5% full scale of the present current range. ZERO : V-Source is selected and set to 0V, compliance is set to the programmed Source I value or to 0.5% full scale of the present current range, whichever is greater. GUAR : I-Source is selected and set to 0A

**property voltage\_range**

A floating point property that controls the measurement voltage range in Volts, which can take values from -210 to 210 V. Auto-range is disabled when this property is set.

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer**(*should\_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)

Blocks the program, waiting for a full buffer. This function returns early if the *should\_stop* function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**property wires**

An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

**write**(*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.

## 7.27.6 Keithley 2700 MultiMeter/Switch System

**class** pymeasure.instruments.keithley.**Keithley2700**(*adapter*, *name*='Keithley 2700 MultiMeter/Switch System', *\*\*kwargs*)

Bases: KeithleyBuffer, *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**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**channels\_from\_rows\_columns**(*rows*, *columns*, *slot*=None)

Determine the channel numbers between column(s) and row(s) of the 7709 connection matrix. Returns a list of channel numbers. Only one of the parameters 'rows' or 'columns' can be "all"

**Parameters**

- **rows** – row number or list of numbers; can also be “all”
- **columns** – column number or list of numbers; can also be “all”
- **slot** – slot number (1 or 2) of the 7709 card to be used

**check\_errors()**

Logs any system errors reported by the instrument.

**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

**close\_rows\_to\_columns(rows, columns, slot=None)**

Closes (connects) the channels between column(s) and row(s) of the 7709 connection matrix. Only one of the parameters ‘rows’ or ‘columns’ can be “all”

**Parameters**

- **rows** – row number or list of numbers; can also be “all”
- **columns** – column number or list of numbers; can also be “all”
- **slot** – slot number (1 or 2) of the 7709 card to be used

**property closed\_channels**

Parameter that controls the opened and closed channels. All mentioned channels are closed, other channels will be opened.

**property complete**

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)**

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**determine\_valid\_channels()**

Determine what cards are installed into the Keithley 2700 and from that determine what channels are valid.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**display\_closed\_channels()**

Show the presently closed channels on the display of the Keithley 2700.

**property display\_text**

A string property that controls the text shown on the display of the Keithley 2700. Text can be up to 12 ASCII characters and must be enabled to show.

**property error**

Returns a tuple of an error code and message from a single error.

**get\_state\_of\_channels(channels)**

Get the open or closed state of the specified channels

**Parameters**

**channels** – a list of channel numbers, or single channel number

**property id**

Get the identification of the instrument.

**is\_buffer\_full()**

Returns True if the buffer is full of measurements.

**open\_all\_channels()**

Open all channels of the Keithley 2700.

**property open\_channels**

A parameter that opens the specified list of channels. Can only be set.

**open\_rows\_to\_columns(rows, columns, slot=None)**

Opens (disconnects) the channels between column(s) and row(s) of the 7709 connection matrix. Only one of the parameters 'rows' or 'columns' can be "all"

**Parameters**

- **rows** – row number or list of numbers; can also be "all"
- **columns** – column number or list of numbers; can also be "all"
- **slot** – slot number (1 or 2) of the 7709 card to be used

**property options**

Property that lists the installed cards in the Keithley 2700. Returns a dict with the integer card numbers on the position.

**read(\*\*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()**

Resets 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()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**property text\_enabled**

A boolean property that controls whether a text message can be shown on the display of the Keithley 2700.

**triad(*base\_frequency*, *duration*)**

Sounds a musical triad using the system beep.

**Parameters**

- **base\_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

**wait\_for(*query\_delay=0*)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer(*should\_stop=<function KeithleyBuffer.<lambda>>*, *timeout=60*, *interval=0.1*)**

Blocks the program, waiting for a full buffer. This function returns early if the **should\_stop** function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**write(*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.

## 7.27.7 Keithley 6221 AC and DC Current Source

```
class pymeasure.instruments.keithley.Keithley6221(adapter, name='Keithley 6221 SourceMeter',
                                                  **kwargs)
```

Bases: KeithleyBuffer, *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**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**check\_errors()**

Logs any system errors reported by the instrument.

**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.

**config\_buffer(*points=64, delay=0*)**

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**define\_arbitrary\_waveform(*datapoints, location=1*)**

Define the data points for the arbitrary waveform and copy the defined waveform into the given storage location.

**Parameters**

- **datapoints** – a list (or numpy array) of the data points; all values have to be between -1 and 1; 100 points maximum.

- **location** – integer storage location to store the waveform in. Value must be in range 1 to 4.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**disable\_output\_trigger()**

Disables the output trigger for the Trigger layer

**disable\_source()**

Disables the source of current or voltage depending on the configuration of the instrument.

**property display\_enabled**

A boolean property that controls whether or not the display of the sourcemeter is enabled. Valid values are True and False.

**enable\_source()**

Enables the source of current or voltage depending on the configuration of the instrument.

**property error**

Returns a tuple of an error code and message from a single error.

**property id**

Get the identification of the instrument.

**is\_buffer\_full()**

Returns True if the buffer is full of measurements.

**property measurement\_event\_enabled**

An integer value that controls which measurement events are registered in the Measurement Summary Bit (MSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

**property measurement\_events**

An integer value that reads which measurement events have been registered in the Measurement event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

**property operation\_event\_enabled**

An integer value that controls which operation events are registered in the Operation Summary Bit (OSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

**property operation\_events**

An integer value that reads which operation events have been registered in the Operation event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

**property options**

Get the device options installed.

**property output\_low\_grounded**

A boolean property that controls whether the low output of the triax connection is connected to earth ground (True) or is floating (False).

**output\_trigger\_on\_external**(*line=1, after='DEL'*)

Configures the output trigger on the specified trigger link line number, with the option of supplying the part of the measurement after which the trigger should be generated (default to delay, which is right before the measurement)

**Parameters**

- **line** – A trigger line from 1 to 4
- **after** – An event string that determines when to trigger

**property questionable\_event\_enabled**

An integer value that controls which questionable events are registered in the Questionable Summary Bit (QSB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

**property questionable\_events**

An integer value that reads which questionable events have been registered in the Questionable event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

**read**(*\*\*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**()

Resets the buffer.

**set\_timed\_arm**(*interval*)

Sets up the measurement to be taken with the internal trigger at a variable sampling rate defined by the interval in seconds between sampling points

**shutdown**()

Disables the output.

**property source\_auto\_range**

A boolean property that controls the auto range of the current source. Valid values are True or False.

**property source\_compliance**

A floating point property that controls the compliance of the current source in Volts. valid values are in range 0.1 [V] to 105 [V].

**property source\_current**

A floating point property that controls the source current in Amps.

**property source\_delay**

A floating point property that sets a manual delay for the source after the output is turned on before a measurement is taken. When this property is set, the auto delay is turned off. Valid values are between 1e-3 [seconds] and 999999.999 [seconds].

**property source\_enabled**

A boolean property that controls whether the source is enabled, takes values True or False. The convenience methods `enable_source()` and `disable_source()` can also be used.

**property source\_range**

A floating point property that controls the source current range in Amps, which can take values between -0.105 A and +0.105 A. Auto-range is disabled when this property is set.

**property srq\_event\_enabled**

An integer value that controls which event registers trigger the Service Request (SRQ) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

**property standard\_event\_enabled**

An integer value that controls which standard events are registered in the Event Summary Bit (ESB) status bit. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits.

**property standard\_events**

An integer value that reads which standard events have been registered in the Standard event registers. Refer to the Model 6220/6221 Reference Manual for more information about programming the status bits. Reading this value clears the register.

**start\_buffer()**

Starts the buffer.

**property status**

Get the status byte and Master Summary Status bit.

**stop\_buffer()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**triad(*base\_frequency*, *duration*)**

Sounds a musical triad using the system beep.

**Parameters**

- **base\_frequency** – A frequency in Hz between 65 Hz and 1.3 MHz
- **duration** – A time in seconds between 0 and 7.9 seconds

**trigger()**

Executes a bus trigger, which can be used when `trigger_on_bus()` is configured.

**trigger\_immediately()**

Configures measurements to be taken with the internal trigger at the maximum sampling rate.

**trigger\_on\_bus()**

Configures the trigger to detect events based on the bus trigger, which can be activated by `trigger()`.

**trigger\_on\_external**(*line=1*)

Configures the measurement trigger to be taken from a specific line of an external trigger

**Parameters**

**line** – A trigger line from 1 to 4

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer**(*should\_stop=<function KeithleyBuffer.<lambda>>, timeout=60, interval=0.1*)

Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**waveform\_abort**()

Abort the waveform output and disarm the waveform function.

**property waveform\_amplitude**

A floating point property that controls the (peak) amplitude of the waveform in Amps. Valid values are in range 2e-12 to 0.105.

**waveform\_arm**()

Arm the current waveform function.

**property waveform\_duration\_cycles**

A floating point property that controls the duration of the waveform in cycles. Valid values are in range 1e-3 to 9999999900.

**waveform\_duration\_set\_infinity**()

Set the waveform duration to infinity.

**property waveform\_duration\_time**

A floating point property that controls the duration of the waveform in seconds. Valid values are in range 100e-9 to 999999.999.

**property waveform\_dutycycle**

A floating point property that controls the duty-cycle of the waveform in percent for the square and ramp waves. Valid values are in range 0 to 100.

**property waveform\_frequency**

A floating point property that controls the frequency of the waveform in Hertz. Valid values are in range 1e-3 to 1e5.

**property waveform\_function**

A string property that controls the selected wave function. Valid values are “sine”, “ramp”, “square”, “arbitrary1”, “arbitrary2”, “arbitrary3” and “arbitrary4”.

**property waveform\_offset**

A floating point property that controls the offset of the waveform in Amps. Valid values are in range -0.105 to 0.105.

**property waveform\_phasemarker\_line**

A numerical property that controls the line of the phase marker.

**property waveform\_phasemarker\_phase**

A numerical property that controls the phase of the phase marker.

**property waveform\_ranging**

A string property that controls the source ranging of the waveform. Valid values are “best” and “fixed”.

**waveform\_start()**

Start the waveform output. Must already be armed

**property waveform\_use\_phasemarker**

A boolean property that controls whether the phase marker option is turned on or of. Valid values True (on) or False (off). Other settings for the phase marker have not yet been implemented.

**write(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.

## 7.27.8 Keithley 6517B Electrometer

```
class pymeasure.instruments.keithley.Keithley6517B(adapter, name='Keithley 6517B
                                                    Electrometer/High Resistance Meter', **kwargs)
```

Bases: KeithleyBuffer, [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
```

(continues on next page)

(continued from previous page)

```
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**

Returns a numpy array of values from the buffer.

**property buffer\_points**

An integer property that controls the number of buffer points. This does not represent actual points in the buffer, but the configuration value instead.

**check\_errors**()

Logs any system errors reported by the instrument.

**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.

**config\_buffer**(*points=64, delay=0*)

Configures the measurement buffer for a number of points, to be taken with a specified delay.

**Parameters**

- **points** – The number of points in the buffer.
- **delay** – The delay time in seconds.

**property current**

Reads the current in Amps, if configured for this reading.

**property current\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC current measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property current\_range**

A floating point property that controls the measurement current range in Amps, which can take values between -20 and +20 mA. Auto-range is disabled when this property is set.

**disable\_buffer()**

Disables the connection between measurements and the buffer, but does not abort the measurement process.

**disable\_source()**

Disables the source of current or voltage depending on the configuration of the instrument.

**enable\_source()**

Enables the source of current or voltage depending on the configuration of the instrument.

**property error**

Returns a tuple of an error code and message from a single error.

**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()**

Returns True if the buffer is full of measurements.

**measure\_current(*nplc=1, current=0.000105, auto\_range=True*)**

Configures the measurement of current.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **current** – Upper limit of current in Amps, from -21 mA to 21 mA
- **auto\_range** – Enables auto\_range if True, else uses the current\_range attribut

**measure\_resistance(*nplc=1, resistance=210000.0, auto\_range=True*)**

Configures the measurement of resistance.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **resistance** – Upper limit of resistance in Ohms, from -210 POhms to 210 POhms
- **auto\_range** – Enables auto\_range if True, else uses the resistance\_range attribut

**measure\_voltage**(*nplc=1, voltage=21.0, auto\_range=True*)

Configures the measurement of voltage.

**Parameters**

- **nplc** – Number of power line cycles (NPLC) from 0.01 to 10
- **voltage** – Upper limit of voltage in Volts, from -1000 V to 1000 V
- **auto\_range** – Enables `auto_range` if True, else uses the `voltage_range` attribut

**property options**

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**()

Resets the buffer.

**property resistance**

Reads the resistance in Ohms, if configured for this reading.

**property resistance\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the 2-wire resistance measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property resistance\_range**

A floating point property that controls the resistance range in Ohms, which can take values from 0 to 100e18 Ohms. Auto-range is disabled when this property is set.

**shutdown()**

Ensures that the current or voltage is turned to zero and disables the output.

**property source\_current\_resistance\_limit**

Boolean property which enables or disables resistance current limit

**property source\_enabled**

Reads a boolean value that is True if the source is enabled.

**property source\_voltage**

A floating point property that controls the source voltage in Volts.

**property source\_voltage\_range**

A floating point property that controls the source voltage range in Volts, which can take values from -1000 to 1000 V. Auto-range is disabled when this property is set.

**start\_buffer()**

Starts the buffer.

**property status**

Get the status byte and Master Summary Status bit.

**stop\_buffer()**

Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**trigger()**

Executes a bus trigger, which can be used when `trigger_on_bus()` is configured.

**trigger\_immediately()**

Configures measurements to be taken with the internal trigger at the maximum sampling rate.

**trigger\_on\_bus()**

Configures the trigger to detect events based on the bus trigger, which can be activated by `trigger()`.

**property voltage**

Reads the voltage in Volts, if configured for this reading.

**property voltage\_nplc**

A floating point property that controls the number of power line cycles (NPLC) for the DC voltage measurements, which sets the integration period and measurement speed. Takes values from 0.01 to 10, where 0.1, 1, and 10 are Fast, Medium, and Slow respectively.

**property voltage\_range**

A floating point property that controls the measurement voltage range in Volts, which can take values from -1000 to 1000 V. Auto-range is disabled when this property is set.

**wait\_for(query\_delay=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer(should\_stop=<function KeithleyBuffer.<lambda>>, timeout=60, interval=0.1)**

Blocks the program, waiting for a full buffer. This function returns early if the `should_stop` function returns True or the timeout is reached before the buffer is full.

**Parameters**

- **should\_stop** – A function that returns True when this function should return early
- **timeout** – A time in seconds after which this function should return early
- **interval** – A time in seconds for how often to check if the buffer is full

**write**(*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.

## 7.27.9 Keithley 2750 Multimeter/Switch System

**class** pymeasure.instruments.keithley.**Keithley2750**(*adapter*, *name*='Keithley 2750 Multimeter/Switch System', *\*\*kwargs*)

Bases: *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 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

**close(channel)**

Closes (connects) the specified channel.

**Parameters**

**channel** (*int*) – 3-digit number for the channel

**Returns**

None

**property closed\_channels**

Reads the list of closed channels

**property complete**

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.

**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()**

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=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command, \*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command, values, \*args, \*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content, \*\*kwargs*)

Write the bytes *content* to the instrument.

### 7.27.10 Keithley 2600 SourceMeter

```
class pymeasure.instruments.keithley.Keithley2600(adapter, name='Keithley 2600 SourceMeter',  
                                                  **kwargs)
```

Bases: [Instrument](#)

Represents the Keithley 2600 series (channel A and B) SourceMeter

**check\_errors()**

Logs 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()**

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 error**

Returns a tuple of an error code and message from a single error.

**property id**

Get the identification of the instrument.

**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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command*, **\*\*kwargs**)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command*, *values*, *\*args*, **\*\*kwargs**)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args*,) – Further arguments to hand to the Adapter.

**write\_bytes**(*content*, **\*\*kwargs**)

Write the bytes *content* to the instrument.

### 7.27.11 Keithley 2200 Series Power Supplies

**class** `pymeasure.instruments.keithley.Keithley2200`(*adapter*, *name='Keithley2200'*, **\*\*kwargs**)

Bases: `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=0, \*\*kwargs)

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

#### Parameters

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

#### Returns

NumPy array of values.

**check\_errors**()

Read all errors from the instrument 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.

**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 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()**

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=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write\_binary\_values**(*command, values, \*args, \*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content, \*\*kwargs*)

Write the bytes *content* to the instrument.

**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.

## 7.28 Keysight

This section contains specific documentation on the keysight instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

### 7.28.1 Keysight DSOX1102G Oscilloscope

**class** pymeasure.instruments.keysight.**KeysightDSOX1102G**(*adapter*, *name*='Keysight DSOX1102G Oscilloscope', *\*\*kwargs*)

Bases: [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”.

## 7.28.2 Keysight N5767A Power Supply

```
class pymeasure.instruments.keysight.KeysightN5767A(adapter, name='Keysight N5767A power supply',
                                                    **kwargs)
```

Bases: [Instrument](#)

Represents the Keysight N5767A Power supply interface for interacting with the instrument.

**property current**

Reads a setting current in Amps.

**property current\_range**

A floating point property that controls the DC current range in Amps, which can take values from 0 to 25 A. Auto-range is disabled when this property is set.

**disable()**

Disables the flow of current.

**enable()**

Enables the flow of current.

**is\_enabled()**

Returns True if the current supply is enabled.

**property voltage**

Reads a DC voltage measurement in Volts.

**property voltage\_range**

A floating point property that controls the DC voltage range in Volts, which can take values from 0 to 60 V. Auto-range is disabled when this property is set.

### 7.28.3 Keysight N7776C Power Supply

```
class pymeasure.instruments.keysight.KeysightN7776C(adapter, name='N7776C Tunable Laser Source',  
                                                    **kwargs)
```

Bases: *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**

Boolean property controlling the lock state (True/False) of the laser source

**next\_step()**

Performs the next sweep step in stepped sweep if it is paused or in manual mode.

**property output\_enabled**

Boolean Property that controls the state (on/off) of the laser source

**previous\_step()**

Performs one sweep step backwards in stepped sweep if its paused or in manual mode.

**property sweep\_mode**

Sweep mode of the swept laser source

**property sweep\_points**

Returns the number of datapoints that the :READout:DATA? command will return.

**property sweep\_speed**

Speed of the sweep (in nanometers per second).

**property sweep\_state**

State of the wavelength sweep. Stops, starts, pauses or continues a wavelength sweep. Possible state values are 0 (not running), 1 (running) and 2 (paused). Refer to the N7776C user manual for exact usage of the paused option.

**property sweep\_step**

Step width of the sweep (in nanometers).

**property sweep\_twoway**

Sets the repeat mode. Applies in stepped, continuous and manual sweep mode.

**property sweep\_wl\_start**

Start Wavelength (in nanometers) for a sweep.

**property sweep\_wl\_stop**

End Wavelength (in nanometers) for a sweep.

**property trigger\_in**

Sets the incoming trigger response and arms the module.

**property trigger\_out**

Specifies if and at which point in a sweep cycle an output trigger is generated and arms the module.

**property wavelength**

Absolute wavelength of the output light (in nanometers)

**property wl\_logging**

State (on/off) of the lambda logging feature of the laser source.

## 7.28.4 Keysight E36312A Triple Output Power Supply

**class** `pymeasure.instruments.keysight.KeysightE36312A`(*adapter*, *name*='Keysight E36312A', *\*\*kwargs*)

Bases: `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
    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"]
```

(continues on next page)

(continued from previous page)

```

→"]})
# 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=0)

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

**Parameters**

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

**Returns**

String returned by the device without read\_termination.

**binary\_values**(*command*, *query\_delay*=0, *\*\*kwargs*)

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

**Parameters**

- **command** – Command to be sent to the instrument.
- **query\_delay** – Delay between writing and reading in seconds.
- **kwargs** – Arguments for read\_binary\_values().

**Returns**

NumPy array of values.

**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.

**static control**(*get\_command*, *set\_command*, *docs*, *validator*=<function CommonBase.<lambda>>, *values*=(), *map\_values*=False, *get\_process*=<function CommonBase.<lambda>>, *set\_process*=<function CommonBase.<lambda>>, *command\_process*=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 take a value and allows processing before value mapping, returning the processed value
- **set\_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command\_process** – A function that takes a command and allows processing before executing the command

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):
```

(continues on next page)

(continued from previous page)

```
# 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=None, get_process=<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 take a value and allows processing before value mapping, returning the processed value
- **command\_process** – A function that take a command and allows processing before executing the command, for getting

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 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()**

Resets 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*=0)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command*, *\*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command*, *values*, *\*args*, *\*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args*,) – Further arguments to hand to the Adapter.

**write\_bytes**(*content*, *\*\*kwargs*)

Write the bytes *content* to the instrument.

```
class pymeasure.instruments.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)

## 7.29 Lake Shore Cryogenics

This section contains specific documentation on the Lake Shore Cryogenics instruments that are implemented. If you are interested in an instrument not included, please consider [adding the instrument](#).

### 7.29.1 Lake Shore 211 Temperature Monitor

```
class pymeasure.instruments.lakeshore.LakeShore211(adapter, name='Lake Shore 211 Temperature
                                                    Monitor', **kwargs)
```

Bases: [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=None, *, module=None, qualname=None, type=None, start=1,
                 boundary=None)
```

Bases: [IntEnum](#)

```
class AnalogRange(value, names=None, *, module=None, qualname=None, type=None, start=1,
                  boundary=None)
```

Bases: [IntEnum](#)

```
class RelayMode(value, names=None, *, module=None, qualname=None, type=None, start=1,
                boundary=None)
```

Bases: [IntEnum](#)

```
class RelayNumber(value, names=None, *, 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

## 7.29.2 Lake Shore 224 Temperature Monitor

**class** pymeasure.instruments.lakeshore.LakeShore224(*adapter*, *name*='Lakeshore Model 224 Temperature Controller', \*\*kwargs)

Bases: *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*

### 7.29.3 Lake Shore 331 Temperature Controller

**class** pymeasure.instruments.lakeshore.LakeShore331(*adapter*, *name*='Lakeshore Model 336 Temperature Controller', **\*\*kwargs**)

Bases: [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.temperature)     # Print the temperature at sensor A.
```

**input\_A**

**Channel**

[LakeShoreTemperatureChannel](#)

**input\_B**

**Channel**

[LakeShoreTemperatureChannel](#)

**output\_1**

**Channel**

[LakeShoreHeaterChannel](#)

**output\_2**

**Channel**

[LakeShoreHeaterChannel](#)

### 7.29.4 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**

A boolean property that returns whether the alarm is triggered.

**property alarm\_audible**

A boolean property that enables or disables the audible alarm beeper.

**property alarm\_high**

Property that controls the upper setpoint for the alarm mode in the current units. This takes into account the field multiplier.

**property alarm\_high\_multiplier**

Returns the multiplier for the upper alarm setpoint field.

**property alarm\_high\_raw**

ALMH %g

**property alarm\_in\_out**

A string property that controls whether an active alarm is caused when the field reading is inside (“Inside”) or outside (“Outside”) of the high and low setpoint values.

**property alarm\_low**

Property that controls the lower setpoint for the alarm mode in the current units. This takes into account the field multiplier.

**property alarm\_low\_multiplier**

Returns the multiplier for the lower alarm setpoint field.

**property alarm\_low\_raw**

ALML %g

**property alarm\_mode\_enabled**

A boolean property that enables or disables the alarm mode.

**property alarm\_sort\_enabled**

A boolean property that enables or disables the alarm Sort Pass/Fail function.

**property auto\_range**

A boolean property that controls the auto-range option of the meter. Valid values are True and False. Note that the auto-range is relatively slow and might not suffice for rapid measurements.

**property display\_filter\_enabled**

A boolean property that controls the display filter to make it more readable when the probe is exposed to a noisy field. The filter function makes a linear average of 8 readings and settles in approximately 2 seconds.

**property fast\_mode**

A boolean property that controls the fast-mode option of the meter. Valid values are True and False. When enabled, the relative mode, Max Hold mode, alarms, and autorange are disabled.

**property field**

Returns the field in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

**property field\_mode**

A string property that controls whether the gaussmeter measures AC or DC magnetic fields. Valid values are “AC” and “DC”.

**property field\_multiplier**

Returns the field multiplier for the returned magnetic field.

**property field\_range**

A floating point property that controls the field range of the meter in the current unit (G or T). Valid values are 30e3, 3e3, 300, 30 (when in Gauss), or 0.003, 0.03, 0.3, and 3 (when in Tesla).

**property field\_range\_raw**

A integer property that controls the field range of the meter. Valid values are 0 (highest) to 3 (lowest).

**property field\_raw**

Returns the field in the current units and multiplier

**property front\_panel\_brightness**

An integer property that controls the brightness of the from panel display. Valid values are 0 (dimkest) to 7 (brightest).

**property front\_panel\_locked**

A boolean property that locks or unlocks all front panel entries except pressing the Alarm key to silence alarms.

**property max\_hold\_enabled**

A boolean property that enables or disables the Max Hold function to store the largest field since the last reset (with max\_hold\_reset).

**property max\_hold\_field**

Returns the largest field since the last reset in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

**property max\_hold\_field\_raw**

Returns the largest field since the last reset in the current units and multiplier.

**property max\_hold\_multiplier**

Returns the multiplier for the returned max hold field.

**max\_hold\_reset()**

Clears the stored Max Hold value.

**property probe\_type**

Returns type of field-probe used with the gaussmeter. Possible values are High Sensitivity, High Stability, or Ultra-High Sensitivity.

**property relative\_field**

Returns the relative field in the current units. This property takes into account the field multiplier. Returns np.nan if field is out of range.

**property relative\_field\_raw**

Returns the relative field in the current units and the current multiplier.

**property relative\_mode\_enabled**

A boolean property that enables or disables the relative mode to see small variations with respect to a given setpoint.

**property relative\_multiplier**

Returns the relative field multiplier for the returned magnetic field.

**property relative\_setpoint**

Property that controls the setpoint for the relative field mode in the current units. This takes into account the field multiplier.

**property relative\_setpoint\_multiplier**

Returns the multiplier for the setpoint field.

**property relative\_setpoint\_raw**

Property that controls the setpoint for the relative field mode in the current units and multiplier.

**property serial\_number**

Returns the serial number of the probe.

**shutdown()**

Closes the serial connection to the system.

**property unit**

A string property that controls the units used by the gaussmeter. Valid values are G (Gauss), T (Tesla).

**write(command)**

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**zero\_probe(wait=True)**

Reset the probe value to 0. It is normally used with a zero gauss chamber, but may also be used with an open probe to cancel the Earth magnetic field. To cancel larger magnetic fields, the relative mode should be used.

**Parameters**

**wait** (*bool*) – Wait for 20 seconds after issuing the command to allow the resetting to finish.

## 7.29.5 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**

Returns the field in the current units

**measure**(*points*, *has\_aborted=<function LakeShore425.<lambda>>*, *delay=0.001*)

Returns the mean and standard deviation of a given number of points while blocking

**property range**

A floating point property that controls the field range in units of Gauss, which can take the values 35, 350, 3500, and 35,000 G.

**property unit**

A string property that controls the units of the instrument, which can take the values of G, T, Oe, or A/m.

**zero\_probe**()

Initiates the zero field sequence to calibrate the probe

## 7.29.6 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, celcius, or sensor units. Also provides a method to block the program until a given stable temperature is reached.

**property celcius**

Read the temperature in celcius from a channel.

**property kelvin**

Read the temperature in kelvin from a channel.

**property sensor**

Read 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, celcius, or sensor units.
- **unit** – ‘kelvin’, ‘celcius’, 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

Manual heater output in percent.

**property** output

Query the heater output in percent of the max.

**property** range

String property controlling heater range, which can take the values: off, low, medium, and high.

**property** setpoint

A floating point property that control the setpoint temperature in the preferred units of the control loop sensor.

## 7.30 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.

### 7.30.1 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

*LeCroyT3DS01204Channel*

ch\_2

Channel

*LeCroyT3DS01204Channel*

ch\_3

Channel

*LeCroyT3DS01204Channel*

ch\_4

Channel

*LeCroyT3DS01204Channel*

**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=0)**

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

**Parameters**

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

**Returns**

String returned by the device without `read_termination`.

**autoscale()**

Autoscale displayed channels.

**binary\_values**(*command*, *query\_delay*=0, *\*\*kwargs*)

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

**Parameters**

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

**Returns**

NumPy array of values.

**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 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.

```
static control(get_command, set_command, docs, validator=<function CommonBase.<lambda>>,
               values=(), map_values=False, get_process=<function CommonBase.<lambda>>,
               set_process=<function CommonBase.<lambda>>, command_process=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 take a value and allows processing before value mapping, returning the processed value
- **set\_process** – A function that takes a value and allows processing before value mapping, returning the processed value
- **command\_process** – A function that takes a command and allows processing before executing the command

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=None*, *get\_process=<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 take a value and allows processing before value mapping, returning the processed value
- **`command_process`** – A function that take a command and allows processing before executing the command, for getting

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 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()**

Resets the instrument.

#### **run()**

Starts repetitive acquisitions.

This is the same as pressing the Run key on the front panel.

**static setting**(*set\_command*, *docs*, *validator=<function CommonBase.<lambda>>*, *values=()*, *map\_values=False*, *set\_process=<function CommonBase.<lambda>>*, *check\_set\_errors=False*, *dynamic=False*)

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/run 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*=0)

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

#### Parameters

**query\_delay** – Delay between writing and reading in seconds.

#### property waveform\_first\_point

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.lecroyT3DS01204.**LeCroyT3DS01204Channel**(*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).

## 7.31 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*.

### 7.31.1 MKS Instruments 937B Vacuum Gauge Controller

```
class pymeasure.instruments.mksinst.mks937b.MKS937B(adapter, name='MKS 937B vacuum gauge
                                                    controller', address=253, **kwargs)
```

Bases: *Instrument*

MKS 937B vacuum gauge controller

Connection to the device is made through an RS232/RS485 serial connection. The communication protocol of this device is as follows:

Query: ‘@<aaa><Command>;FF’ with the response ‘@<aaa>ACK<Response>;FF’ Set command: ‘@<aaa><Command>!<parameter>;FF’ with the response ‘@<aaa>ACK<Response>;FF’ Above <aaa> is an address from 001 to 254 which can be specified upon initialization. Since ‘;FF’ is not supported by pyvisa as terminator this class overloads the device communication methods.

**Parameters**

- **adapter** – pyvisa resource name of the instrument or adapter instance
- **name** (*string*) – The name of the instrument.
- **address** – device address included in every message to the instrument (default=253)
- **kwargs** – Any valid key-word argument for Instrument

**ch\_1**

**Channel**

*IonGaugeAndPressureChannel*

**ch\_2**

**Channel**

*PressureChannel*

**ch\_3**

**Channel**

*IonGaugeAndPressureChannel*

**ch\_4**

**Channel**

*PressureChannel*

**ch\_5**

**Channel**

*IonGaugeAndPressureChannel*

**ch\_6**

**Channel**

*PressureChannel*

**property all\_pressures**

Read pressures on all channels in selected units

**check\_set\_errors()**

Check reply string for acknowledgement string.

**property combined\_pressure1**

Read pressure on channel 1 and its combination sensor

**property combined\_pressure2**

Read pressure on channel 2 and its combination sensor

**read()**

Reads from the instrument including the correct termination characters

**property serial**

Serial number of the instrument

**property unit**

Pressure unit used for all pressure readings from the instrument

**write(command)**

Write to the instrument including the device address.

**Parameters**

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

**class** pymeasure.instruments.mksinst.mks937b.**IonGaugeAndPressureChannel**(parent, id)

Bases: *PressureChannel*

Channel having both a pressure and an ion gauge sensor

**property ion\_gauge\_status**

Ion gauge status of the channel

**class** pymeasure.instruments.mksinst.mks937b.**PressureChannel**(parent, id)

Bases: *Channel*

**property power\_enabled**

Power status of the channel

**property pressure**

Pressure on the channel in units selected on the device

## 7.32 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](#).

### 7.32.1 ESP 300 Motion Controller

```
class pymeasure.instruments.newport.ESP300(adapter, name='Newport ESP 300 Motion Controller',
                                           **kwargs)
```

Bases: [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 position.

**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.

## 7.33 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](#).

### 7.33.1 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 FGGEN 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 (FGGEN 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 range 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 range 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 comparison ('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 arm-strap/pyvirtualbench Python wrapper.

## 7.34 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*.

### 7.34.1 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).

## 7.35 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](#).

### 7.35.1 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

## 7.35.2 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=None, *, 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.

### 7.35.3 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
                             # 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 be 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.

### 7.35.4 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.

## 7.36 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](#).

### 7.36.1 Parker GV6 Servo Motor Controller

**class** pymeasure.instruments.parker.**ParkerGV6**(*adapter*, *name*='Parker GV6 Motor Controller',  
\*\**kwargs*)

Bases: `Instrument`

Represents the Parker Gemini GV6 Servo Motor Controller and provides a high-level interface for interacting with the instrument

**property** `angle`

Returns the angle in degrees based on the position and whether relative or absolute positioning is enabled, returning None on error

**property** `angle_error`

Returns the angle error in degrees based on the position error, or returns None on error

**disable**()

Disables the motor from moving

**enable**()

Enables the motor to move

**is\_moving**()

Returns True if the motor is currently moving

**kill**()

Stops the motor

**move**()

Initiates the motor to move to the setpoint

**property** `position`

Returns an integer number of counts that correspond to the angular position where 1 revolution equals 4000 counts

**property position\_error**

Returns the error in the number of counts that corresponds to the error in the angular position where 1 revolution equals 4000 counts

**read()**

Overwrites the Instrument.read command to provide the correct functionality

**reset()**

Resets the motor controller while blocking and (CAUTION) resets the absolute position value of the motor

**set\_defaults()**

Sets up the default values for the motor, which is run upon construction

**set\_hardware\_limits(positive=True, negative=True)**

Enables (True) or disables (False) the hardware limits for the motor

**set\_software\_limits(positive, negative)**

Sets the software limits for motion based on the count unit where 4000 counts is 1 revolution

**property status**

Returns a list of the motor status in readable format

**stop()**

Stops the motor during movement

**use\_absolute\_position()**

Sets the motor to accept setpoints from an absolute zero position

**use\_relative\_position()**

Sets the motor to accept setpoints that are relative to the last position

## 7.37 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](#).

### 7.37.1 Pendulum CNT91 frequency counter

```
class pymeasure.instruments.pendulum.cnt91.CNT91(adapter, name='Pendulum CNT-91', **kwargs)
```

Bases: [Instrument](#)

Represents a Pendulum CNT-91 frequency counter.

**property batch\_size**

Maximum number of buffer entries that can be transmitted at once.

**buffer\_frequency\_time\_series(channel, n\_samples, sample\_rate, trigger\_source=None)**

Record a time series to the buffer and read it out after completion.

**Parameters**

- **channel** – Channel that should be used
- **n\_samples** – The number of samples
- **sample\_rate** – Sample rate in Hz

- **trigger\_source** – Optionally specify a trigger source to start the measurement

**configure\_frequency\_array\_measurement**(*n\_samples*, *channel*)

Configure the counter for an array of measurements.

**Parameters**

- **n\_samples** – The number of samples
- **channel** – Measurement channel (A, B, C, E, INTREF)

**property continuous**

Controls whether to perform continuous measurements.

**property external\_arming\_start\_slope**

Set slope for the start arming condition.

**property external\_start\_arming\_source**

Select arming input or switch off the start arming function. Options are 'A', 'B' and 'E' (rear). 'IMM' turns trigger off.

**property format**

Response format (ASCII or REAL).

**property interpolator\_autocalibrated**

Controls if interpolators should be calibrated automatically.

**property measurement\_time**

Gate time for one measurement in s.

**read\_buffer**(*expected\_length=0*)

Read out the entire buffer.

**Parameters**

**expected\_length** – The expected length of the buffer. If more data is read, values at the end are removed. Defaults to 0, which means that the entire buffer is returned independent of its length.

**Returns**

Frequency values from the buffer.

## 7.38 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](#).

### 7.38.1 Razorbill RP100 custom power supply for Razorbill Instrums stress & strain cells

**class** pymeasure.instruments.razorbill.**razorbillRP100**(*adapter*, *name*='Razorbill RP100 Piezo Stack Powersupply', *\*\*kwargs*)

Bases: [Instrument](#)

Represents Razorbill RP100 strain cell controller

```
scontrol = razorbillRP100("ASRL/dev/ttyACM0::INSTR")

scontrol.output_1 = True      # turns output on
scontrol.slew_rate_1 = 1     # sets slew rate to 1V/s
scontrol.voltage_1 = 10      # sets voltage on output 1 to 10V
```

**property contact\_current\_1**

Returns the current in amps present at the front panel output of channel 1

**property contact\_current\_2**

Returns the current in amps present at the front panel output of channel 2

**property contact\_voltage\_1**

Returns the Voltage in volts present at the front panel output of channel 1

**property contact\_voltage\_2**

Returns the Voltage in volts present at the front panel output of channel 2

**property instant\_voltage\_1**

Returns the instantaneous output of source one in volts

**property instant\_voltage\_2**

Returns the instanteneous output of source two in volts

**property output\_1**

Turns output of channel 1 on or off

**property output\_2**

Turns output of channel 2 on or off

**property slew\_rate\_1**

Sets or queries the source slew rate in volts/sec of channel 1

**property slew\_rate\_2**

Sets or queries the source slew rate in volts/sec of channel 2

**property voltage\_1**

Sets or queries the output voltage of channel 1

**property voltage\_2**

Sets or queries the output voltage of channel 2

## 7.39 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*.

### 7.39.1 R&S SFM TV test transmitter

**class** `pymeasure.instruments.rohdeschwarz.sfm.SFM`(*adapter*, *name*='Rohde&Schwarz SFM', *\*\*kwargs*)

Bases: [`Instrument`](#)

Represents the Rohde&Schwarz SFM TV test transmitter interface for interacting with the instrument.

---

**Note:** The current implementation only works with the first system in this unit.

Further source extension for system 2-6 would be required.

The intermodulation subsystem is also not yet implmented.

---

**property** `R75_out`

A bool property that controls the use of the 75R output (if installed)

Value	Meaning
False	50R output active (N)
True	75R output active (BNC)

refer also to chapter 3.6.5 of the manual

**property** `TV_country`

A string property that controls the country specifics of the video/sound system to be used

Possible values are:

Value	Meaning
BG_G	BG General
DK_G	DK General
I_G	I General
L_G	L General
GERM	Germany
BELG	Belgium
NETH	Netherlands
FIN	Finland
AUST	Australia
BG_T	BG Th
DENM	Denmark
NORW	Norway
SWED	Sweden
GUS	Russia
POL1	Poland
POL2	Poland
HUNG	Hungary
CHEC	Czech Republic
CHINA1	China
CHINA2	China
GRE	Great Britain
SAFR	South Africa
FRAN	France
USA	United States
KOR	Korea
JAP	Japan
CAN	Canada
SAM	South America

Please confirm with the manual about the details for these settings.

**property TV\_standard**

A string property that controls the type of video standard

Possible values are:

Value	Lines	System
BG	625	PAL
DK	625	SECAM
I	625	PAL
K1	625	SECAM
L	625	SECAM
M	525	NTSC
N	625	NTSC

Please confirm with the manual about the details for these settings.

**property basic\_info**

A String property containing 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 immediately!

---

**property gpib\_address**

A int property that controls the GPIB address of the unit

valid range: 0..30

**property high\_frequency\_resolution**

A property that controls the frequency resolution,

Possible selections are:

Value	Meaning
False	Low resolution (1000Hz)
True	High resolution (1Hz)

**property level**

A float property controlling the output level in dBm,

- Minimum -99dBm
- Maximum 10dBm (depending on output mode)

refer also to chapter 3.6.6.2 of the manual

**property level\_mode**

A string property controlling the output attenuator and linearity mode

Possible selections are:

Value	Meaning	max. output level
NORM	Normal mode	+6 dBm
LOWN	low noise mode	+10 dBm
CONT	continous mode	+10 dBm
LOWD	low distortion mode	+0 dBm

Continous mode allows up to 14 dB of level setting without use of the mechanical attenuator.

**property lower\_sideband\_enabled**

A bool property that controls the use of the lower sideband

refer also to chapter 3.6.6.10 of the manual

**property modulation\_enabled**

A bool property that controls the modulation status

**property nicam\_IQ\_inverted**

A bool property that controls if the NICAM IQ signals are inverted or not

Value	Meaning
False	normal (IQ)
True	inverted (QI)

**property nicam\_additional\_bits**

A int property that controls the additional data in the NICAM modulator

valid range: 0 .. 2047

**property nicam\_audio\_frequency**

A int property that controls the frequency of the internal sound generator

valid range: 0 Hz .. 15 kHz

**property nicam\_audio\_volume**

A float property that controls the audio volume in the NICAM modulator in dB

valid range: 0..60 dB

**property nicam\_bit\_error\_enabled**

A bool property that controls the status of an artifical bit error rate to be applied

**property nicam\_bit\_error\_rate**

A float property that controls the artifical bit error rate.

valid range: 1.2E-7 .. 2E-3

**property nicam\_carrier\_enabled**

A bool property that controls if the NICAM carrier is switched on or off

**property nicam\_carrier\_frequency**

A float property that controls the frequency of the NICAM carrier

valid range: 33.05 MHz +/- 0.2 Mhz

**property nicam\_carrier\_level**

A float property that controls the value of the NICAM carrier

valid range: -40 .. -13 dB

**property nicam\_control\_bits**

A int property that controls the additional data in the NICAM modulator

valid range: 0 .. 3

**property nicam\_data**

A int property that controls the data in the NICAM modulator

valid range: 0 .. 2047

**property nicam\_intercarrier\_frequency**

A float property that controls the inter-carrier frequency of the NICAM carrier

valid range: 5 .. 9 MHz

**property nicam\_mode**

A string property that controls the signal type to be sent via NICAM

Possible values are:

Value	Meaning
MON	Mono sound + NICAM data
STER	Stereo sound
DUAL	Dual channel sound
DATA	NICAM data only

refer also to chapter 3.6.6.6 of the manual

**property nicam\_preemphasis\_enabled**

A bool property that controls the status of the J17 preemphasis

**property nicam\_source**

A string property that controls the signal source for NICAM

Possible values are:

Value	Meaning
INT	Internal audio generator(s)
EXT	External audio source
CW	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 questionable\_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 (vestigial sideband filter) in the vision modulator

**property vision\_videosignal\_enabled**

A bool property that controls if the video signal is switched on or off

**class** pymeasure.instruments.rohdeschwarz.sfm.**Sound\_Channel**(*instrument, number*)

Bases: object

Class object for the two sound channels

refere also to chapter 3.6.6.7 of the user manual

**property carrier\_enabled**

A bool property that controls if the audio carrier is switched on or off

**property carrier\_frequency**

A float property that controls the frequency of the sound carrier

valid range: 32 .. 46 MHz

**property carrier\_level**

A float property that controls the level of the audio carrier in dB relative to the vision carrier (0dB)

valid range: -34 .. -6 dB

**property deviation**

A int property that controls deviation of the selected audio signal

valid range: 0 .. 110 kHz

**property frequency**

A int property that controls the frequency of the internal sound generator

valid range: 300 Hz .. 15 kHz

**property modulation\_degree**

A float property that controls the modulation depth for the audio signal (Note: only for the use of AM in Standard L)

valid range: 0 .. 1 (100%)

**property modulation\_enabled**

A bool property that controls the audio modulation status

Value	Meaning
False	modulation disabled
True	modulation enabled

**property preemphasis\_enabled**

A bool property that controls if the preemphasis for the audio is switched on or off

**property preemphasis\_time**

A int property that controls if the mode of the preemphasis for the audio signal

Value	Meaning
50	50 us preemphasis
75	75 us preemphasis

**property use\_external\_source**

A bool property for the audio source selection

Value	Meaning
False	Internal audio generator(s)
True	External signal source

**values**(*command*, *\*\*kwargs*)

Reads a set of values from the instrument through the adapter, passing on any keyword arguments.

## 7.39.2 R&S FSL spectrum analyzer

### Connecting to the instrument via network

Once connected to the network, the instrument's IP address can be found by clicking the "Setup" button and navigating to "General Settings" -> "Network Address".

It can then be connected like this:

```
from pymeasure.instruments.rohdeschwarz import FSL
fsl = FSL("TCPIP::192.168.1.123::INSTR")
```

## Getting and setting parameters

Most parameters are implemented as properties, which means they can be read and written (getting and setting) in a consistent and simple way. If numerical values are provided, base units are used (s, Hz, dB, ...). Alternatively, the values can also be provided with a unit, e.g. "1.5 GHz" or "1.5GHz". Return values are always numerical.

```
# Getting the current center frequency
fsl.freq_center

9000000000.0
```

```
# Changing it to 10 MHz by providing the numerical value
fsl.freq_center = 10e6
```

```
# Verifying:
fsl.freq_center

10000000.0
```

```
# Changing it to 9 GHz by providing a string and verifying the result
fsl.freq_center = '9GHz'
fsl.freq_center

9000000000.0
```

```
# Setting the span to maximum
fsl.freq_span = '7 GHz'
```

## Reading a trace

We will read the current trace

```
x, y = fsl.read_trace()
```

## Markers

Markers are implemented as their own class. You can create them like this:

```
m1 = fsl.create_marker()
```

Set peak excursion:

```
m1.peak_excursion = 3
```

Set marker to a specific position:

```
m1.x = 10e9
```

Find the next peak to the left and get the level:

```
m1.to_next_peak('left')
m1.y

-34.9349060059
```

## Delta markers

Delta markers can be created by setting the appropriate keyword.

```
d2 = fsl.create_marker(is_delta_marker=True)
d2.name

'DELT2'
```

## Example program

Here is an example of a simple script for recording the peak of a signal.

```
m1 = fsl.create_marker() # create marker 1

# Set standard settings, set to full span
fsl.continuous_sweep = False
fsl.freq_span = '18 GHz'
fsl.res_bandwidth = "AUTO"
fsl.video_bandwidth = "AUTO"
fsl.sweep_time = "AUTO"

# Perform a sweep on full span, set the marker to the peak and some to that marker
fsl.single_sweep()
m1.to_peak()
m1.zoom('20 MHz')

# take data from the zoomed-in region
fsl.single_sweep()
x, y = fsl.read_trace()
```

```
class pymeasure.instruments.rohdeschwarz.fsl.FSL(adapter, name='Rohde&Schwarz FSL', **kwargs)
```

Bases: *Instrument*

Represents a Rohde&Schwarz FSL spectrum analyzer.

All physical values that can be set can either be as a string of a value and a unit (e.g. “1.2 GHz”) or as a float value in the base units (Hz, dBm, etc.).

### property attenuation

Attenuation in dB.

### continue\_single\_sweep()

Continue with single sweep with synchronization.

### property continuous\_sweep

Continuous (True) or single sweep (False)

**create\_marker**(*num=1, is\_delta\_marker=False*)

Create a marker.

**Parameters**

- **num** – The marker number (1-4)
- **is\_delta\_marker** – True if the marker is a delta marker, default is False.

**Returns**

The marker object.

**property freq\_center**

Center frequency in Hz.

**property freq\_span**

Frequency span in Hz.

**property freq\_start**

Start frequency in Hz.

**property freq\_stop**

Stop frequency in Hz.

**read\_trace**(*n\_trace=1*)

Read trace data.

**Parameters**

- **n\_trace** – The trace number (1-6). Default is 1.

**Returns**

2d numpy array of the trace data, [[frequency], [amplitude]].

**property res\_bandwidth**

Resolution bandwidth in Hz. Can be set to 'AUTO'

**single\_sweep**()

Perform a single sweep with synchronization.

**property sweep\_time**

Sweep time in s. Can be set to 'AUTO'.

**property trace\_mode**

Trace mode ('WRIT', 'MAXH', 'MINH', 'AVER' or 'VIEW')

**property video\_bandwidth**

Video bandwidth in Hz. Can be set to 'AUTO'

### 7.39.3 R&S HMP4040 Power Supply

**class** pymeasure.instruments.rohdeschwarz.hmp.HMP4040(*adapter, \*\*kwargs*)

Bases: [Instrument](#)

Represents a Rohde&Schwarz HMP4040 power supply.

**beep**()

Emit a single beep from the instrument.

**clear\_sequence(channel)**

Clear the sequence of the selected channel.

**property control\_method**

Enables manual front panel ('LOC'), remote ('REM') or manual/remote control ('MIX') control or locks the the front panel control ('RWL').

**property current**

Output current in A. Range depends on instrument type.

**property current\_step**

Current step in A.

**current\_to\_max()**

Set current of the selected channel to its maximum value.

**current\_to\_min()**

Set current of the selected channel to its minimum value.

**load\_sequence(slot)**

Load a saved waveform from internal memory (slot 1, 2 or 3).

**property max\_current**

Maximum current in A.

**property max\_voltage**

Maximum voltage in V.

**property measured\_current**

Measured current in A.

**property measured\_voltage**

Measured voltage in V.

**property min\_current**

Minimum current in A.

**property min\_voltage**

Minimum voltage in V.

**property output\_enabled**

Set the output on or off or check the output status.

**property repetitions**

Number of repetitions (0..255). If 0 is entered, the sequence is repeated indefinitely.

**save\_sequence(slot)**

Save the sequence defined in the sequence property to internal memory (slot 1, 2 or 3).

**property selected\_channel**

Selected channel.

**property selected\_channel\_active**

Set the selected channel to active or inactive or check its status.

**property sequence**

Define sequence of triplets of voltage (V), current (A) and dwell time (s).

**set\_channel\_state**(*channel*, *state*)

Set the state of the channel to active or inactive.

**Parameters**

- **channel** (*int*) – Channel number to set the state of.
- **state** (*bool*) – State of the channel, i.e. True for active, False for inactive.

**start\_sequence**(*channel*)

Start the sequence of the selected channel.

**step\_current\_down**()

Decreases current by one step.

**step\_current\_up**()

Increase current by one step.

**step\_voltage\_down**()

Decrease voltage by one step.

**step\_voltage\_up**()

Increase voltage by one step.

**stop\_sequence**(*channel*)

Stop the sequence defined in the sequence property of the selected channel.

**transfer\_sequence**(*channel*)

Transfer the sequence defined in the sequence property to the selected channel.

**property version**

The SCPI version the instrument's command set complies with.

**property voltage**

Output voltage in V. Increment 0.001 V.

**property voltage\_and\_current**

Output voltage (V) and current (A).

**property voltage\_step**

Voltage step in V. Default 1 V.

**voltage\_to\_max**()

Set voltage of the selected channel to its maximum value.

**voltage\_to\_min**()

Set voltage of the selected channel to its minimum value.

## 7.40 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](#).

### 7.40.1 Siglent Technologies Base Class

```
class pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDBase(adapter,
                                                                    name='Siglent
                                                                    SPDxxxxX instrument
                                                                    Base Class',
                                                                    **kwargs)
```

Bases: *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**

Read the error code and information of the instrument.

**Type**

string

**property fw\_version**

Read the software version of the instrument.

**Type**

string

**recall\_config**(*index*)

Recall a config from memory.

**Parameters**

**index** – int: index of the location from which to recall the configuration

**save\_config**(*index*)

Save the current config to memory.

**Parameters**

**index** – int: index of the location to save the configuration

**property selected\_channel**

Control the selected channel of the instrument.

:type : int (dynamic)

**shutdown**()

Ensure that the voltage is turned to zero and disable the output.

**property system\_status\_code**

Read the system status register.

**Type**

*SystemStatusCode*

```
class pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDSingleChannelBase(adapter,
                                                                                   name='Siglent
                                                                                   SPDxxxxX
                                                                                   in-
                                                                                   stru-
                                                                                   ment
                                                                                   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.SPDSingleChannel(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=None,
*, module=None,
qualified_name=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

## 7.40.2 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](#)

### 7.40.3 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*

## 7.41 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*.

### 7.41.1 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 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 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  $\mu$ A (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=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer**(*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.

## 7.41.2 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 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 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  $\mu$ A (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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**wait\_for\_buffer(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.

## 7.42 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](#).

### 7.42.1 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.

## 7.42.2 SR570 Lock-in Amplifier

**class** `pymeasure.instruments.srs.SR570`(*adapter*, *name*='Stanford Research Systems SR570 Lock-in amplifier', \*\*kwargs)

Bases: `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

### 7.42.3 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.

**property channel1**

A string property that represents the type of Channel 1, taking the values X, R, X Noise, Aux In 1, or Aux In 2. This property can be set.

**property channel2**

A string property that represents the type of Channel 2, taking the values Y, Theta, Y Noise, Aux In 3, or Aux In 4. This property can be set.

**property dac1**

A floating point property that controls the output of Aux output 1 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

**property dac2**

A floating point property that controls the output of Aux output 2 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

**property dac3**

A floating point property that controls the output of Aux output 3 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

**property dac4**

A floating point property that controls the output of Aux output 4 in Volts, taking values between -10.5 V and +10.5 V. This property can be set.

**property err\_status**

Reads the value of the lockin error (ERR) status byte. Returns an IntFlag type with positions within the string corresponding to different error flags:

Bit	Status
0	unused
1	backup error
2	RAM error
3	unused
4	ROM error
5	GPIB error
6	DSP error
7	DSP error

**property filter\_slope**

An integer property that controls the filter slope, which can take on the values 6, 12, 18, and 24 dB/octave. Values are truncated to the next highest level if they are not exact.

**property filter\_synchronous**

A boolean property that controls the synchronous filter. This property can be set. Allowed values are: True or False

**property frequency**

A floating point property that represents the lock-in frequency in Hz. This property can be set.

**get\_buffer(channel=1, start=0, end=None)**

Acquires the 32 bit floating point data through binary transfer

**get\_scaling(channel)**

Returns the offset present 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 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

**property magnitude**

Reads the magnitude in Volts.

**output\_conversion(channel)**

Returns a function that can be used to determine the signal from the channel output (X, Y, or R)

**property phase**

A floating point property that represents the lock-in phase in degrees. This property can be set.

**quick\_range()**

While the magnitude is out of range, increase the sensitivity by one setting

**property reference\_source**

A string property that controls the reference source. Allowed values are: ['External', 'Internal']

**property reference\_source\_trigger**

A string property that controls the reference source triggering. Allowed values are: ['SINE', 'POS EDGE', 'NEG EDGE']

**property sample\_frequency**

Gets the sample frequency in Hz

**property sensitivity**

A floating point property that controls the sensitivity in Volts, which can take discrete values from 2 nV to 1 V. Values are truncated to the next highest level if they are not exact.

**set\_scaling(channel, precent, expand=0)**

Sets the offset of a channel (X=1, Y=2, R=3) to a certain precent (-105% to 105%) of the signal, with an optional expansion term (0, 10=1, 100=2)

**property sine\_voltage**

A floating point property that represents the reference sine-wave voltage in Volts. This property can be set.

**snap(val1='X', val2='Y', \*vals)**

Method that records and retrieves 2 to 6 parameters at a single instant. The parameters can be one of: X, Y, R, Theta, Aux In 1, Aux In 2, Aux In 3, Aux In 4, Frequency, CH1, CH2. Default is "X" and "Y".

**Parameters**

- **val1** – first parameter to retrieve
- **val2** – second parameter to retrieve
- **vals** – other parameters to retrieve (optional)

**property theta**

Reads the theta value in degrees.

**property time\_constant**

A floating point property that controls the time constant in seconds, which can take discrete values from 10 microseconds to 30,000 seconds. Values are truncated to the next highest level if they are not exact.

**wait\_for\_buffer**(*count*, *has\_aborted*=<function SR830.<lambda>>, *timeout*=60, *timestep*=0.01)

Wait for the buffer to fill a certain count

**property x**

Reads the X value in Volts.

**property xy**

Reads the X and Y values in Volts.

**property y**

Reads the Y value in Volts.

## 7.42.4 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 extfrequency**

Returns the external frequency in Hz.

**property filer\_synchronous**

A string property that represents the synchronous filter. This property can be set. Allowed values are: ['Off', 'On']

**property filter\_advanced**

A string property that represents the advanced filter. This property can be set. Allowed values are: ['Off', 'On']

**property filter\_slope**

A integer property that sets the filter slope to 6 dB/oct(i=0), 12 DB/oct(i=1), 18 dB/oct(i=2), 24 dB/oct(i=3).

**property frequency**

A floating point property that represents the lock-in frequency in Hz. This property can be set.

**property frequencypreset1**

A floating point property that represents the preset frequency for the F1 preset button. This property can be set.

**property frequencypreset2**

A floating point property that represents the preset frequency for the F2 preset button. This property can be set.

**property frequencypreset3**

A floating point property that represents the preset frequency for the F3 preset button. This property can be set.

**property frequencypreset4**

A floating point property that represents the preset frequency for the F4 preset button. This property can be set.

**property front\_panel**

Turns the front panel blanking on(i=0) or off(i=1).

**property get\_noise\_bandwidth**

Returns the equivalent noise bandwidth, in hertz.

**property get\_signal\_strength\_indicator**

Returns the signal strength indicator.

**property gettimebase**

Returns the current 10 MHz timebase source.

**property harmonic**

An integer property that controls the harmonic that is measured. Allowed values are 1 to 99. Can be set.

**property harmonicdual**

An integer property that controls the harmonic in dual reference mode that is measured. Allowed values are 1 to 99. Can be set.

**property horizontal\_time\_div**

A integer property for the horizontal time/div according to the following table: ['0=0.5s', '1=1s', '2=2s', '3=5s', '4=10s', '5=30s', '6=1min', '7=2min', '8=5min', '9=10min', '10=30min', '11=1hour', '12=2hour', '13=6hour', '14=12hour', '15=1day', '16=2days']

**property input\_coupling**

A string property that represents the input coupling. This property can be set. Allowed values are: ['AC', 'DC']

**property input\_current\_gain**

A string property that represents the current input gain. This property can be set. Allowed values are: ['1MEG', '100MEG']

**property input\_range**

A string property that represents the input range. This property can be set. Allowed values are: ['1V', '300M', '100M', '30M', '10M']

**property input\_shields**

A string property that represents the input shield grounding. This property can be set. Allowed values are: ['Float', 'Ground']

**property input\_signal**

A string property that represents the signal input. This property can be set. Allowed values are: ['VOLT', 'CURR', 'voltage', 'current']

**property input\_voltage\_mode**

A string property that represents the voltage input mode. This property can be set. Allowed values are: ['A', 'A-B']

**property internalfrequency**

A floating property that represents the internal lock-in frequency in Hz This property can be set.

**property magnitude**

Reads the magnitude in Volts.

**property parameter\_DAT1**

A integer property that assigns a parameter to data channel 1(green). This parameters can be set. Allowed values are:['i=', '0=Xoutput', '1=Youtput', '2=Routput', 'Thetaoutput', '4=Aux IN1', '5=Aux IN2', '6=Aux IN3', '7=Aux IN4', '8=Xnoise', '9=Ynoise', '10=AUXOut1', '11=AuxOut2', '12=Phase', '13=Sine Out amplitude', '14=DCLLevel', '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=DCLLevel', '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=DCLLevel', '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=DCLLevel', '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:['500HMS', '1MEG']

**property reference\_source**

A string property that represents the reference source. This property can be set. Allowed values are:['INT', 'EXT', 'DUAL', 'CHOP']

**property reference\_triggermode**

A string property that represents the external reference trigger mode. This property can be set. Allowed values are:['SIN', 'POS', 'NEG', 'POSTTL', 'NEGTTL']

**property screen\_layout**

A integer property that Sets the screen layout to trend(i=0), full strip chart history(i=1), half strip chart history(i=2), full FFT(i=3), half FFT(i=4) or big numerical(i=5).

**screenshot()**

Take screenshot on device The DCAP command saves a screenshot to a USB memory stick. This command is the same as pressing the [Screen Shot] key. A USB memory stick must be present in the front panel USB port.

**property 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 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

#### **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

## 7.43 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](#).

### 7.43.1 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=None, \*, 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

## 7.44 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](#).

### 7.44.1 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 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 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 the actual output voltage.

**property voltage\_setpoint**

Control the programmed (set) output voltage.(dynamic)

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command, \*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command, values, \*args, \*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content, \*\*kwargs*)

Write the bytes *content* to the instrument.

## 7.44.2 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 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 the actual output voltage.

**property voltage\_setpoint**

Control the programmed (set) output voltage.(dynamic)

**wait\_for**(*query\_delay=0*)

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*command, \*\*kwargs*)

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values**(*command, values, \*args, \*\*kwargs*)

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs** (*\*args,*) – Further arguments to hand to the Adapter.

**write\_bytes**(*content, \*\*kwargs*)

Write the bytes *content* to the instrument.

## 7.45 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](#).

### 7.45.1 TDS2000 Oscilloscope

```
class pymeasure.instruments.tektronix.TDS2000(adapter, name='Tektronix TDS 2000 Oscilloscope',
                                              **kwargs)
```

Bases: [Instrument](#)

Represents the Tektronix TDS 2000 Oscilloscope and provides a high-level for interacting with the instrument

### 7.45.2 AFG3152C Arbitrary function generator

```
class pymeasure.instruments.tektronix.AFG3152C(adapter, name='Tektronix AFG3152C arbitrary
function generator', **kwargs)
```

Bases: [Instrument](#)

Represents the Tektronix AFG 3000 series (one or two channels) arbitrary function generator and provides a high-level for interacting with the instrument.

```
afg=AFG3152C("GPIB::1")      # AFG on GPIB 1
afg.reset()                   # Reset to default
afg.ch1.shape='sinusoidal'    # Sinusoidal shape
afg.ch1.unit='VPP'            # Sets CH1 unit to VPP
afg.ch1.amp_vpp=1             # Sets the CH1 level to 1 VPP
afg.ch1.frequency=1e3         # Sets the CH1 frequency to 1KHz
afg.ch1.enable()              # Enables the output from CH1
```

## 7.46 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.

### 7.46.1 Teledyne T3AFG Arbitrary Waveform Generator

```
class pymeasure.instruments.teledyne.TeledyneT3AFG(adapter, name='Teledyne T3AFG', **kwargs)
```

Bases: [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 initially.

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 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 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=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write(command, \*\*kwargs)**

Write a string command to the instrument appending *write\_termination*.

**Parameters**

- **command** – command string to be sent to the instrument
- **kwargs** – Keyword arguments for the adapter.

**write\_binary\_values(command, values, \*args, \*\*kwargs)**

Write binary values to the device.

**Parameters**

- **command** – Command to send.
- **values** – The values to transmit.
- **\*\*kwargs (\*args,)** – Further arguments to hand to the Adapter.

**write\_bytes(content, \*\*kwargs)**

Write the bytes *content* to the instrument.

**class** pymeasure.instruments.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. If your device is missing, the base class might already work well enough. If adding a new device, these base classes should limit the amount of new code necessary.

## 7.46.2 Teledyne Oscilloscope base classes

### Teledyne Oscilloscope

```
class pymeasure.instruments.teledyne.TeledyneOscilloscope(adapter, name='Teledyne Oscilloscope',
                                                         **kwargs)
```

Bases: *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 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.

**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 programming guide)
- “hold\_value1”: hold value1 (refer to page 172 of programming guide)
- “hold\_value2”: hold value2 (refer to page 172 of programming 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/run 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 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)

## 7.47 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](#).

### 7.47.1 Temptronic Base Class

**class** `pymeasure.instruments.temptronic.ATSBBase(adapter, name='ATSBBase', **kwargs)`

Bases: `Instrument`

The base class for Temptronic ATSXXX instruments.

**property** `air_temperature`

Read air temperature in 0.1 °C increments.

**Type**  
float

**at\_temperature()**

**Returns**  
True if at temperature.

**property** `auxiliary_condition_code`

Read out auxiliary condition status register.

**Type**  
int

Relevant flags are:

Bit	Meaning
10	None
9	Ramp mode
8	Mode: 0 programming, 1 manual
7	None
6	TS status: 0 start-up, 1 ready
5	Flow: 0 off, 1 on
4	Sense mode: 0 air, 1 DUT
3	Compressor: 0 on, 1 off (heating possible)
2	Head: 0 lower, upper
1	None
0	None

Refere to chapter 4 in the manual

**clear()**

Clear device-specific errors.

See [error\\_code](#) for further information.

**property compressor\_enable**

True enables compressors, False disables it.

**Type**

Boolean

**configure**(*temp\_window=1, dut\_type='T', soak\_time=30, dut\_constant=100, temp\_limit\_air\_low=-60, temp\_limit\_air\_high=220, temp\_limit\_air\_dut=50, maximum\_test\_time=1000*)

Convenience method for most relevant configuration properties.

**Parameters**

- **dut\_type** – string: indicating which DUT type to use
- **soak\_time** – float: elapsed time in soak\_window before settling is indicated
- **soak\_window** – float: Soak window size or temperature settlings bounds (K)
- **dut\_constant** – float: time constant of DUT, higher values indicate higher thermal mass
- **temp\_limit\_air\_low** – float: minimum flow temperature limit (°C)
- **temp\_limit\_air\_high** – float: maximum flow temperature limit (°C)
- **temp\_limit\_air\_dut** – float: allowed temperature difference (K) between DUT and Flow
- **maximum\_test\_time** – float: maximum test time (seconds) for a single temperature point (safety)

**Returns**

self

**property copy\_active\_setup\_file**

Copy active setup file (0) to setup n (1 - 12).

**Type**

int

**property current\_cycle\_count**

Read the number of cycles to do

**Type**

int

**property cycling\_enable**

CYCL Start/stop cycling.

**Type**

bool

cycling\_enable = True (start cycling) cycling\_enable = False (stop cycling)

**cycling\_stopped()****Returns**

True if cycling has stopped.

**property dut\_constant**

Control thermal constant (default 100) of DUT.

**Type**

float

Lower values indicate lower thermal mass, higher values indicate higher thermal mass respectively.

**property dut\_mode**

On enables DUT mode, OFF enables air mode

**Type**

string

**property dut\_temperature**

Read DUT temperature, in 0.1 °C increments.

**Type**

float

**property dut\_type**

Control DUT sensor type.

**Type**

string

Possible values are:

String	Meaning
''	no DUT
'T'	T-DUT
'K'	K-DUT

Warning: If in DUT mode without DUT being connected, TS flags DUT error

**property dynamic\_temperature\_setpoint**

Read the dynamic temperature setpoint.

**Type**

float

**property enable\_air\_flow**

Set TS air flow.

True enables air flow, False disables it

**Type**

bool

**end\_of\_all\_cycles()****Returns**

True if cycling has stopped.

**end\_of\_one\_cycle()****Returns**

True if TS is at end of one cycle.

**end\_of\_test()**

**Returns**

True if TS is at end of test.

**enter\_cycle()**

Enter Cycle by sending RMPC 1.

**Returns**

self

**enter\_ramp()**

Enter Ramp by sending RMPS 0.

**Returns**

self

**property error\_code**

Read the device-specific error register (16 bits).

**Type**

ErrorCode

**error\_status()**

Returns error status code (maybe used for logging).

**Returns**

ErrorCode

**property head**

Control TS head position.

**Type**

string

down: transfer head to lower position up: transfer head to elevated position

**property learn\_mode**

Control DUT automatic tuning (learning).

**Type**

bool False: off True: automatic tuning on

**property load\_setup\_file**

loads setup file SFIL.

Valid range is between 1 to 12.

**Type**

int

**property local\_lockout**

True disables TS GUI, False enables it.

**property main\_air\_flow\_rate**

Read main nozzle air flow rate in liters/sec.

**property maximum\_test\_time**

Control maximum allowed test time (s).

**Type**

float

This prevents TS from staying at a single temperature forever. Valid range: 0 to 9999

**property mode**

Returns a string indicating what the system is doing at the time the query is processed.

**Type**

string

(dynamic)

**next\_setpoint()**

Step to the next setpoint during temperature cycling.

**not\_at\_temperature()**

**Returns**

True if not at temperature.

**property nozzle\_air\_flow\_rate**

Read main nozzle air flow rate in scfm.

**property ramp\_rate**

Control ramp rate (K / min).

**Type**

float

allowed values: nn.n: 0 to 99.9 in 0.1 K per minute steps. nnnn: 100 to 9999 in 1 K per minute steps.

**property remote\_mode**

True disables TS GUI but displays a “Return to local” switch.

**reset()**

Reset (force) the System to the Operator screen.

**Returns**

self

**property set\_point\_number**

Select a setpoint to be the current setpoint.

**Type**

int

Valid range is 0 to 17 when on the Cycle screen or or 0 to 2 in case of operator screen (0=hot, 1=ambient, 2=cold).

**set\_temperature(*set\_temp*)**

sweep to a specified setpoint.

**Parameters**

**set\_temp** – target temperature for DUT (float)

**Returns**

self

**shutdown(*head=False*)**

Turn down TS (flow and remote operation).

**Parameters**

**head** – Lift head if True

**Returns**

self

**start**(*enable\_air\_flow=True*)

start TS in remote mode.

**Parameters****enable\_air\_flow** – flow starts if True**Returns**

self

**property temperature**

Read current temperature with 0.1 °C resolution.

**Type**

float

Temperature readings origin depends on *dut\_mode* setting. Reading higher than 400 (°C) indicates invalidity.

**property temperature\_condition\_status\_code**

Temperature condition status register.

**Type***TemperatureStatusCode***property temperature\_event\_status**

temperature event status register.

**Type***TemperatureStatusCode*

Hint: Reading will clear register content.

**property temperature\_limit\_air\_dut**

Air to DUT temperature limit.

**Type**

float

Allowed difference between nozzle air and DUT temperature during settling. Valid range between 10 to 300 °C in 1 degree increments.

**property temperature\_limit\_air\_high**

upper air temperature limit.

**Type**

float

Valid range between 25 to 255 (°C). Setpoints above current value cause “out of range” error in TS.

**property temperature\_limit\_air\_low**

Control lower air temperature limit.

**Type**

float

Valid range between -99 to 25 (°C). Setpoints below current value cause “out of range” error in TS. (dynamic)

**property temperature\_setpoint**

Set or get selected setpoint's temperature.

**Type**

float

Valid range is -99.9 to 225.0 (°C) or as indicated by `temperature_limit_air_high` and `temperature_limit_air_low`. Use convenience function `set_temperature()` to prevent unexpected behavior.

**property temperature\_setpoint\_window**

Setpoint's temperature window.

**Type**

float

Valid range is between 0.1 to 9.9 (°C). Temperature status register flags at `temperature` in case soak time elapsed while temperature stays in between bounds given by this value around the current setpoint.

**property temperature\_soak\_time**

Set the soak time for the currently selected setpoint.

**Type**

float

Valid range is between 0 to 9999 (s). Lower values shorten cycle times. Higher values increase cycle times, but may reduce settling errors. See `temperature_setpoint_window` for further information.

**property total\_cycle\_count**

Set or read current cycle count (1 - 9999).

**Type**

int

Sending 0 will stop cycling

**wait\_for\_settling**(*time\_limit=300*)

block script execution until TS is settled.

**Parameters**

**time\_limit** – set the maximum blocking time within TS has to settle (float).

**Returns**

self

Script execution is blocked until either TS has settled or `time_limit` has been exceeded (float).

```
class pymeasure.instruments.temptronic.temptronic_base.TemperatureStatusCode(value,
                                                                              names=None, *,
                                                                              module=None,
                                                                              qual-
                                                                              name=None,
                                                                              type=None,
                                                                              start=1, bound-
                                                                              ary=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=None, *,
                                                                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

### 7.47.2 Temptronic ATS525 Thermostream

```
class pymeasure.instruments.temptronic.ATS525(adapter, name='Temptronic ATS-525 Thermostream',
                                              **kwargs)
```

Bases: [ATSBBase](#)

Represent the TemptronicATS525 instruments.

**property system\_current**

Operating current.

### 7.47.3 Temptronic ATS545 Thermostream

```
class pymeasure.instruments.temptronic.ATS545(adapter, name='Temptronic ATS-545 Thermostream',  
                                              **kwargs)
```

Bases: [ATSBBase](#)

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

### 7.47.4 Temptronic ECO560 Thermostream

```
class pymeasure.instruments.temptronic.ECO560(adapter, name='Temptronic ECO-560 Thermostream',  
                                              **kwargs)
```

Bases: [ATSBBase](#)

Represent the TemptronicECO560 instruments.

**copy\_active\_setup\_file** = None

## 7.48 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](#).

### 7.48.1 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**

Simultaneous control of voltage (volts) and current (amps). Values need to be supplied as tuple of (voltage, current). Depending on whether the instrument is in constant current or constant voltage mode, the values achieved by the instrument will differ from the ones set.

#### **check\_errors()**

Logs 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()**

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 current**

Reads the current (in Ampere) the dc power supply is putting out.

#### **property current\_limit**

A floating point property that controls the source current in amps. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual current achieved.

**property error**

Returns a tuple of an error code and message from a single error.

**property id**

Get the identification of the instrument.

**property options**

Get the device options installed.

**property output\_enabled**

A boolean property that controls whether the source is enabled, takes values True or False.

**property power**

Reads the power (in Watt) the dc power supply is putting out.

**read(\*\*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()**

Disable output, call parent function

**property status**

Get the status byte and Master Summary Status bit.

**property voltage**

Reads the voltage (in Volt) the dc power supply is putting out.

**property voltage\_setpoint**

A floating point property that controls the source voltage in volts. This is not checked against the allowed range. Depending on whether the instrument is in constant current or constant voltage mode, this might differ from the actual voltage achieved.

**wait\_for(query\_delay=0)**

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

**Parameters**

**query\_delay** – Delay between writing and reading in seconds.

**write**(*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.

## 7.49 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](#).

### 7.49.1 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*=None, *\**, *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`

Reads the instrument identification

**Returns**

String

**initialize\_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**

Gets 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**

A floating point property that controls the setpoint of the oven in Celsius. This property can be set. “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**

Reads 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.

## 7.50 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](#).

### 7.50.1 Thorlabs PM100USB Powermeter

```
class pymeasure.instruments.thorlabs.ThorlabsPM100USB(adapter, name='ThorlabsPM100USB
                                                    powermeter', **kwargs)
```

Bases: [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

### 7.50.2 Thorlabs Pro 8000 modular laser driver

```
class pymeasure.instruments.thorlabs.ThorlabsPro8000(adapter, name='Thorlabs Pro 8000',
                                                         **kwargs)
```

Bases: [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)

## 7.51 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](#).

### 7.51.1 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 address 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

## 7.51.2 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.thyracont 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**

**address** – The device address in the range 1-16.

**class Sources**(*value, names=None, \*, 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=0*)

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=0*)

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** (*int*) – Time to wait between writing and reading.

**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 sensor 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.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

## 7.52 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*.

### 7.52.1 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\_continuous()**

Enable continuous emission mode.

**enable\_pulsing()**

Enable pulsing mode.

The optical output is controlled by a digital input signal on a dedicated connector 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  $\mu\text{W}$  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.i beamsmart.**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  $\mu\text{W}$  (float up to 200000).

## 7.53 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](#).

### 7.53.1 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.

**id = None**

**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**

Switch off a set of channels. See [switch\\_on](#) for more details.

**property switch\_on**

” Switch on a set of channels. 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=None, *, module=None,
qualname=None, type=None, start=1,
boundary=None)
```

Bases: `IntFlag`

Use to identify switch channels.

## 7.54 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](#).

### 7.54.1 Yokogawa 7651 Programmable Supply

```
class pymeasure.instruments.yokogawa.Yokogawa7651(adapter, name='Yokogawa 7651 Programmable
DC Source', **kwargs)
```

Bases: `Instrument`

Represents the Yokogawa 7651 Programmable DC Source and provides a high-level for interacting with the instrument.

```
yoko = Yokogawa7651("GPIB::1")

yoko.apply_current()           # Sets up to source current
yoko.source_current_range = 10e-3 # Sets the current range to 10 mA
yoko.compliance_voltage = 10     # Sets the compliance voltage to 10 V
yoko.source_current = 0          # Sets the source current to 0 mA

yoko.enable_source()           # Enables the current output
yoko.ramp_to_current(5e-3)      # Ramps the current to 5 mA

yoko.shutdown()                # Ramps the current to 0 mA and disables output
```

**apply\_current**(*max\_current=0.001*, *compliance\_voltage=1*)

Configures the instrument to apply a source current, which can take optional parameters that defer to the `source_current_range` and `compliance_voltage` properties.

**apply\_voltage**(*max\_voltage=1*, *compliance\_current=0.01*)

Configures the instrument to apply a source voltage, which can take optional parameters that defer to the `source_voltage_range` and `compliance_current` properties.

**property compliance\_current**

A floating point property that sets the compliance current in Amps, which can take values from 5 to 120 mA.

**property compliance\_voltage**

A floating point property that sets the compliance voltage in Volts, which can take values between 1 and 30 V.

**disable\_source()**

Disables the source of current or voltage depending on the configuration of the instrument.

**enable\_source()**

Enables the source of current or voltage depending on the configuration of the instrument.

**property id**

Returns the identification of the instrument

**ramp\_to\_current(*current*, *steps*=25, *duration*=0.5)**

Ramps the current to a value in Amps by traversing a linear spacing of current steps over a duration, defined in seconds.

**Parameters**

- **steps** – A number of linear steps to traverse
- **duration** – A time in seconds over which to ramp

**ramp\_to\_voltage(*voltage*, *steps*=25, *duration*=0.5)**

Ramps the voltage to a value in Volts by traversing a linear spacing of voltage steps over a duration, defined in seconds.

**Parameters**

- **steps** – A number of linear steps to traverse
- **duration** – A time in seconds over which to ramp

**shutdown()**

Shuts down the instrument, and ramps the current or voltage to zero before disabling the source.

**property source\_current**

A floating point property that controls the source current in Amps, if that mode is active.

**property source\_current\_range**

A floating point property that sets the current voltage range in Amps, which can take values: 1 mA, 10 mA, and 100 mA. Currents are truncated to an appropriate value if needed.

**property source\_enabled**

Reads a boolean value that is True if the source is enabled, determined by checking if the 5th bit of the OC flag is a binary 1.

**property source\_mode**

A string property that controls the source mode, which can take the values 'current' or 'voltage'. The convenience methods [apply\\_current\(\)](#) and [apply\\_voltage\(\)](#) can also be used.

**property source\_voltage**

A floating point property that controls the source voltage in Volts, if that mode is active.

**property source\_voltage\_range**

A floating point property that sets the source voltage range in Volts, which can take values: 10 mV, 100 mV, 1 V, 10 V, and 30 V. Voltages are truncated to an appropriate value if needed.

## 7.54.2 Yokogawa GS200 Source

```
class pymeasure.instruments.yokogawa.YokogawaGS200(adapter, name='Yokogawa GS200 Source',  
                                                    **kwargs)
```

Bases: *Instrument*

Represents the Yokogawa GS200 source and provides a high-level interface for interacting with the instrument.

### property **current\_limit**

Floating point number that controls the current limit. “Limit” refers to maximum value of the electrical value that is conjugate to the mode (current is conjugate to voltage, and vice versa). Thus, current limit is only applicable when in ‘voltage’ mode

### property **source\_enabled**

A boolean property that controls whether the source is enabled, takes values True or False.

### property **source\_level**

Floating point number that controls the output level, either a voltage or a current, depending on the source mode.

### property **source\_mode**

String property that controls the source mode. Can be either ‘current’ or ‘voltage’.

### property **source\_range**

Floating point number that controls the range (either in voltage or current) of the output. “Range” refers to the maximum source level.

### **trigger\_ramp\_to\_level**(level, ramp\_time)

Ramp the output level from its current value to “level” in time “ramp\_time”. This method will NOT wait until the ramp is finished (thus, it will not block further code evaluation).

#### Parameters

- **level** (*float*) – final output level
- **ramp\_time** (*float*) – time in seconds to ramp

#### Returns

None

### property **voltage\_limit**

Floating point number that controls the voltage limit. “Limit” refers to maximum value of the electrical value that is conjugate to the mode (current is conjugate to voltage, and vice versa). Thus, voltage limit is only applicable when in ‘current’ mode



## CONTRIBUTING

Contributions to the instrument repository and the main code base are highly encouraged. This section outlines the basic work-flow for new contributors.

### 8.1 Using the development version

New features are added to the development version of PyMeasure, hosted on [GitHub](#). We use [Git version control](#) to track and manage changes to the source code. On Windows, we recommend using [GitHub Desktop](#). Make sure you have an appropriate version of Git (or GitHub Desktop) installed and that you have a GitHub account.

In order to add your feature, you need to first [fork](#) PyMeasure. This will create a copy of the repository under your GitHub account.

The instructions below assume that you have set up Anaconda, as described in the [Quick Start guide](#) and describe the terminal commands necessary. If you are using GitHub Desktop, take a look through [their documentation](#) to understand the corresponding steps.

Clone your fork of PyMeasure `your-github-username/pymasure`. In the following terminal commands replace your desired path and GitHub username.

```
cd /path/for/code
git clone https://github.com/your-github-username/pymasure.git
```

If you had already installed PyMeasure using `pip`, make sure to uninstall it before continuing.

```
pip uninstall pymasure
```

Install PyMeasure in the editable mode.

```
cd /path/for/code/pymasure
pip install -e .
```

This will allow you to edit the files of PyMeasure and see the changes reflected. Make sure to reset your notebook kernel or Python console when doing so. Now you have your own copy of the development version of PyMeasure installed!

## 8.2 Working on a new feature

We use branches in Git to allow multiple features to be worked on simultaneously, without causing conflicts. The master branch contains the stable development version. Instead of working on the master branch, you will create your own branch off the master and merge it back into the master when you are finished.

Create a new branch for your feature before editing the code. For example, if you want to add the new instrument “Extreme 5000” you will make a new branch “dev/extreme-5000”.

```
git branch dev/extreme-5000
```

You can also [make a new branch](#) on GitHub. If you do so, you will have to fetch these changes before the branch will show up on your local computer.

```
git fetch
```

Once you have created the branch, change your current branch to match the new one.

```
git checkout dev/extreme-5000
```

Now you are ready to write your new feature and make changes to the code. To ensure consistency, please follow the [coding standards for PyMeasure](#). Use `git status` to check on the files that have been changed. As you go, commit your changes and push them to your fork.

```
git add file-that-changed.py
git commit -m "A short description about what changed"
git push
```

## 8.3 Making a pull request

While you are working, it is helpful to start a pull request (PR) targeting the master branch of `pymeasure/pymeasure`. This will allow you to discuss your feature with other contributors. We encourage you to start this pull request already after your first commit. You may mark a pull request as a draft, if it is in an early state.

[Start a pull request on the PyMeasure GitHub page](#).

There is some automation in place to run the unit tests and check some coding standards. Annotations in the “Files changed” tab indicate problems for you to correct (e.g. linting or docstring warnings).

Your pull-request will be reviewed by the PyMeasure maintainers. Frequently there is some iteration and discussion based on that feedback until a pull request can be merged. This will happen either in the conversation tab or in inline code comments.

Be aware that due to maintainer manpower limitations it might take a long time until PRs get reviewed and/or merged. In general, review effort scales badly with PR size. Therefore, **smaller PRs are much preferred**. Try to limit your contribution to one “aspect”, e.g. one instrument (or a few if closely related), one bug fix, or one feature contribution.

If you placed your contribution in a separate branch as suggested above, you can easily use your contribution in the meantime – just check out your feature branch instead of *master*.

## 8.4 Unit testing

Unit tests are run each time a new commit is made to a branch. The purpose is to catch changes that break the current functionality, by testing each feature unit. PyMeasure relies on `pytest` to perform 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!



## **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.



## 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. For a new manufacturer, the manufacturer should also be added to `pymeasure/pymeasure/instruments/__init__.py`.

#### 10.1.2 Add test files

Test files (pytest) for each instrument are highly encouraged, as they help verify the code and implement changes. Testing new code parts with a test (Test Driven Development) is a good way for fast and good programming, as you catch errors early on.

```
pymeasure/tests/instruments/extreme/  
|--> test_extreme5000.py
```

### 10.1.3 Adding documentation

Documentation for each instrument is required, and helps others understand the features you have implemented. Add a new reStructuredText file to the documentation.

```
pymeaure/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 `pymeaure/docs/api/instruments/index.rst`.

## 10.2 Instrument file

All standard instruments should be child class of `Instrument`. This provides the basic functionality for working with `Adapters`, which perform the actual communication.

The most basic instrument, for our “Extreme 5000” example starts like this:

```
#
# This file is part of the PyMeasure package.
#
# Copyright (c) 2013-2023 PyMeasure Developers
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# from pymeaure.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`, `lim`, `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.

#### 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}')
...
>>>
```

(continues on next page)

(continued from previous page)

```
<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 `*IDN?` command, we can write the following property to be added after the `def __init__` line in our above example class, or added to the class after the fact as in the code here:

```
Extreme5000.cell_temp = Instrument.measurement(
    ":TEMP?",
    """Measure the temperature of the reaction cell.""",
)
```

You will notice that a documentation string is required, see [Docstrings](#) for details.

When we use this property we will get the temperature of the reaction cell.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.cell_temp # Sends ":TEMP?" to the device
127.2
```

The [Instrument.control](#) function extends this behavior by creating a property that you can read and set. For example, if our “Extreme 5000” has the `:VOLT?` and `:VOLT <float>` commands that are in Volts, we can write the following property.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float)."""
)
```

You will notice that we use the [Python string format %g](#) to format passed-through values as floating point.

We can use this property to set the voltage to 100 mV, which will send the appropriate command, and then to request the current voltage:

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.1 # Sends ":VOLT 0.1" to set the voltage to 100 mV
>>> extreme.voltage # Sends ":VOLT?" to query for the current value
0.1
```

Finally, the [Instrument.setting](#) function can only set, but not read values.

Using the [Instrument.control](#), [Instrument.measurement](#), and [Instrument.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` function also provide means to achieve this.

### In a restricted range

If you have a property with a restricted range, you can use the `strict_range` and `truncated_range` functions.

For example, if our “Extreme 5000” can only support voltages from -1 V to 1 V, we can modify our previous example to use a strict validator over this range.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float strictly from -1 to 1).""",
    validator=strict_range,
    values=[-1, 1]
)
```

Now our voltage will raise a `ValueError` if the value is out of the range.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100
Traceback (most recent call last):
...
ValueError: Value of 100 is not in range [-1,1]
```

This is useful if you want to alert the programmer that they are using an invalid value. However, sometimes it can be nicer to truncate the value to be within the range.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float from -1 to 1).

    Invalid voltages are truncated.
    """,
    validator=truncated_range,
    values=[-1, 1]
)
```

Now our voltage will not raise an error, and will truncate the value to the range bounds.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100 # Executes ":VOLT 1"
>>> extreme.voltage
1.0
```

## In a discrete set

Often a control property should only take a few discrete values. You can use the `strict_discrete_set` and `truncated_discrete_set` functions to handle these situations. The strict version raises an error if the value is not in the set, as in the range examples above.

For example, if our “Extreme 5000” has a `:RANG <float>` command that sets the voltage range that can take values of 10 mV, 100 mV, and 1 V in Volts, then we can write a control as follows.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %g",
    """Control the voltage range in Volts (float in 10e-3, 100e-3, 1).""",
    validator=truncated_discrete_set,
    values=[10e-3, 100e-3, 1]
)
```

Now we can set the voltage range, which will automatically truncate to an appropriate value.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.08
>>> extreme.voltage
0.1
```

## 10.5.3 Mapping values

Now that you are familiar with the validators, you can additionally use maps to satisfy instruments which require non-physical values. The `map_values` argument of `Instrument.control` enables this feature.

If your set of values is a list, then the command will use the index of the list. For example, if our “Extreme 5000” instead has a `:RANG <integer>`, where 0, 1, and 2 correspond to 10 mV, 100 mV, and 1 V, then we can use the following control.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %d",
    """Control the voltage range in Volts (float in 10 mV, 100 mV and 1 V).
    """,
    validator=truncated_discrete_set,
    values=[10e-3, 100e-3, 1],
    map_values=True
)
```

Now the actual GPIB/SCIP command is “:RANG 1” for a value of 100 mV, since the index of 100 mV in the values list is 1.

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100e-3
>>> extreme.read()
'1'
>>> extreme.voltage = 1
>>> extreme.voltage
1
```

Dictionaries provide a more flexible method for mapping between real-values and those required by the instrument. If instead the `:RANG <integer>` took 1, 2, and 3 to correspond to 10 mV, 100 mV, and 1 V, then we can replace our previous control with the following.

```
Extreme5000.voltage = Instrument.control(
    ":RANG?", ":RANG %d",
    """Control the voltage range in Volts (float in 10 mV, 100 mV and 1 V).
    """,
    validator=truncated_discrete_set,
    values={10e-3:1, 100e-3:2, 1:3},
    map_values=True
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 10e-3
>>> extreme.read()
'1'
>>> extreme.voltage = 100e-3
>>> extreme.voltage
0.1
```

The dictionary now maps the keys to specific values. The values and keys can be any type, so this can support properties that use strings:

```
Extreme5000.channel = Instrument.control(
    ":CHAN?", ":CHAN %d",
    """Control the measurement channel (string strictly in 'X', 'Y', 'Z').""",
    validator=strict_discrete_set,
    values={'X':1, 'Y':2, 'Z':3},
    map_values=True
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.channel = 'X'
>>> extreme.read()
'1'
>>> extreme.channel = 'Y'
>>> extreme.channel
'Y'
```

As you have seen, the `Instrument.control` function can be significantly extended by using validators and maps.

### 10.5.4 Boolean properties

The idea of using maps can be leveraged to implement properties where the user-facing values are booleans, so you can interact in a pythonic way using `True` and `False`:

```
Extreme5000.output_enabled = Instrument.control(
    "OUTP?", "OUTP %d",
    """Control the instrument output is enabled (boolean).""",
    validator=strict_discrete_set,
    map_values=True,
    values={True: 1, False: 0}, # the dict values could also be "on" and "off", etc.
    ↪ depending on the device
)
```

```

>>> extreme = Extreme5000("GPIB::1")
>>> extreme.output_enabled = True
>>> extreme.read()
'1'
>>> extreme.output_enabled = False
>>> extreme.output_enabled
False
>>> # Invalid input raises an exception
>>> extreme.output_enabled = 34
Traceback (most recent call last):
...
ValueError: Value of 34 is not in the discrete set {True: 1, False: 0}

```

Good names for boolean properties are chosen such that they could also be a yes/no question: “Is the output enabled?”  
 -> output\_enabled, display\_active, etc.

### 10.5.5 Processing of set values

The `Instrument.control`, and `Instrument.setting` allow a keyword argument `set_process` which must be a function that takes a value after validation and performs processing before value mapping. This function must return the processed value. This can be typically used for unit conversions as in the following example:

```

Extreme5000.current = Instrument.setting(
    ":CURR %g",
    """Set the measurement current in A (float strictly from 0 to 10).""",
    validator=strict_range,
    values=[0, 10],
    set_process=lambda v: 1e3*v, # convert current to mA
)

```

```

>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 1 # set current to 1000 mA

```

### 10.5.6 Processing of return values

Similar to `set_process` the `Instrument.control`, and `Instrument.measurement` functions allow a `get_process` argument which if specified must be a function that takes a value and performs processing before value mapping. The function must return the processed value. In analogy to the example above this can be used for example for unit conversion:

```

Extreme5000.current = Instrument.control(
    ":CURR?", ":CURR %g",
    """Control the measurement current in A (float strictly from 0 to 10).""",
    validator=strict_range,
    values=[0, 10],
    set_process=lambda v: 1e3*v, # convert to mA
    get_process=lambda v: 1e-3*v, # convert to A
)

```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 3.1
>>> extreme.current
3.1
```

Another use-case of *set-process*, *get-process* is conversion from/to a `pint.Quantity`. Modifying above example to set or return a quantity, we get:

```
from pymeasure.units import ureg

Extreme5000.current = Instrument.control(
    ":CURR?", ":CURR %g",
    """Control the measurement current (float).""",
    set_process=lambda v: v.m_as(ureg.mA), # send the value as mA to the device
    get_process=lambda v: ureg.Quantity(v, ureg.mA), # convert to quantity
)
```

```
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.current = 3.1 * ureg.A
>>> extreme.current.m_as(ureg.A)
3.1
```

---

**Note:** This is, how quantities can be used in pymeasure instruments right now. [Issue 666](#) develops a more convenient implementation of quantities in the property factories.

---

*get\_process* can also be used to perform string processing. Let's say your instrument returns a value with its unit (e.g. 1.23 nF), which has to be removed. This could be achieved by the following code:

```
Extreme5000.capacity = Instrument.measurement(
    ":CAP?",
    """Measure the capacity in nF (float).""",
    get_process=lambda v: float(v.replace('nF', ''))
)
```

The same can be also achieved by the *preprocess\_reply* keyword argument to *Instrument.control* or *Instrument.measurement*. This function is forwarded to *Adapter.values* and runs directly after receiving the reply from the device. One can therefore take advantage of the built in casting abilities and simplify the code accordingly:

```
Extreme5000.capacity = Instrument.measurement(
    ":CAP?",
    """Measure the capacity in nF (float).""",
    preprocess_reply=lambda v: v.replace('nF', '')
    # notice how we don't need to cast to float anymore
)
```

### 10.5.7 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 the commands for this are `:VOLT:FREQ?` and `:VOLT:FREQ <float>,<float>`, we could use the following property:

```
Extreme5000.combination = Instrument.control(
    ":VOLT:FREQ?", ":VOLT:FREQ %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 ":VOLT:FREQ 0.2,931"
>>> extreme.combination                       # Reads ":VOLT:FREQ?"
[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 accomodating some interesting use cases with a very compact syntax.

Dynamic features of a property are enabled by setting its `dynamic` parameter to `True`.

Afterwards, creating specifically-named attributes (either in class definitions or on instances) allows modifying the parameters used at the time of property definition. You need to define an attribute whose name is `<property name>_<property parameter>` and assign to it the desired value. Pay attention *not* to inadvertently define other class attribute or instance attribute names matching this pattern, since they could unintentionally modify the property behaviour.

---

**Note:** To clearly distinguish these special attributes from normal class/instance attributes, they can only be set, not read.

---

The mechanism works for all the parameters in properties, except dynamic and docs – see [Instrument.control](#), [Instrument.measurement](#), [Instrument.setting](#).

### Dynamic validity range

Let's assume we have an instrument with a command that accepts a different valid range of values depending on its current state. The code below shows how this can be accomplished with dynamic properties.

```
Extreme5000.voltage = Instrument.control(
    ":VOLT?", ":VOLT %g",
    """Control the voltage in Volts (float).""",
    validator=strict_range,
    values=[-1, 1],
    dynamic = True,
)
def set_bipolar_mode(self, enabled = True):
    """Safely switch between bipolar/unipolar mode."""

    # some code to switch off the output first
    # ...

    if enabled:
        self.mode = "BIPOLAR"
        # set valid range of "voltage" property
        self.voltage_values = [-1, 1]
    else:
        self.mode = "UNIPOLAR"
        # note the "propertyname_parametername" form of the attribute
        self.voltage_values = [0, 1]
```

Now our voltage property has a dynamic validity range, either [-1, 1] or [0, 1]. A side effect of this is that the property's docstring should be less specific, to avoid it containing dynamically changed information (like the admissible value range). In this example, the property name was voltage and the parameter to adjust was values, so we used `self.voltage_values` to set our desired values.

## 10.6 Instruments with similar features

When instruments have a similar set of features, it makes sense to use inheritance to obtain most of the functionality from a parent instrument class, instead of copy-pasting code.

---

**Note:** Don't forget to update the instrument's name attribute accordingly, by either supplying an appropriate argument (if available) during the `super().__init__()` call, or by setting it anew below that call.

---

In some cases, one only needs to add additional properties and methods. In other cases, some of the already present properties/methods need to be completely replaced by defining them again in the derived class. Often, however, only some details need to be changed. This can be dealt with efficiently using dynamic properties.

### 10.6.1 Instrument family with different parameter values

A common case is to have a family of similar instruments with some parameter range different for each family member. In this case you would update the specific class parameter range without rewriting the entire property:

```
class FictionalInstrumentFamily(Instrument):
    frequency = Instrument.setting(
        "FREQ %g",
        """Set the frequency (float).""",
        validator=strict_range,
        values=[0, 1e9],
        dynamic=True,
        # ... other possible parameters follow
    )
    #
    # ... complete class implementation here
    #

class FictionalInstrument_1GHz(FictionalInstrumentFamily):
    pass

class FictionalInstrument_3GHz(FictionalInstrumentFamily):
    frequency_values = [0, 3e9]

class FictionalInstrument_9GHz(FictionalInstrumentFamily):
    frequency_values = [0, 9e9]
```

Notice how easily you can derive the different family members from a common class, and the fact that the attribute is now defined at class level and not at instance level.

### 10.6.2 Instruments with similar command syntax

Another use case involves maintaining compatibility between instruments with commands having different syntax, like in the following example.

```
class MultimeterA(Instrument):
    voltage = Instrument.measurement(get_command="VOLT?", ...)

    # ...full class definition code here

class MultimeterB(MultimeterA):
    # Same as brand A multimeter, but the command to read voltage
    # is slightly different
    voltage_get_command = "VOLTAGE?"
```

In the above example, MultimeterA and MultimeterB use a different command to read the voltage, but the rest of the behaviour is identical. MultimeterB can be defined subclassing MultimeterA and just implementing the difference.

## 10.7 Instruments with channels

Some instruments, like oscilloscopes and voltage sources, have channels whose commands differ only in the channel name. For this case, we have *Channel*, which is similar to *Instrument* and its property factories, but does expect an *Instrument* instance (i.e., a parent instrument) instead of an *Adapter* as parameter. All the channel communication is routed through the instrument's methods (*write*, *read*, etc.). However, *Channel.insert\_id* uses `str.format` to insert the channel's id at any 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.
↳R00111122233334444'
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'\x0201010K\x03')],
    ):
        pass # Verify the expected communication.

def test_information_getter():
    with expected_protocol(
        TC038,
        [(b'\x0201010WRS01D0002\x03', b'\x0201010K\x03'),
         (b'\x0201010INF6\x03', b'\x0201010KUT150333 V01.R00111122233334444\x03')],
    ) as inst:
        assert inst.information == 'UT150333 V01.R00111122233334444'

@pytest.mark.parametrize("comm_pairs, value", (
    [(b'\x0201010WRS01D0002\x03', b'\x0201010K\x03'),
     (b'\x0201010WWRD0120,01,00C8\x03', b'\x0201010K\x03')],
    20),
    [(b'\x0201010WRS01D0002\x03', b'\x0201010K\x03'),
     (b'\x0201010WWRD0120,01,0258\x03', b'\x0201010K\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(
        TC038,
        [(b'\x0201010WRS01D0002\x03', b'\x0201010K\x03'),

```

(continues on next page)

(continued from previous page)

```
(b'\x0201010WRDD0120,01\x03', b'\x0201010K00C8\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 TCP/IP::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](#)



## CODING STANDARDS

In order to maintain consistency across the different instruments in the PyMeasure repository, we enforce the following standards.

### 11.1 Python style guides

The [PEP8 style guide](#) and [PEP257 docstring conventions](#) should be followed.

Function and variable names should be lower case with underscores as needed to separate words. CamelCase should only be used for class names, unless working with Qt, where its use is common.

In addition, there is a configuration for the [flake8](#) linter present. Our codebase should not trigger any warnings. Many editors/IDEs can run this tool in the background while you work, showing results inline. Alternatively, you can run `flake8` in the repository root to check for problems. In addition, our automation on Github also runs some checkers. As this results in a much slower feedback loop for you, it's not recommended to rely only on this.

It is allowed but not required to use the [black](#) code formatter. To avoid introducing unrelated changes when working on an existing file, it is recommended to use the [darker](#) tool instead of `black`. This helps to keep the focus on the implementation instead of unrelated formatting, and thereby facilitates code reviews. `darker` is compatible with `black`, but only formats regions that show as changed in Git. If there are conflicts between `black/darker`'s output and `flake8` (especially related to [E203](#)), `flake8` takes precedence. Use `#noqa : E203` to disable E203 warnings for a specific line if appropriate.

There are no plans to support type hinting in PyMeasure code. This adds a lot of additional code to manage, without a clear advantage for this project. Type documentation should be placed in the docstring where not clear from the variable name.

### 11.2 Documentation

PyMeasure documents code using reStructuredText and the [Sphinx documentation generator](#). All functions, classes, and methods should be documented in the code using a docstring, see section [Docstrings](#).

## 11.3 Usage of getter and setter functions

Getter and setter functions are discouraged, since properties provide a more fluid experience. Given the extensive tools available for defining properties, detailed in the sections starting with *Writing properties*, these types of properties are 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”), settable (“Set”), or both (“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.

## 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  
Max Herbold  
Alexander Wichers  
Ashok Bruno  
Robert Roos  
Sebastien Weber  
Sebastian Neusch  
Ulrich Sauter

**LICENSE**

Copyright (c) 2013-2023 PyMeasure Developers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## CHANGELOG

### 14.1 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).

### 14.2 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.

#### 14.2.1 Deprecated features

- Attocube ANC300: The `stepu` and `stepd` properties are deprecated, use the new `move_raw` method instead. (@dkriegner, #938)

#### 14.2.2 Instruments

- Adds a test generator (@bmoneke, #882)
- Adds Thyracont Smartline v2 vacuum sensor transmitter (@bmoneke, #940)
- Adds Thyracont Smartline v1 vacuum gauge (@dkriegner, #937)
- Adds Teledyne 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)

### 14.2.3 GUI

- Adds console mode (@msmttchr, #500)
- Fixes Dock widget (@msmttchr, #961)

### 14.2.4 Miscellaneous

- Change CI from conda to mamba (@bmoneke, #947)
- Add support for python 3.11 (@CasperSchippers, #896)

### 14.2.5 New Contributors

@waveman68, @omahs, @ileu

**Full Changelog:** <https://github.com/pymasure/pymasure/compare/v0.12.0...v0.13.0>

## 14.3 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.

### 14.3.1 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.

### 14.3.2 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 `gpi_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)

### 14.3.3 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)

### 14.3.4 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)

### 14.3.5 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)

### 14.3.6 Miscellaneous

- Adds a collection of solutions for instrument implementation challenges (@bmoneke, #853, #861)
- Updates Tutorials/Making\_a\_measurement/ example\_codes (@sansanda, #749)

### 14.3.7 New Contributors

@JCarl-OS, @aruzniето, @scandey, @tomverbeure, @wichers, @Max-Herbold, @RobertoRoos

**Full Changelog:** <https://github.com/pymasure/pymasure/compare/v0.11.1...v0.12.0>

## 14.4 Version 0.11.1 (2022-12-31)

### 14.4.1 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/pymasure/pymasure/compare/v0.11.0...v0.11.1>

### 14.4.2 Dependency updates

- Required version of `PyQtGraph` is increased from `pyqtgraph >= 0.9.10` to `pyqtgraph >= 0.12` to support new PyMeasure display widgets.

### 14.4.3 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)

## 14.5 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.

### 14.5.1 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.

### 14.5.2 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.

### 14.5.3 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)

### 14.5.4 Automation

- Add storing unitful experiment results (@bmoneke, #642)
- Add storing conditions in file (@CasperSchippers, #503)

### 14.5.5 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)

### 14.5.6 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.

### 14.5.7 New Contributors

@jcarbelbide, @mroeleke, @bmoneke, @garzetti, @AsafYagoda, @AidenDawn, @LastStarDust, @sansanda

**Full Changelog:** <https://github.com/pymasure/pymasure/compare/v0.10.0...v0.11.0>

## 14.6 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

### 14.6.1 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 (@bicarsen)
- Fix Attocube ANC300 with new SCPI Instrument properties (@dkriegner)
- Fix bug in wait\_for\_trigger of Agilent33220A (neal-kepler)

## 14.6.2 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)

## 14.6.3 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 (@mcd0486)
- 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 (@mcd0486)
- Change copyright date to 2022 (@LongnoseRob)
- Removed unused code (@msmttchr)

#### 14.6.4 New Contributors

@LongnoseRob, @neal, @hududed, @cornea, @Daivesd, @samcondon4, @maederan201, @bleykauf, @mfikes, @bi-carlsen, @md12g12, @CodingMarco, @jarvas24, @mcd0486!

**Full Changelog:** <https://github.com/pymasure/pymasure/compare/v0.9...v0.10.0>

### 14.7 Version 0.9 – released 2/7/21

- PyMeasure is now officially at [github.com/pymasure/pymasure](https://github.com/pymasure/pymasure)
- 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 compatibility (@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)
- Improvements 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)

## 14.8 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)

## 14.9 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)

## 14.10 Version 0.6.1 – released 4/21/19

- Added Elektronika 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)

## 14.11 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 instruments (@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)

## 14.12 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)

## 14.13 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)

## 14.14 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

## 14.15 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)

## 14.16 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)

## 14.17 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

## 14.18 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 Keithley 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

## 14.19 Version 0.4.1 – released 7/31/16

- Critical fix in setup.py for importing instruments (also added to documentation)

## 14.20 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

## 14.21 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

## 14.22 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

## 14.23 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

## 14.24 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

## 14.25 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

## 14.26 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

## 14.27 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

## 14.28 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

## 14.29 Version 0.1.0 – released 7/15/14

- Initial release



## PYTHON MODULE INDEX

### p

`pymeasure.display.browser`, 83  
`pymeasure.display.console`, 83  
`pymeasure.display.curves`, 84  
`pymeasure.display.inputs`, 85  
`pymeasure.display.listeners`, 87  
`pymeasure.display.log`, 87  
`pymeasure.display.manager`, 87  
`pymeasure.display.plotter`, 89  
`pymeasure.display.thread`, 89  
`pymeasure.display.widgets.browser_widget`, 90  
`pymeasure.display.widgets.directory_widget`, 90  
`pymeasure.display.widgets.dock_widget`, 95  
`pymeasure.display.widgets.estimator_widget`, 90  
`pymeasure.display.widgets.image_frame`, 90  
`pymeasure.display.widgets.image_widget`, 90  
`pymeasure.display.widgets.inputs_widget`, 91  
`pymeasure.display.widgets.log_widget`, 91  
`pymeasure.display.widgets.plot_frame`, 91  
`pymeasure.display.widgets.plot_widget`, 92  
`pymeasure.display.widgets.results_dialog`, 92  
`pymeasure.display.widgets.sequencer_widget`, 92  
`pymeasure.display.widgets.tab_widget`, 94  
`pymeasure.display.widgets.table_widget`, 95  
`pymeasure.display.windows.managed_dock_window`, 101  
`pymeasure.display.windows.managed_image_window`, 98  
`pymeasure.display.windows.managed_window`, 98  
`pymeasure.display.windows.plotter_window`, 100  
`pymeasure.experiment.experiment`, 71  
`pymeasure.experiment.listeners`, 72  
`pymeasure.experiment.parameters`, 74  
`pymeasure.experiment.procedure`, 73  
`pymeasure.experiment.results`, 79  
`pymeasure.experiment.workers`, 79  
`pymeasure.instruments`, 101  
`pymeasure.instruments.activetechnologies`, 116  
`pymeasure.instruments.advantest`, 121  
`pymeasure.instruments.advantest.advantestR3767CG`, 122  
`pymeasure.instruments.advantest.advantestR624X`, 143  
`pymeasure.instruments.agilent`, 152  
`pymeasure.instruments.agilent.agilent4156`, 163  
`pymeasure.instruments.agilent.agilentB1500`, 193  
`pymeasure.instruments.aja`, 196  
`pymeasure.instruments.ametek`, 198  
`pymeasure.instruments.ami`, 200  
`pymeasure.instruments.anaheimautomation`, 202  
`pymeasure.instruments.anapico`, 204  
`pymeasure.instruments.andeenhagerling`, 205  
`pymeasure.instruments.anritsu`, 208  
`pymeasure.instruments.attocube`, 223  
`pymeasure.instruments.bkprecision`, 226  
`pymeasure.instruments.comedi`, 116  
`pymeasure.instruments.danfysik`, 226  
`pymeasure.instruments.deltaelektronika`, 229  
`pymeasure.instruments.edwards`, 231  
`pymeasure.instruments.eurotest`, 231  
`pymeasure.instruments.fluke`, 233  
`pymeasure.instruments.fwbell`, 234  
`pymeasure.instruments.hcp`, 237  
`pymeasure.instruments.heidenhain`, 237  
`pymeasure.instruments.hp`, 239  
`pymeasure.instruments.ipgphotonics`, 286  
`pymeasure.instruments.keithley`, 287  
`pymeasure.instruments.keysight`, 338  
`pymeasure.instruments.lakeshore`, 351  
`pymeasure.instruments.lecroy`, 360  
`pymeasure.instruments.mksinst`, 375  
`pymeasure.instruments.newport`, 376  
`pymeasure.instruments.ni`, 378  
`pymeasure.instruments.novanta`, 392  
`pymeasure.instruments.oxfordinstruments`, 393  
`pymeasure.instruments.parker`, 403  
`pymeasure.instruments.pendulum`, 404  
`pymeasure.instruments.razorbill`, 405  
`pymeasure.instruments.rohdeschwarz`, 406

`pymeasure.instruments.siglenttechnologies`,  
423  
`pymeasure.instruments.signalrecovery`, 427  
`pymeasure.instruments.srs`, 440  
`pymeasure.instruments.tcpowerconversion`, 452  
`pymeasure.instruments.tdk`, 456  
`pymeasure.instruments.tektronix`, 464  
`pymeasure.instruments.teledyne`, 465  
`pymeasure.instruments.temptronic`, 476  
`pymeasure.instruments.texio`, 484  
`pymeasure.instruments.thermotron`, 487  
`pymeasure.instruments.thorlabs`, 488  
`pymeasure.instruments.thyracont`, 489  
`pymeasure.instruments.toptica`, 494  
`pymeasure.instruments.validators`, 113  
`pymeasure.instruments.velleman`, 497  
`pymeasure.instruments.yokogawa`, 499  
`pymeasure.test`, 66

## Symbols

- `__call__()` (*pymeasure.instruments.agilent.agilentB1500.Ranging* method), 192
- `__str__()` (*pymeasure.instruments.agilent.agilentB1500.CustomUnitEnum* method), 193
- `_format_binary_values()` (*pymeasure.adapters.PrologixAdapter* method), 57
- `_format_binary_values()` (*pymeasure.adapters.SerialAdapter* method), 53
- A**
- A** (*pymeasure.instruments.hp.hp856Xx.Trace* attribute), 277
- `abort()` (*pymeasure.display.console.ManagedConsole* method), 84
- `abort()` (*pymeasure.display.manager.BaseManager* method), 87
- `abort()` (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500* method), 184
- `abort()` (*pymeasure.instruments.anritsu.AnritsuMS2090A* method), 211
- `absolute_position` (*pymeasure.instruments.anaheimautomation.DPSeriesMotorController* property), 202
- `absolute_to_steps()` (*pymeasure.instruments.anaheimautomation.DPSeriesMotorController* method), 202
- AC** (*pymeasure.instruments.hp.hp856Xx.CouplingMode* attribute), 278
- `ac_current` (*pymeasure.instruments.agilent.AgilentE4980* property), 157
- `ac_mode()` (*pymeasure.instruments.lakeshore.LakeShore425* method), 358
- `ac_voltage` (*pymeasure.instruments.agilent.AgilentE4980* property), 157
- `acquire_digital_input_output()` (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 390
- `acquire_digital_multimeter()` (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 390
- `acquire_function_generator()` (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 390
- `acquire_mixed_signal_oscilloscope()` (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 390
- `acquire_power_supply()` (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 390
- `acquire_reference()` (*pymeasure.instruments.keithley.Keithley2000* method), 287
- `acquisition_average` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 362
- `acquisition_mode` (*pymeasure.instruments.keysight.KeysightDSOX1102G* property), 339
- `acquisition_sample_size()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 362
- `acquisition_sample_size_c1` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 362
- `acquisition_sample_size_c2` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 362
- `acquisition_sample_size_c3` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 362
- `acquisition_sample_size_c4` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 362
- `acquisition_sampling_rate` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 363
- `acquisition_status` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* property), 363
- `acquisition_type` (*pymeasure.instruments.keysight.KeysightDSOX1102G* property), 339

- `acquisition_type` (`pymea-  
sure.instruments.lecroy.LeCroyT3DSO1204  
property`), 363
- `activate()` (`pymea-  
sure.instruments.anritsu.AnritsuMS464xB.MeasureChannel  
method`), 220
- `activate()` (`pymea-  
sure.instruments.anritsu.AnritsuMS464xB.Trace  
method`), 223
- `activate_source_peak_tracking()` (`pymea-  
sure.instruments.hp.HP8560A  
method`), 273
- `active_channel` (`pymea-  
sure.instruments.anritsu.AnritsuMS464xB  
property`), 216
- `active_connectors` (`pymea-  
sure.instruments.hp.HP3478A  
property`), 245
- `active_gun` (`pymea-  
sure.instruments.aja.DCXS  
property`), 196
- `active_state` (`pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property`), 211
- `active_trace` (`pymea-  
sure.instruments.anritsu.AnritsuMS464xB.MeasureChannel  
property`), 220
- `activity` (`pymea-  
sure.instruments.oxfordinstruments.IPS1200  
property`), 399
- `Adapter` (class in `pymea-  
sure.adapters`), 47
- `adc1` (`pymea-  
sure.instruments.ametek.Ametek7270  
property`), 198
- `adc1` (`pymea-  
sure.instruments.signalrecovery.DSP7225  
property`), 427
- `adc1` (`pymea-  
sure.instruments.signalrecovery.DSP7265  
property`), 434
- `adc1` (`pymea-  
sure.instruments.srs.SR830  
property`), 442
- `adc1` (`pymea-  
sure.instruments.srs.SR860  
property`), 446
- `adc2` (`pymea-  
sure.instruments.ametek.Ametek7270  
property`), 198
- `adc2` (`pymea-  
sure.instruments.signalrecovery.DSP7225  
property`), 427
- `adc2` (`pymea-  
sure.instruments.signalrecovery.DSP7265  
property`), 434
- `adc2` (`pymea-  
sure.instruments.srs.SR830  
property`), 442
- `adc2` (`pymea-  
sure.instruments.srs.SR860  
property`), 446
- `adc3` (`pymea-  
sure.instruments.ametek.Ametek7270  
property`), 198
- `adc3` (`pymea-  
sure.instruments.signalrecovery.DSP7265  
property`), 434
- `adc3` (`pymea-  
sure.instruments.srs.SR830  
property`), 442
- `adc3` (`pymea-  
sure.instruments.srs.SR860  
property`), 446
- `adc3_time` (`pymea-  
sure.instruments.signalrecovery.DSP7265  
property`), 434
- `adc4` (`pymea-  
sure.instruments.ametek.Ametek7270  
property`), 198
- `adc4` (`pymea-  
sure.instruments.srs.SR830  
property`), 442
- `adc4` (`pymea-  
sure.instruments.srs.SR860  
property`), 446
- `adc_auto_zero` (`pymea-  
sure.instruments.agilent.agilentB1500.AgilentB1500  
property`), 186
- `add_averaging_channel()` (`pymea-  
sure.instruments.agilent.agilentB1500.AgilentB1500  
method`), 185
- `adc_setup()` (`pymea-  
sure.instruments.agilent.agilentB1500.AgilentB1500  
method`), 185
- `adc_type` (`pymea-  
sure.instruments.agilent.agilentB1500.SMU  
property`), 189
- `ADCMode` (class in `pymea-  
sure.instruments.agilent.agilentB1500`), 193
- `ADCType` (class in `pymea-  
sure.instruments.agilent.agilentB1500`), 193
- `add()` (`pymea-  
sure.display.browser.Browser  
method`), 83
- `add_child()` (`pymea-  
sure.instruments.common_base.CommonBase  
method`), 104
- `add_child()` (`pymea-  
sure.instruments.keithley.Keithley2200  
method`), 335
- `add_child()` (`pymea-  
sure.instruments.keysight.KeysightE36312A  
method`), 345
- `add_child()` (`pymea-  
sure.instruments.lecroy.LeCroyT3DSO1204  
method`), 363
- `add_node()` (`pymea-  
sure.display.widgets.sequencer_widget.SequencerTree  
method`), 93
- `add_ramp_step()` (`pymea-  
sure.instruments.danfysik.Danfysik8500  
method`), 227
- `address` (`pymea-  
sure.instruments.anaheimautomation.DPSeriesMotorCont  
property`), 202
- `address` (`pymea-  
sure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38  
property`), 456
- `address` (`pymea-  
sure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65  
property`), 461
- `adjust_all()` (`pymea-  
sure.instruments.hp.hp856Xx.HP856Xx  
method`), 256
- `adjust_if` (`pymea-  
sure.instruments.hp.hp856Xx.HP856Xx  
attribute`), 256
- `AdvantestR3767CG` (class in `pymea-  
sure.instruments.advantest.advantestR3767CG`), 122
- `AdvantestR6245` (class in `pymea-  
sure.instruments.advantest.advantestR624X`), 122
- `AdvantestR6246` (class in `pymea-  
sure.instruments.advantest.advantestR624X`), 122
- `AdvantestR624X` (class in `pymea-  
sure.instruments.advantest.advantestR624X`), 123
- `AFG3152C` (class in `pymea-  
sure.instruments.tektronix`), 465
- `Agilent33220A` (class in `pymea-`

*sure.instruments.agilent*), 170

**Agilent33500** (class in *pymeasure.instruments.agilent*), 172

**Agilent33500Channel** (class in *pymeasure.instruments.agilent.agilent33500*), 176

**Agilent33521A** (class in *pymeasure.instruments.agilent*), 176

**Agilent34410A** (class in *pymeasure.instruments.agilent*), 159

**Agilent34450A** (class in *pymeasure.instruments.agilent*), 159

**Agilent4156** (class in *pymeasure.instruments.agilent.agilent4156*), 163

**Agilent8257D** (class in *pymeasure.instruments.agilent*), 152

**Agilent8722ES** (class in *pymeasure.instruments.agilent*), 155

**AgilentB1500** (class in *pymeasure.instruments.agilent.agilentB1500*), 183

**AgilentE4408B** (class in *pymeasure.instruments.agilent*), 156

**AgilentE4980** (class in *pymeasure.instruments.agilent*), 157

**AH2500A** (class in *pymeasure.instruments.andeenhagerling*), 205

**AH2700A** (class in *pymeasure.instruments.andeenhagerling*), 206

**air\_temperature** (*pymeasure.instruments.temptronic.ATSB* property), 476

**alarm\_active** (*pymeasure.instruments.lakeshore.LakeShore421* property), 355

**alarm\_audible** (*pymeasure.instruments.lakeshore.LakeShore421* property), 355

**alarm\_high** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_high\_multiplier** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_high\_raw** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_in\_out** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_low** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_low\_multiplier** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_low\_raw** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_mode\_enabled** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**alarm\_sort\_enabled** (*pymeasure.instruments.lakeshore.LakeShore421* property), 356

**all\_pressures** (*pymeasure.instruments.mksinst.mks937b.MKS937B* property), 376

**am\_depth** (*pymeasure.instruments.hp.HP8657B* property), 281

**am\_source** (*pymeasure.instruments.hp.HP8657B* property), 281

**Ametek7270** (class in *pymeasure.instruments.ametek*), 198

**AMI430** (class in *pymeasure.instruments.ami*), 200

**amplitude** (*pymeasure.instruments.agilent.Agilent33220A* property), 170

**amplitude** (*pymeasure.instruments.agilent.Agilent33500* property), 173

**amplitude** (*pymeasure.instruments.agilent.agilent33500.Agilent33500Channel* property), 176

**amplitude** (*pymeasure.instruments.hp.HP33120A* property), 239

**amplitude** (*pymeasure.instruments.hp.HP8116A* property), 248

**Amplitude** (*pymeasure.instruments.hp.hp856Xx.DemodulationMode* attribute), 278

**amplitude** (*pymeasure.instruments.teledyne.teledyneT3AFG.SignalChannel* property), 467

**amplitude\_depth** (*pymeasure.instruments.agilent.Agilent8257D* property), 152

**amplitude\_source** (*pymeasure.instruments.agilent.Agilent8257D* property), 152

**amplitude\_unit** (*pymeasure.instruments.agilent.Agilent33220A* property), 170

**amplitude\_unit** (*pymeasure.instruments.agilent.Agilent33500* property), 173

**amplitude\_unit** (*pymeasure.instruments.agilent.agilent33500.Agilent33500Channel* property), 176

**amplitude\_unit** (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 251

**amplitude\_units** (*pymeasure.instruments.hp.HP33120A* property), 240

**AmplitudeUnits** (class in *pymeasure.instruments.hp.hp856Xx*), 277

analog\_configuration (pymeasure.instruments.lakeshore.LakeShore211 property), 351  
 analog\_input (pymeasure.instruments.advantest.advantestR624X.SMUChannel property), 132  
 analog\_out (pymeasure.instruments.lakeshore.LakeShore211 property), 352  
 analog\_output\_setting (pymeasure.instruments.thyracont.smartline\_v2.Smartline\_v2 property), 492  
 analysis (pymeasure.instruments.anritsu.AnritsuMS9710C property), 208  
 analysis\_result (pymeasure.instruments.anritsu.AnritsuMS9710C property), 209  
 analyzer\_mode (pymeasure.instruments.agilent.agilent4156.Agilent4156 property), 164  
 ANC300Controller (class in pymeasure.instruments.attocube.anc300), 223  
 angle (pymeasure.instruments.parker.ParkerGV6 property), 403  
 angle\_error (pymeasure.instruments.parker.ParkerGV6 property), 403  
 annotation\_enabled (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 256  
 AnritsuMG3692C (class in pymeasure.instruments.anritsu), 208  
 AnritsuMS2090A (class in pymeasure.instruments.anritsu), 211  
 AnritsuMS4642B (class in pymeasure.instruments.anritsu), 215  
 AnritsuMS4644B (class in pymeasure.instruments.anritsu), 215  
 AnritsuMS4645B (class in pymeasure.instruments.anritsu), 215  
 AnritsuMS4647B (class in pymeasure.instruments.anritsu), 215  
 AnritsuMS464xB (class in pymeasure.instruments.anritsu), 215  
 AnritsuMS9710C (class in pymeasure.instruments.anritsu), 208  
 AnritsuMS9740A (class in pymeasure.instruments.anritsu), 211  
 aperture() (pymeasure.instruments.agilent.AgilentE4980 method), 157  
 append() (pymeasure.display.curves.BufferCurve method), 84  
 application\_type (pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel property), 220  
 applied (pymeasure.instruments.keithley.Keithley2260B property), 295  
 applied (pymeasure.instruments.texio.TexioPSW360L30 property), 485  
 apply\_current() (pymeasure.instruments.keithley.Keithley2400 method), 300  
 apply\_current() (pymeasure.instruments.keithley.Keithley2450 method), 308  
 apply\_current() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 499  
 apply\_voltage() (pymeasure.instruments.keithley.Keithley2400 method), 300  
 apply\_voltage() (pymeasure.instruments.keithley.Keithley2450 method), 308  
 apply\_voltage() (pymeasure.instruments.keithley.Keithley6517B method), 326  
 apply\_voltage() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 499  
 APSIN12G (class in pymeasure.instruments.anapico), 205  
 arb\_advance (pymeasure.instruments.agilent.Agilent33500 property), 173  
 arb\_advance (pymeasure.instruments.agilent.agilent33500.Agilent33500C property), 176  
 arb\_file (pymeasure.instruments.agilent.Agilent33500 property), 173  
 arb\_file (pymeasure.instruments.agilent.agilent33500.Agilent33500Chan property), 176  
 arb\_filter (pymeasure.instruments.agilent.Agilent33500 property), 173  
 arb\_filter (pymeasure.instruments.agilent.agilent33500.Agilent33500Chan property), 176  
 arb\_srate (pymeasure.instruments.agilent.Agilent33500 property), 173  
 arb\_srate (pymeasure.instruments.agilent.agilent33500.Agilent33500Chan property), 176  
 arb\_srate (pymeasure.instruments.agilent.Agilent33521A property), 176  
 ask() (pymeasure.adapters.Adapter method), 47  
 ask() (pymeasure.adapters.FakeAdapter method), 67  
 ask() (pymeasure.adapters.PrologixAdapter method), 57  
 ask() (pymeasure.adapters.SerialAdapter method), 53  
 ask() (pymeasure.adapters.TelnetAdapter method), 64  
 ask() (pymeasure.adapters.VISAAdapter method), 50  
 ask() (pymeasure.adapters.VXIIAdapter method), 61  
 ask() (pymeasure.instruments.agilent.agilentB1500.SMU method), 188  
 ask() (pymeasure.instruments.aja.DCXS method), 196  
 ask() (pymeasure.instruments.ametek.Ametek7270 method), 188

method), 198

ask() (pymeasure.instruments.common\_base.CommonBase method), 105

ask() (pymeasure.instruments.eurotest.EurotestHPP120256 method), 232

ask() (pymeasure.instruments.hp.HP8116A method), 248

ask() (pymeasure.instruments.keysight.KeysightE36312A method), 345

ask() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 363

ask() (pymeasure.instruments.oxfordinstruments.base.OxfordInstrumentsBase method), 394

ask() (pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 method), 492

ask\_manually() (pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 method), 492

ask\_raw() (pymeasure.adapters.VXII1Adapter method), 61

ask\_values() (pymeasure.adapters.PrologixAdapter method), 57

ask\_values() (pymeasure.adapters.VISAAAdapter method), 50

at\_temperature() (pymeasure.instruments.temptronic.ATSBBase method), 476

ATSS25 (class in pymeasure.instruments.temptronic), 483

ATSS45 (class in pymeasure.instruments.temptronic), 484

ATSBBase (class in pymeasure.instruments.temptronic), 476

attenuation (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 251

attenuation (pymeasure.instruments.rohdeschwarz.fsl.FSL attribute), 420

auto (pymeasure.adapters.PrologixAdapter property), 57

AUTO (pymeasure.instruments.agilent.agilentB1500.ADCMode attribute), 193

AUTO (pymeasure.instruments.agilent.agilentB1500.AutoManual attribute), 194

AUTO (pymeasure.instruments.agilent.agilentB1500.CompliancePolarity attribute), 196

AUTO (pymeasure.instruments.hp.hp856Xx.AmplitudeUnits attribute), 277

auto\_calibration (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 property), 184

auto\_gain (pymeasure.instruments.signalrecovery.DSP7225 property), 427

auto\_gain (pymeasure.instruments.signalrecovery.DSP7265 property), 434

auto\_input\_impedance\_enabled (pymeasure.instruments.hp.HP34401A property), 240

auto\_offset() (pymeasure.instruments.srs.SR830 method), 442

auto\_output\_off (pymeasure.instruments.keithley.Keithley2400 property), 300

auto\_phase() (pymeasure.instruments.signalrecovery.DSP7225 method), 427

auto\_phase() (pymeasure.instruments.signalrecovery.DSP7265 method), 434

auto\_range (pymeasure.instruments.oxfordinstruments.ITC503 property), 395

auto\_range (pymeasure.instruments.oxfordinstruments.ITC503 property), 395

auto\_range (pymeasure.instruments.lakeshore.LakeShore421 property), 356

auto\_range() (pymeasure.instruments.fwbell.FWBBell5080 method), 235

auto\_range() (pymeasure.instruments.keithley.Keithley2000 method), 288

auto\_range() (pymeasure.instruments.lakeshore.LakeShore425 method), 358

auto\_range\_enabled (pymeasure.instruments.hp.HP3478A property), 245

auto\_range\_source() (pymeasure.instruments.keithley.Keithley2400 method), 300

auto\_range\_source() (pymeasure.instruments.keithley.Keithley2450 method), 308

auto\_range\_source() (pymeasure.instruments.keithley.Keithley6517B method), 326

auto\_restart\_enabled (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property), 456

auto\_restart\_enabled (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property), 461

auto\_sensitivity() (pymeasure.instruments.signalrecovery.DSP7225 method), 428

auto\_sensitivity() (pymeasure.instruments.signalrecovery.DSP7265 method), 434

auto\_setup() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 385

`auto_zero` (`pymeasure.instruments.keithley.Keithley2400` property), 300

`auto_zero_enabled` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 141

`auto_zero_enabled` (`pymeasure.instruments.hp.HP3478A` property), 245

`AutoManual` (class in `pymeasure.instruments.agilent.agilentB1500`), 194

`autorange` (`pymeasure.instruments.hp.HP34401A` property), 241

`autoscale()` (`pymeasure.instruments.keysight.KeysightDSOX1102G` method), 339

`autoscale()` (`pymeasure.instruments.lecroy.LeCroyT3DSO4120A` method), 363

`autoscale()` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` method), 468

`autovernier_enabled` (`pymeasure.instruments.hp.HP8116A` property), 248

`autozero_enabled` (`pymeasure.instruments.hp.HP34401A` property), 241

`aux_in_1` (`pymeasure.instruments.srs.SR830` property), 442

`aux_in_1` (`pymeasure.instruments.srs.SR860` property), 446

`aux_in_2` (`pymeasure.instruments.srs.SR830` property), 442

`aux_in_2` (`pymeasure.instruments.srs.SR860` property), 446

`aux_in_3` (`pymeasure.instruments.srs.SR830` property), 442

`aux_in_3` (`pymeasure.instruments.srs.SR860` property), 446

`aux_in_4` (`pymeasure.instruments.srs.SR830` property), 443

`aux_in_4` (`pymeasure.instruments.srs.SR860` property), 446

`aux_out_1` (`pymeasure.instruments.srs.SR830` property), 443

`aux_out_1` (`pymeasure.instruments.srs.SR860` property), 446

`aux_out_2` (`pymeasure.instruments.srs.SR830` property), 443

`aux_out_2` (`pymeasure.instruments.srs.SR860` property), 446

`aux_out_3` (`pymeasure.instruments.srs.SR830` property), 443

`aux_out_3` (`pymeasure.instruments.srs.SR860` property), 446

`aux_out_4` (`pymeasure.instruments.srs.SR830` property), 443

`aux_out_4` (`pymeasure.instruments.srs.SR860` property), 447

`auxiliary_condition_code` (`pymeasure.instruments.temptronic.ATSB` property), 476

`average_count` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 220

`average_point` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209

`average_sweep` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209

`average_sweep` (`pymeasure.instruments.anritsu.AnritsuMS9740A` property), 211

`average_sweep_count` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 220

`average_type` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 220

`averages` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155

`averaging_enabled` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155

`averaging_enabled` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 221

`AWG401x_AFG` (class in `pymeasure.instruments.activetechnologies`), 116

`AWG401x_AWG` (class in `pymeasure.instruments.activetechnologies`), 117

`AWG401x_AWG.DummyEntriesElements` (class in `pymeasure.instruments.activetechnologies`), 117

`AWG401x_AWG.WaveformsLazyDict` (class in `pymeasure.instruments.activetechnologies`), 118

`axes` (`pymeasure.instruments.newport.ESP300` property), 377

`Axis` (class in `pymeasure.instruments.attocube.anc300`), 224

`Axis` (class in `pymeasure.instruments.newport.esp300`), 377

`AxisError` (class in `pymeasure.instruments.newport.esp300`), 378

## B

`B` (`pymeasure.instruments.hp.hp856Xx.Trace` attribute), 277

`bandwidth` (`pymeasure.instruments.anritsu.anritsuMS464xB.Measurement` property), 221

bandwidth\_enhancer\_enabled (pymeasure.instruments.anritsu.AnritsuMS464xB property), 216

BASE (pymeasure.instruments.agilent.agilentB1500.SamplingPostOutput property), 170

BaseBrowserItem (class in pymeasure.display.browser), 83

baseline\_offset (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG method), 120

baseline\_offset\_max (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG method), 120

baseline\_offset\_min (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG method), 120

BaseManager (class in pymeasure.display.manager), 87

basespeed (pymeasure.instruments.anaheimautomation.DPSeriesMotorController property), 202

basic\_info (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 408

batch\_size (pymeasure.instruments.pendulum.cnt91.CNT91 property), 404

baud\_rate (pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 method), 492

beep() (pymeasure.instruments.agilent.Agilent33220A method), 170

beep() (pymeasure.instruments.agilent.Agilent33500 method), 173

beep() (pymeasure.instruments.agilent.Agilent34450A method), 159

beep() (pymeasure.instruments.hp.HP33120A method), 240

beep() (pymeasure.instruments.hp.HP34401A method), 241

beep() (pymeasure.instruments.keithley.Keithley2000 method), 288

beep() (pymeasure.instruments.keithley.Keithley2400 method), 300

beep() (pymeasure.instruments.keithley.Keithley2450 method), 308

beep() (pymeasure.instruments.keithley.Keithley2700 method), 315

beep() (pymeasure.instruments.keithley.Keithley6221 method), 319

beep() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 421

beep\_state (pymeasure.instruments.keithley.Keithley2000 property), 288

beeper\_enabled (pymeasure.instruments.hp.HP34401A property), 241

beeper\_enabled (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 408

beeper\_state (pymeasure.instruments.agilent.Agilent33220A property), 170

BIAS (pymeasure.instruments.agilent.agilentB1500.SamplingPostOutput property), 195

bias\_enabled (pymeasure.instruments.srs.SR570 property), 441

bias\_level (pymeasure.instruments.srs.SR570 property), 441

binary\_data\_byte\_order (pymeasure.instruments.anritsu.AnritsuMS464xB property), 216

binary\_values() (pymeasure.adapters.Adapter method), 47

binary\_values() (pymeasure.adapters.FakeAdapter method), 67

BinaryMotorController (pymeasure.adapters.PrologixAdapter method), 57

binary\_values() (pymeasure.adapters.SerialAdapter method), 53

binary\_values() (pymeasure.adapters.TelnetAdapter method), 64

binary\_values() (pymeasure.adapters.VISAAdapter method), 50

binary\_values() (pymeasure.adapters.VXI11Adapter method), 61

binary\_values() (pymeasure.instruments.common\_base.CommonBase method), 105

binary\_values() (pymeasure.instruments.keithley.Keithley2200 method), 336

binary\_values() (pymeasure.instruments.keysight.KeysightE36312A method), 346

binary\_values() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 364

BKPrecision9130B (class in pymeasure.instruments.bkprecision), 226

blank\_front() (pymeasure.instruments.srs.SR570 method), 441

blank\_trace() (pymeasure.instruments.hp.hp856Xx.HP856Xx method), 265

blanking (pymeasure.instruments.anapico.APSIN12G property), 205

BooleanInput (class in pymeasure.display.inputs), 85

BooleanParameter (class in pymeasure.experiment.parameters), 74

bootloader\_version (pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 property), 205

*property*), 492

`both_channels_enabled` (*pymea-*  
*sure.instruments.keithley.Keithley2306 prop-*  
*erty*), 297

`Browser` (class in *pymea-*  
*sure.display.browser*), 83

`BrowserItem` (class in *pymea-*  
*sure.display.browser*), 83

`BrowserWidget` (class in *pymea-*  
*sure.display.widgets.browser\_widget*), 90

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley2000*  
*property*), 288

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley2400*  
*property*), 301

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley2450*  
*property*), 309

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley2700*  
*property*), 315

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley6221*  
*property*), 320

`buffer_data` (*pymea-*  
*sure.instruments.keithley.Keithley6517B*  
*property*), 326

`buffer_frequency_time_series()` (*pymea-*  
*sure.instruments.pendulum.cnt91.CNT91*  
*method*), 404

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley2000 prop-*  
*erty*), 288

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley2400 prop-*  
*erty*), 301

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley2450 prop-*  
*erty*), 309

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley2700 prop-*  
*erty*), 315

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley6221 prop-*  
*erty*), 320

`buffer_points` (*pymea-*  
*sure.instruments.keithley.Keithley6517B*  
*property*), 326

`buffer_to_float()` (*pymea-*  
*sure.instruments.signalrecovery.DSP7225*  
*method*), 428

`buffer_to_float()` (*pymea-*  
*sure.instruments.signalrecovery.DSP7265*  
*method*), 434

`BufferCurve` (class in *pymea-*  
*sure.display.curves*), 84

`burst_count` (*pymea-*  
*sure.instruments.activetechnologies.AWG401x\_AWG*  
*property*), 118

`burst_count_max` (*pymea-*  
*sure.instruments.activetechnologies.AWG401x\_AWG*  
*property*), 118

`burst_count_min` (*pymea-*  
*sure.instruments.activetechnologies.AWG401x\_AWG*  
*property*), 118

`burst_mode` (*pymea-*  
*sure.instruments.agilent.Agilent33220A*  
*property*), 170

`burst_mode` (*pymea-*  
*sure.instruments.agilent.Agilent33500*  
*property*), 173

`burst_mode` (*pymea-*  
*sure.instruments.agilent.agilent33500.Agilent33500Ch-*  
*annel*  
*property*), 176

`burst_ncycles` (*pymea-*  
*sure.instruments.agilent.Agilent33220A*  
*property*), 170

`burst_ncycles` (*pymea-*  
*sure.instruments.agilent.Agilent33500 prop-*  
*erty*), 173

`burst_ncycles` (*pymea-*  
*sure.instruments.agilent.agilent33500.Agilent33500Channel*  
*property*), 176

`burst_number` (*pymea-*  
*sure.instruments.hp.HP8116A*  
*property*), 248

`burst_period` (*pymea-*  
*sure.instruments.agilent.Agilent33500 prop-*  
*erty*), 173

`burst_period` (*pymea-*  
*sure.instruments.agilent.agilent33500.Agilent33500Channel*  
*property*), 177

`burst_state` (*pymea-*  
*sure.instruments.agilent.Agilent33220A*  
*property*), 170

`burst_state` (*pymea-*  
*sure.instruments.agilent.Agilent33500*  
*property*), 173

`burst_state` (*pymea-*  
*sure.instruments.agilent.agilent33500.Agilent33500C-*  
*hannel*  
*property*), 177

`busy` (*pymea-*  
*sure.instruments.anaheimautomation.DPSeriesMotorControlle-*  
*r*  
*property*), 203

`bwlimit` (*pymea-*  
*sure.instruments.lecroy.LeCroyT3DSO1204*  
*property*), 364

`bwlimit` (*pymea-*  
*sure.instruments.lecroy.lecroyT3DSO1204.LeCroyT3DSO1204*  
*property*), 374

`bwlimit` (*pymea-*  
*sure.instruments.teledyne.teledyne\_oscilloscope.TeledyneC-*  
*hannel*  
*property*), 473

`bwlimit` (*pymea-*  
*sure.instruments.teledyne.TeledyneOscilloscope*  
*property*), 468

## C

`calibration()` (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
*method*), 409

`calibration_data` (*pymea-*  
*sure.instruments.hp.HP3478A*  
*property*),

`calibration_enabled` (*pymea-*  
*sure.instruments.anritsu.anritsuMS464xB.MeasurementChannel*  
*property*), 221

`calibration_enabled` (*pymea-*  
*sure.instruments.hp.HP3478A*  
*property*),

246

calibration\_factor (pymeasure.instruments.agilent.AgilentE4408B property), 156

calibration\_generation\_factor (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 259

calibration\_init() (pymeasure.instruments.lectroy.LeCroyT3DSOI204 method), 364

calibration\_measured\_value (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 468

calibration\_store\_factor() (pymeasure.instruments.keithley.Keithley2306 method), 297

capacitance (pymeasure.instruments.agilent.Agilent34450A property), 159

capacitance\_auto\_range (pymeasure.instruments.agilent.Agilent34450A property), 159

capacitance\_range (pymeasure.instruments.agilent.Agilent34450A property), 159

capacity (pymeasure.instruments.attocube.anc300.Axis property), 225

caplossvolt (pymeasure.instruments.andeenhagerling.AH2500A property), 205

caplossvolt (pymeasure.instruments.andeenhagerling.AH2700A property), 206

carrier\_enabled (pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel property), 417

carrier\_frequency (pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel property), 417

carrier\_level (pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel property), 417

cathode\_enabled (pymeasure.instruments.thyracont.smartline\_v1.SmartlineV1 property), 490

celcius (pymeasure.instruments.lakeshore.lakeshore\_base.LakeShoreTemperatureChannel property), 359

center\_at\_peak() (pymeasure.instruments.anritsu.AnritsuMS9710C method), 209

center\_frequency (pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG property), 122

center\_frequency (pymeasure.instruments.agilent.Agilent8257D property), 152

center\_frequency (pymeasure.instruments.agilent.AgilentE4408B property), 156

center\_frequency (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 259

center\_trigger() (pymeasure.instruments.lectroy.LeCroyT3DSOI204 method), 364

center\_trigger() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 468

ch() (pymeasure.instruments.keithley.Keithley2306 method), 297

ch() (pymeasure.instruments.lectroy.LeCroyT3DSOI204 method), 364

ch() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 469

ch\_1 (pymeasure.instruments.activetechnologies.AWG401x\_AFG attribute), 117

ch\_1 (pymeasure.instruments.agilent.Agilent33500 attribute), 172

ch\_1 (pymeasure.instruments.agilent.Agilent33521A attribute), 176

ch\_1 (pymeasure.instruments.keithley.Keithley2200 attribute), 334

ch\_1 (pymeasure.instruments.keysight.KeysightE36312A attribute), 343

ch\_1 (pymeasure.instruments.lectroy.LeCroyT3DSOI204 attribute), 360

ch\_1 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 375

ch\_1 (pymeasure.instruments.siglenttechnologies.SPD1168X attribute), 426

ch\_1 (pymeasure.instruments.siglenttechnologies.SPD1305X attribute), 427

ch\_1 (pymeasure.instruments.teledyne.TeledyneOscilloscope attribute), 468

ch\_1 (pymeasure.instruments.teledyne.TeledyneT3AFG attribute), 465

ch\_1 (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart attribute), 495

ch\_2 (pymeasure.instruments.activetechnologies.AWG401x\_AFG attribute), 173

ch\_2 (pymeasure.instruments.agilent.Agilent33500 attribute), 173

ch\_2 (pymeasure.instruments.agilent.Agilent33521A attribute), 176

ch\_2 (pymeasure.instruments.keithley.Keithley2200 attribute), 334

ch\_2 (pymeasure.instruments.keysight.KeysightE36312A attribute), 343

ch\_2 (pymeasure.instruments.lectroy.LeCroyT3DSOI204 attribute), 361

ch\_2 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 375

attribute), 375

ch\_2 (pymeasure.instruments.teledyne.TeledyneOscilloscope attribute), 468

ch\_2 (pymeasure.instruments.teledyne.TeledyneT3AFG attribute), 465

ch\_2 (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart attribute), 495

ch\_3 (pymeasure.instruments.keithley.Keithley2200 attribute), 334

ch\_3 (pymeasure.instruments.keysight.KeysightE36312A attribute), 343

ch\_3 (pymeasure.instruments.lecroy.LeCroyT3DSO1204 attribute), 361

ch\_3 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 375

ch\_3 (pymeasure.instruments.teledyne.TeledyneOscilloscope attribute), 468

ch\_3 (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart attribute), 495

ch\_4 (pymeasure.instruments.lecroy.LeCroyT3DSO1204 attribute), 361

ch\_4 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 375

ch\_4 (pymeasure.instruments.teledyne.TeledyneOscilloscope attribute), 468

ch\_4 (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart attribute), 495

ch\_5 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 376

ch\_5 (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart attribute), 495

ch\_6 (pymeasure.instruments.mksinst.mks937b.MKS937B attribute), 376

ch\_A (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X attribute), 122

ch\_A (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X attribute), 123

ch\_B (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X attribute), 122

ch\_B (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X attribute), 123

change\_source\_current (pymeasure.instruments.advantest.advantestR624X.SMU property), 138

change\_source\_voltage (pymeasure.instruments.advantest.advantestR624X.SMU property), 135

Channel (class in pymeasure.instruments), 111

channel (pymeasure.instruments.bkprecision.BKPrecision9130B property), 226

channel1 (pymeasure.instruments.srs.SR830 property), 443

channel1\_enabled (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart property), 495

channel12 (pymeasure.instruments.srs.SR830 property), 443

channel12\_enabled (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart property), 496

channel\_down\_relative() (pymeasure.instruments.rohdeschwarz.sfm.SFM method), 409

channel\_function (pymeasure.instruments.agilent.agilent4156.SMU property), 166

channel\_function (pymeasure.instruments.agilent.agilent4156.VSU property), 169

channel\_mode (pymeasure.instruments.agilent.agilent4156.SMU property), 166

channel\_mode (pymeasure.instruments.agilent.agilent4156.VMU property), 169

channel\_mode (pymeasure.instruments.agilent.agilent4156.VSU property), 169

channel\_sweep\_start (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 409

channel\_sweep\_step (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 409

channel\_sweep\_stop (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 409

channel\_up\_relative() (pymeasure.instruments.rohdeschwarz.sfm.SFM method), 409

Channels (class in pymeasure.instruments.activetechnologies.AWG401x), 120

channels\_from\_rows\_columns() (pymeasure.instruments.keithley.Keithley2700 method), 315

check\_done() (pymeasure.instruments.hp.hp856Xx.HP856Xx method), 253

check\_errors() (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X method), 123

check\_errors() (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 method), 184

- `check_errors()` (*pymeasure.instruments.agilent.agilentB1500.SMU* method), 188
- `check_errors()` (*pymeasure.instruments.anaheimautomation.DPSeriesMotorController* method), 203
- `check_errors()` (*pymeasure.instruments.andenhagerling.AH2700A* method), 206
- `check_errors()` (*pymeasure.instruments.anritsu.AnritsuMS464xB* method), 216
- `check_errors()` (*pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel* method), 221
- `check_errors()` (*pymeasure.instruments.Channel* method), 111
- `check_errors()` (*pymeasure.instruments.common\_base.CommonBase* method), 105
- `check_errors()` (*pymeasure.instruments.fwbell.FWBell5080* method), 235
- `check_errors()` (*pymeasure.instruments.heidenhain.ND287* method), 237
- `check_errors()` (*pymeasure.instruments.hp.HP3437A* method), 244
- `check_errors()` (*pymeasure.instruments.hp.HP3478A* method), 246
- `check_errors()` (*pymeasure.instruments.hp.HP6632A* method), 285
- `check_errors()` (*pymeasure.instruments.hp.HP8116A* method), 248
- `check_errors()` (*pymeasure.instruments.hp.HP8657B* method), 282
- `check_errors()` (*pymeasure.instruments.Instrument* method), 109
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2000* method), 288
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2200* method), 336
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2260B* method), 295
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2306* method), 297
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2400* method), 301
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2450* method), 309
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2600* method), 332
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2700* method), 316
- `check_errors()` (*pymeasure.instruments.keithley.Keithley2750* method), 330
- `check_errors()` (*pymeasure.instruments.keithley.Keithley6221* method), 320
- `check_errors()` (*pymeasure.instruments.keithley.Keithley6517B* method), 326
- `check_errors()` (*pymeasure.instruments.keysight.KeysightE36312A* method), 346
- `check_errors()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 364
- `check_errors()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 428
- `check_errors()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 435
- `check_errors()` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* method), 456
- `check_errors()` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* method), 461
- `check_errors()` (*pymeasure.instruments.teledyne.TeledyneT3AFG* method), 466
- `check_errors()` (*pymeasure.instruments.texio.TexioPSW360L30* method), 485
- `check_get_errors()` (*pymeasure.instruments.andenhagerling.AH2700A* method), 206
- `check_get_errors()` (*pymeasure.instruments.Channel* method), 111
- `check_get_errors()` (*pymeasure.instruments.common\_base.CommonBase* method), 105
- `check_get_errors()` (*pymeasure.instruments.fwbell.FWBell5080* method), 235
- `check_get_errors()` (*pymeasure.instruments.Instrument* method), 109

<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2000</i> method), 288	<code>check_get_errors()</code> ( <i>pymeasure.instruments.texio.TexioPSW360L30</i> method), 485
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2200</i> method), 336	<code>check_get_estimates_signature()</code> ( <i>pymeasure.display.widgets.estimator_widget.EstimatorWidget</i> method), 90
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2260B</i> method), 295	<code>check_idle()</code> ( <i>pymeasure.instruments.agilent.agilentB1500.AgilentB1500</i> method), 184
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2306</i> method), 298	<code>check_parameters()</code> ( <i>pymeasure.experiment.procedure.Procedure</i> method), 73
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2400</i> method), 301	<code>check_selftest_errors()</code> ( <i>pymeasure.instruments.hp.HP6632A</i> method), 285
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2450</i> method), 309	<code>check_set_errors()</code> ( <i>pymeasure.instruments.ametek.Ametek7270</i> method), 198
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2600</i> method), 332	<code>check_set_errors()</code> ( <i>pymeasure.instruments.andeenhagerling.AH2700A</i> method), 206
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2700</i> method), 316	<code>check_set_errors()</code> ( <i>pymeasure.instruments.attocube.anc300.ANC300Controller</i> method), 224
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2750</i> method), 330	<code>check_set_errors()</code> ( <i>pymeasure.instruments.Channel</i> method), 111
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley6221</i> method), 320	<code>check_set_errors()</code> ( <i>pymeasure.instruments.common_base.CommonBase</i> method), 105
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley6517B</i> method), 326	<code>check_set_errors()</code> ( <i>pymeasure.instruments.fwbell.FWBell5080</i> method), 235
<code>check_get_errors()</code> ( <i>pymeasure.instruments.keysight.KeysightE36312A</i> method), 346	<code>check_set_errors()</code> ( <i>pymeasure.instruments.hcp.TC038</i> method), 238
<code>check_get_errors()</code> ( <i>pymeasure.instruments.lecroy.LeCroyT3DSO1204</i> method), 364	<code>check_set_errors()</code> ( <i>pymeasure.instruments.hcp.TC038D</i> method), 239
<code>check_get_errors()</code> ( <i>pymeasure.instruments.signalrecovery.DSP7225</i> method), 428	<code>check_set_errors()</code> ( <i>pymeasure.instruments.Instrument</i> method), 110
<code>check_get_errors()</code> ( <i>pymeasure.instruments.signalrecovery.DSP7265</i> method), 435	<code>check_set_errors()</code> ( <i>pymeasure.instruments.ipgphotonics.yar.YAR</i> method), 286
<code>check_get_errors()</code> ( <i>pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38</i> method), 456	<code>check_set_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2000</i> method), 288
<code>check_get_errors()</code> ( <i>pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65</i> method), 461	<code>check_set_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2200</i> method), 336
<code>check_get_errors()</code> ( <i>pymeasure.instruments.teledyne.TeledyneT3AFG</i> method), 466	<code>check_set_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2260B</i> method), 295
	<code>check_set_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2306</i> method), 298
	<code>check_set_errors()</code> ( <i>pymeasure.instruments.keithley.Keithley2400</i> method), 301

- [method](#)), 301
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley2450](#) [method](#)), 309
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley2600](#) [method](#)), 332
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley2700](#) [method](#)), 316
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley2750](#) [method](#)), 330
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley6221](#) [method](#)), 320
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keithley.Keithley6517B](#) [method](#)), 326
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.keysight.KeysightE36312A](#) [method](#)), 346
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.lecroy.LeCroyT3DSO1204](#) [method](#)), 364
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.mksinst.mks937b.MKS937B](#) [method](#)), 376
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.novanta.Fpu60](#) [method](#)), 392
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [method](#)), 428
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [method](#)), 435
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.tdk.tdk\\_gen40\\_38.TDK\\_Gen40\\_38](#) [method](#)), 457
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.tdk.tdk\\_gen80\\_65.TDK\\_Gen80\\_65](#) [method](#)), 461
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.teledyne.TeledyneT3AFG](#) [method](#)), 466
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.texio.TexioPSW360L30](#) [method](#)), 485
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.thyracont.smartline\\_v1.SmartlineV1](#) [method](#)), 490
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.thyracont.smartline\\_v2.SmartlineV2](#) [method](#)), 492
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.toptica.ibeamsmart.IBeamSmart](#) [method](#)), 496
- [check\\_set\\_errors\(\)](#) ([pymeasure.instruments.velleman.VellemanK8090](#) [method](#)), 498
- [check\\_stop\(\)](#) ([pymeasure.display.windows.plotter\\_window.PlotterWindow](#) [method](#)), 100
- [choices](#) ([pymeasure.experiment.parameters.ListParameter](#) [property](#)), 76
- [clear\(\)](#) ([pymeasure.display.manager.BaseManager](#) [method](#)), 87
- [clear\(\)](#) ([pymeasure.instruments.andenhagerling.AH2700A](#) [method](#)), 206
- [clear\(\)](#) ([pymeasure.instruments.fwbell.FWBell5080](#) [method](#)), 235
- [clear\(\)](#) ([pymeasure.instruments.hp.HP6632A](#) [method](#)), 285
- [clear\(\)](#) ([pymeasure.instruments.hp.HP8657B](#) [method](#)), 282
- [clear\(\)](#) ([pymeasure.instruments.Instrument](#) [method](#)), 110
- [clear\(\)](#) ([pymeasure.instruments.ipgphotonics.yar.YAR](#) [method](#)), 286
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2000](#) [method](#)), 288
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2260B](#) [method](#)), 296
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2306](#) [method](#)), 298
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2400](#) [method](#)), 301
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2450](#) [method](#)), 309
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2600](#) [method](#)), 333
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2700](#) [method](#)), 316
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley2750](#) [method](#)), 330
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley6221](#) [method](#)), 320
- [clear\(\)](#) ([pymeasure.instruments.keithley.Keithley6517B](#) [method](#)), 326
- [clear\(\)](#) ([pymeasure.instruments.keysight.KeysightE36312A](#) [method](#)), 346
- [clear\(\)](#) ([pymeasure.instruments.lecroy.LeCroyT3DSO1204](#) [method](#)), 364
- [clear\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [method](#)), 429
- [clear\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [method](#)), 435

`clear()` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` method), 457  
`clear()` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` method), 97  
`clear()` (`pymeasure.instruments.teledyne.TeledyneT3AFG` method), 466  
`clear()` (`pymeasure.instruments.temptronic.ATSBASE` method), 476  
`clear()` (`pymeasure.instruments.texio.TexioPSW360L30` method), 485  
`clear_average_count()` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` method), 221  
`clear_buffer()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 184  
`clear_display()` (`pymeasure.instruments.agilent.Agilent33500` method), 173  
`clear_errors()` (`pymeasure.instruments.newport.ESP300` method), 377  
`clear_measurement_buffer()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 132  
`clear_overload()` (`pymeasure.instruments.srs.SR570` method), 441  
`clear_plot()` (`pymeasure.experiment.experiment.Experiment` method), 71  
`clear_ramp_set()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 227  
`clear_sequence()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 227  
`clear_sequence()` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 421  
`clear_status()` (`pymeasure.instruments.keysight.KeysightDSOX1102G` method), 339  
`clear_status_register()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 124  
`clear_timer()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 184  
`clear_widget()` (`pymeasure.display.widgets.plot_widget.PlotWidget` method), 92  
`clear_widget()` (`pymeasure.display.widgets.tab_widget.TabWidget` method), 94  
`clear_write_trace()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 265  
`cli_args` (`pymeasure.experiment.parameters.Parameter` property), 77  
`close()` (`pymeasure.adapters.Adapter` method), 48  
`close()` (`pymeasure.adapters.FakeAdapter` method), 67  
`close()` (`pymeasure.adapters.PrologixAdapter` method), 68  
`close()` (`pymeasure.adapters.SerialAdapter` method), 54  
`close()` (`pymeasure.adapters.TelnetAdapter` method), 64  
`close()` (`pymeasure.adapters.VISAAdapter` method), 51  
`close()` (`pymeasure.adapters.VXI11Adapter` method), 61  
`close()` (`pymeasure.instruments.keithley.Keithley2750` method), 331  
`close()` (`pymeasure.instruments.keysight.KeysightN7776C` method), 342  
`close_rows_to_columns()` (`pymeasure.instruments.keithley.Keithley2700` method), 316  
`closed_channels` (`pymeasure.instruments.keithley.Keithley2700` property), 316  
`closed_channels` (`pymeasure.instruments.keithley.Keithley2750` property), 331  
`CMU_MEASUREMENT` (`pymeasure.instruments.agilent.agilentB1500.WaitTimeType` attribute), 196  
`CNT91` (class in `pymeasure.instruments.pendulum.cnt91`), 404  
`coder_adjust()` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` method), 409  
`coder_id_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 409  
`coder_modulation_degree` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410  
`coder_pilot_deviation` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410  
`coder_pilot_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410  
`coilconst` (`pymeasure.instruments.ami.AMI430` property), 201  
`collapse_channel_string()` (`pymeasure.instruments.ni.virtualbench.VirtualBench`

- method*), 391
- `colormap()` (*pymeasure.display.curves.ResultsImage method*), 85
- `columnCount()` (*pymeasure.display.widgets.sequencer\_widget.SequencerTableModel method*), 93
- `columnCount()` (*pymeasure.display.widgets.table\_widget.PandasModelBase method*), 96
- `combined_pressure1` (*pymeasure.instruments.mksinst.mks937b.MKS937B property*), 376
- `combined_pressure2` (*pymeasure.instruments.mksinst.mks937b.MKS937B property*), 376
- `ComboBoxDelegate` (*class in pymeasure.display.widgets.sequencer\_widget*), 92
- `COMMAND_COMPLETE` (*pymeasure.instruments.hp.hp856Xx.StatusRegister attribute*), 279
- `CommonBase` (*class in pymeasure.instruments.common\_base*), 103
- `CommonBase.BaseChannelCreator` (*class in pymeasure.instruments.common\_base*), 103
- `CommonBase.ChannelCreator` (*class in pymeasure.instruments.common\_base*), 103
- `CommonBase.MultiChannelCreator` (*class in pymeasure.instruments.common\_base*), 104
- `complement_enabled` (*pymeasure.instruments.hp.HP8116A property*), 249
- `complete` (*pymeasure.instruments.andeenhagerling.AH2700A property*), 206
- `complete` (*pymeasure.instruments.fwbell.FWBell5080 property*), 235
- `complete` (*pymeasure.instruments.hp.HP8116A property*), 249
- `complete` (*pymeasure.instruments.Instrument property*), 110
- `complete` (*pymeasure.instruments.keithley.Keithley2000 property*), 288
- `complete` (*pymeasure.instruments.keithley.Keithley2200 property*), 337
- `complete` (*pymeasure.instruments.keithley.Keithley2260B property*), 296
- `complete` (*pymeasure.instruments.keithley.Keithley2306 property*), 298
- `complete` (*pymeasure.instruments.keithley.Keithley2400 property*), 301
- `complete` (*pymeasure.instruments.keithley.Keithley2450 property*), 309
- `complete` (*pymeasure.instruments.keithley.Keithley2600 property*), 333
- `complete` (*pymeasure.instruments.keithley.Keithley2700 property*), 316
- `complete` (*pymeasure.instruments.keithley.Keithley2750 property*), 331
- `complete` (*pymeasure.instruments.keithley.Keithley6221 property*), 320
- `complete` (*pymeasure.instruments.keithley.Keithley6517B property*), 326
- `complete` (*pymeasure.instruments.keysight.KeysightE36312A property*), 346
- `complete` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204 property*), 364
- `complete` (*pymeasure.instruments.signalrecovery.DSP7225 property*), 429
- `complete` (*pymeasure.instruments.signalrecovery.DSP7265 property*), 435
- `complete` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property*), 457
- `complete` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property*), 461
- `complete` (*pymeasure.instruments.teledyne.TeledyneT3AFG property*), 466
- `complete` (*pymeasure.instruments.texio.TexioPSW360L30 property*), 485
- `compliance` (*pymeasure.instruments.agilent.agilent4156.SMU property*), 167
- `compliance` (*pymeasure.instruments.agilent.agilent4156.VARD property*), 168
- `compliance` (*pymeasure.instruments.agilent.agilent4156.VARX property*), 168
- `COMPLIANCE_AND_FORCE_SIDE` (*pymeasure.instruments.agilent.agilentB1500.MeasOpMode attribute*), 194
- `compliance_current` (*pymeasure.instruments.keithley.Keithley2400 property*), 301
- `compliance_current` (*pymeasure.instruments.keithley.Keithley2450 property*), 309
- `compliance_current` (*pymeasure.instruments.yokogawa.Yokogawa7651 property*), 500
- `COMPLIANCE_SIDE` (*pymeasure.instruments.agilent.agilentB1500.MeasOpMode attribute*), 194
- `compliance_voltage` (*pymeasure.instruments.keithley.Keithley2400 property*), 301
- `compliance_voltage` (*pymeasure.instruments.keithley.Keithley2450 property*), 309
- `compliance_voltage` (*pymeasure.instruments.yokogawa.Yokogawa7651 property*), 500

CompliancePolarity (class in pymeasure.instruments.agilent.agilentB1500), 195

compressor\_enable (pymeasure.instruments.temptronic.ATSBBase property), 477

config (pymeasure.instruments.andeenhagerling.AH2500A property), 205

config (pymeasure.instruments.andeenhagerling.AH2700A property), 206

config\_amplitude\_modulation() (pymeasure.instruments.agilent.Agilent8257D method), 152

config\_buffer() (pymeasure.instruments.keithley.Keithley2000 method), 288

config\_buffer() (pymeasure.instruments.keithley.Keithley2400 method), 301

config\_buffer() (pymeasure.instruments.keithley.Keithley2450 method), 309

config\_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 316

config\_buffer() (pymeasure.instruments.keithley.Keithley6221 method), 320

config\_buffer() (pymeasure.instruments.keithley.Keithley6517B method), 326

config\_low\_freq\_out() (pymeasure.instruments.agilent.Agilent8257D method), 153

config\_pulse\_modulation() (pymeasure.instruments.agilent.Agilent8257D method), 153

config\_step\_sweep() (pymeasure.instruments.agilent.Agilent8257D method), 153

configure() (pymeasure.instruments.agilent.agilent4156.Agilent4156 method), 165

configure() (pymeasure.instruments.temptronic.ATSBBase method), 477

configure\_ac\_current() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 381

configure\_alarm() (pymeasure.instruments.lakeshore.LakeShore211 method), 352

configure\_analog\_channel() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 385

configure\_analog\_channel\_characteristics() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 385

configure\_analog\_edge\_trigger() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 386

configure\_analog\_pulse\_width\_trigger() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 386

configure\_arbitrary\_waveform() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 383

configure\_arbitrary\_waveform\_gain\_and\_offset() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 383

configure\_capacitance() (pymeasure.instruments.agilent.Agilent34450A method), 159

configure\_continuity() (pymeasure.instruments.agilent.Agilent34450A method), 160

configure\_current() (pymeasure.instruments.agilent.Agilent34450A method), 160

configure\_current\_output() (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply method), 389

configure\_dc\_current() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 381

configure\_dc\_voltage() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 381

configure\_diode() (pymeasure.instruments.agilent.Agilent34450A method), 160

configure\_frequency() (pymeasure.instruments.agilent.Agilent34450A method), 160

configure\_frequency\_array\_measurement() (pymeasure.instruments.pendulum.cnt91.CNT91 method), 405

configure\_immediate\_trigger() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 386

configure\_measurement() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 381

configure\_relay() (pymeasure.instruments.lakeshore.LakeShore211 method), 352

configure\_resistance() (pymeasure.instruments.agilent.Agilent34450A method), 160

configure\_standard\_waveform() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 383

- `method`), 383
- `configure_temperature()` (`pymeasure.instruments.agilent.Agilent34450A` `method`), 160
- `configure_timer()` (`pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDCtime` `method`), 425
- `configure_timing()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalModule` `method`), 386
- `configure_trigger_delay()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalModule` `method`), 386
- `configure_voltage()` (`pymeasure.instruments.agilent.Agilent34450A` `method`), 160
- `configure_voltage_output()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply` `method`), 389
- `ConsoleArgumentParser` (class in `pymeasure.display.console`), 83
- `ConsoleBrowserItem` (class in `pymeasure.display.console`), 83
- `constant_value` (`pymeasure.instruments.agilent.agilent4156.SMU` `property`), 167
- `constant_value` (`pymeasure.instruments.agilent.agilent4156.VSU` `property`), 169
- `contact_current_1` (`pymeasure.instruments.razorbill.razorbillRP100` `property`), 406
- `contact_current_2` (`pymeasure.instruments.razorbill.razorbillRP100` `property`), 406
- `contact_voltage_1` (`pymeasure.instruments.razorbill.razorbillRP100` `property`), 406
- `contact_voltage_2` (`pymeasure.instruments.razorbill.razorbillRP100` `property`), 406
- `contextMenuEvent()` (`pymeasure.display.widgets.dock_widget.DockWidget` `method`), 95
- `continue_single_sweep()` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` `method`), 420
- `continuity` (`pymeasure.instruments.agilent.Agilent34450A` `property`), 161
- `continuous` (`pymeasure.instruments.pendulum.cnt91.CNT91` `property`), 405
- `continuous_sweep` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` `property`), 420
- `control()` (`pymeasure.instruments.common_base.CommonBase` `static method`), 105
- `control()` (`pymeasure.instruments.fakes.FakeInstrument` `static method`), 113
- `control()` (`pymeasure.instruments.keysight.KeysightE36312A` `static method`), 346
- `control()` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` `static method`), 365
- `control_mode` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` `property`), 422
- `control_mode` (`pymeasure.instruments.hp.HP8116A` `property`), 249
- `control_mode` (`pymeasure.instruments.oxfordinstruments.IPS120_10` `property`), 400
- `control_mode` (`pymeasure.instruments.oxfordinstruments.ITC503` `property`), 395
- `controllerBoardVersion` (`pymeasure.instruments.attocube.anc300.ANC300Controller` `property`), 224
- `conversion_loss` (`pymeasure.instruments.hp.HP8561B` `property`), 275
- `convert()` (`pymeasure.experiment.parameters.BooleanParameter` `method`), 74
- `convert()` (`pymeasure.experiment.parameters.FloatParameter` `method`), 75
- `convert()` (`pymeasure.experiment.parameters.IntegerParameter` `method`), 76
- `convert()` (`pymeasure.experiment.parameters.ListParameter` `method`), 76
- `convert()` (`pymeasure.experiment.parameters.Parameter` `method`), 77
- `convert()` (`pymeasure.experiment.parameters.PhysicalParameter` `method`), 78
- `convert()` (`pymeasure.experiment.parameters.VectorParameter` `method`), 78
- `convert_timestamp_to_values()` (`pymeasure.instruments.ni.virtualbench.VirtualBench` `method`), 391
- `convert_values_to_datetime()` (`pymeasure.instruments.ni.virtualbench.VirtualBench` `method`), 391
- `convert_values_to_timestamp()` (`pymeasure.instruments.ni.virtualbench.VirtualBench` `method`), 391
- `copy_active_setup_file` (`pymeasure.instruments.temptronic.ATSB` `property`), 477
- `copy_active_setup_file` (`pymeasure.instruments.temptronic.ECO560` `attribute`), 484

`copy_data_file()` (*pymeasure.instruments.anritsu.AnritsuMS464xB method*), 216

`COR` (class in *pymeasure.instruments.advantest.advantestR624X*), 144

`coupling` (*pymeasure.instruments.hp.hp856Xx.HP856Xx attribute*), 252

`coupling` (*pymeasure.instruments.signalrecovery.DSP7225 property*), 429

`coupling` (*pymeasure.instruments.signalrecovery.DSP7265 property*), 435

`coupling` (*pymeasure.instruments.teledyne.teledyne\_oscilloscope.teledyne\_oscilloscope property*), 473

`CouplingMode` (class in *pymeasure.instruments.hp.hp856Xx*), 277

`create_directory()` (*pymeasure.instruments.anritsu.AnritsuMS464xB method*), 216

`create_fft_trace_window()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx method*), 262

`create_filename()` (in module *pymeasure.experiment.experiment*), 72

`create_marker()` (*pymeasure.instruments.rohdeschwarz.fsl.FSL method*), 420

`createEditor()` (*pymeasure.display.widgets.sequencer\_widget.ComboBoxDelegate method*), 92

`createEditor()` (*pymeasure.display.widgets.sequencer\_widget.LineEditDelegate method*), 92

`Crosshairs` (class in *pymeasure.display.curves*), 84

`CSVFormatter` (class in *pymeasure.experiment.results*), 79

`current` (*pymeasure.instruments.agilent.Agilent34450A property*), 161

`CURRENT` (*pymeasure.instruments.agilent.agilentB1500.MeasOpMode property*), 194

`current` (*pymeasure.instruments.aja.DCXS property*), 197

`current` (*pymeasure.instruments.bkprecision.BKPrecision9130B property*), 226

`current` (*pymeasure.instruments.danfysik.Danfysik8500 property*), 227

`current` (*pymeasure.instruments.deltaelektronika.SM7045D property*), 230

`current` (*pymeasure.instruments.eurotest.EurotestHPP120256 property*), 232

`current` (*pymeasure.instruments.hp.HP6632A property*), 285

`current` (*pymeasure.instruments.ipgphotonics.yar.YAR property*), 286

`current` (*pymeasure.instruments.keithley.Keithley2000 property*), 289

`current` (*pymeasure.instruments.keithley.keithley2200.PSChannel property*), 338

`current` (*pymeasure.instruments.keithley.Keithley2260B property*), 296

`current` (*pymeasure.instruments.keithley.Keithley2400 property*), 301

`current` (*pymeasure.instruments.keithley.Keithley2450 property*), 309

`current` (*pymeasure.instruments.keithley.Keithley6517B property*), 327

`current` (*pymeasure.instruments.teledyne\_oscilloscope.teledyne\_oscilloscope property*), 351

`current` (*pymeasure.instruments.keysight.KeysightE36312A.VoltageChannel property*), 341

`current` (*pymeasure.instruments.keysight.KeysightN5767A property*), 341

`current` (*pymeasure.instruments.novanta.Fpu60 property*), 392

`current` (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040 property*), 422

`current` (*pymeasure.instruments.siglenttechnologies.siglent\_spdbase.SPDBASE property*), 425

`current` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property*), 457

`current` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property*), 461

`current` (*pymeasure.instruments.texio.TexioPSW360L30 property*), 485

`current` (*pymeasure.instruments.toptica.ibeamsmart.IBeamSmart property*), 496

`current_ac` (*pymeasure.instruments.agilent.Agilent34410A property*), 159

`current_ac` (*pymeasure.instruments.agilent.Agilent34450A property*), 161

`current_ac` (*pymeasure.instruments.hp.HP34401A property*), 241

`current_ac_auto_range` (*pymeasure.instruments.agilent.Agilent34450A property*), 161

`current_ac_bandwidth` (*pymeasure.instruments.keithley.Keithley2000 property*), 289

`current_ac_digits` (*pymeasure.instruments.keithley.Keithley2000 property*), 289

`current_ac_nplc` (*pymeasure.instruments.keithley.Keithley2000 property*), 289

`current_ac_range` (*pymeasure.instruments.agilent.Agilent34450A property*), 161

`current_ac_range` (*pymeasure.instruments.keithley.Keithley2000 property*), 289

`current_ac_reference` (`pymeasure.instruments.keithley.Keithley2000` property), 289  
`current_ac_resolution` (`pymeasure.instruments.agilent.Agilent34450A` property), 161  
`current_auto_range` (`pymeasure.instruments.agilent.Agilent34450A` property), 161  
`current_configuration` (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscopeChannel` property), 473  
`current_cycle_count` (`pymeasure.instruments.temptronic.ATSBBase` property), 477  
`current_dc` (`pymeasure.instruments.agilent.Agilent34410A` property), 159  
`current_dc` (`pymeasure.instruments.hp.HP34401A` property), 241  
`current_digits` (`pymeasure.instruments.keithley.Keithley2000` property), 289  
`current_filter_count` (`pymeasure.instruments.keithley.Keithley2450` property), 309  
`current_filter_state` (`pymeasure.instruments.keithley.Keithley2450` property), 310  
`current_filter_type` (`pymeasure.instruments.keithley.Keithley2450` property), 310  
`current_fixed_level_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 138  
`current_fixed_pulsed_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 138  
`current_limit` (`pymeasure.instruments.eurotest.EurotestHPP120256` property), 232  
`current_limit` (`pymeasure.instruments.keithley.keithley2200.PSChannel` property), 338  
`current_limit` (`pymeasure.instruments.keithley.Keithley2260B` property), 296  
`current_limit` (`pymeasure.instruments.keysight.keysightE36312A.VoltageChannel` property), 351  
`current_limit` (`pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDCChannel` property), 425  
`current_limit` (`pymeasure.instruments.texio.TexioPSW360L30` property), 485  
`current_limit` (`pymeasure.instruments.yokogawa.YokogawaGS200` property), 501  
`current_measured` (`pymeasure.instruments.oxfordinstruments.IPS120_10` property), 400  
`current_name` (`pymeasure.instruments.agilent.agilent4156.SMU` property), 167  
`current_nplc` (`pymeasure.instruments.keithley.Keithley2400` property), 302  
`current_nplc` (`pymeasure.instruments.keithley.Keithley2450` property), 310  
`current_nplc` (`pymeasure.instruments.keithley.Keithley6517B` property), 327  
`current_output_off_state` (`pymeasure.instruments.keithley.Keithley2450` property), 310  
`current_ppm` (`pymeasure.instruments.danfysik.Danfysik8500` property), 227  
`current_pulsed_source()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 138  
`current_pulsed_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 139  
`current_random_pulsed_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 140  
`current_random_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 139  
`current_range` (`pymeasure.instruments.agilent.Agilent34450A` property), 161  
`current_range` (`pymeasure.instruments.eurotest.EurotestHPP120256` property), 232  
`current_range` (`pymeasure.instruments.keithley.Keithley2000` property), 289  
`current_range` (`pymeasure.instruments.keithley.Keithley2400` property), 302  
`current_range` (`pymeasure.instruments.keithley.Keithley2450` property), 310

- `current_range` (`pymeasure.instruments.keithley.Keithley6517B` property), 327
- `current_range` (`pymeasure.instruments.keysight.KeysightN5767A` property), 341
- `current_reference` (`pymeasure.instruments.keithley.Keithley2000` property), 289
- `current_resolution` (`pymeasure.instruments.agilent.Agilent34450A` property), 161
- `current_set_random_memory()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 140
- `current_setpoint` (`pymeasure.instruments.danfysik.Danfysik8500` property), 227
- `current_setpoint` (`pymeasure.instruments.oxfordinstruments.IPS120_10` property), 400
- `current_setpoint` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` property), 457
- `current_setpoint` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` property), 461
- `current_source()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 137
- `current_step` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` property), 422
- `current_sweep()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 138
- `current_to_max()` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 422
- `current_to_min()` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 422
- `CurrentRange` (class in `pymeasure.instruments.advantest.advantestR624X`), 143
- `curve_buffer_bits` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `curve_buffer_bits` (`pymeasure.instruments.signalrecovery.DSP7265` property), 435
- `curve_buffer_interval` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `curve_buffer_interval` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `curve_buffer_length` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `curve_buffer_length` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `curve_buffer_status` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `curve_buffer_status` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `CustomIntEnum` (class in `pymeasure.instruments.agilent.agilentB1500`), 193
- `cw_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410
- `cw_mode_enabled` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 221
- `cw_number_of_points` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 221
- `CXN` (class in `pymeasure.instruments.tcpowerconversion`), 452
- `CXNStatus` (class in `pymeasure.instruments.tcpowerconversion`), 453
- `cycling_enable` (`pymeasure.instruments.temptronic.ATSBBase` property), 477
- `cycling_stopped()` (`pymeasure.instruments.temptronic.ATSBBase` method), 477
- ## D
- `dac1` (`pymeasure.instruments.ametek.Ametek7270` property), 198
- `dac1` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `dac1` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `dac1` (`pymeasure.instruments.srs.SR830` property), 443
- `dac1` (`pymeasure.instruments.srs.SR860` property), 447
- `dac2` (`pymeasure.instruments.ametek.Ametek7270` property), 198
- `dac2` (`pymeasure.instruments.signalrecovery.DSP7225` property), 429
- `dac2` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `dac2` (`pymeasure.instruments.srs.SR830` property), 443
- `dac2` (`pymeasure.instruments.srs.SR860` property), 447

- `dac3` (`pymeasure.instruments.ametek.Ametek7270` property), 199
- `dac3` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `dac3` (`pymeasure.instruments.srs.SR830` property), 443
- `dac3` (`pymeasure.instruments.srs.SR860` property), 447
- `dac4` (`pymeasure.instruments.ametek.Ametek7270` property), 199
- `dac4` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436
- `dac4` (`pymeasure.instruments.srs.SR830` property), 443
- `dac4` (`pymeasure.instruments.srs.SR860` property), 447
- `Danfysik8500` (class in `pymeasure.instruments.danfysik`), 227
- `data` (`pymeasure.experiment.experiment.Experiment` property), 72
- `data` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155
- `data()` (`pymeasure.display.widgets.sequencer_widget.SequencerTreeModel` method), 93
- `data()` (`pymeasure.display.widgets.table_widget.PandasModelBase` method), 96
- `data_arb()` (`pymeasure.instruments.agilent.Agilent33500` method), 173
- `data_arb()` (`pymeasure.instruments.agilent.agilent33500.Agilent33500Channel` method), 177
- `data_complex` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155
- `data_drawing_enabled` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 216
- `data_format()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 184
- `data_log_magnitude` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155
- `data_magnitude` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155
- `data_memory_a_condition` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_a_size` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_a_values` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_b_condition` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_b_size` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_b_values` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_select` (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 209
- `data_memory_select` (`pymeasure.instruments.anritsu.AnritsuMS9740A` property), 211
- `data_phase` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155
- `data_variables` (`pymeasure.instruments.agilent.agilent4156.Agilent4156` property), 165
- `data_volatile_clear()` (`pymeasure.instruments.agilent.Agilent33500` method), 177
- `data_volatile_clear()` (`pymeasure.instruments.agilent.agilent33500.Agilent33500Channel` method), 177
- `datablock_header_format` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 216
- `datablock_numeric_format` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 217
- `datafile_frequency_unit` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 217
- `datafile_include_heading` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 217
- `datafile_parameter_format` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 217
- `date` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410
- `DBM` (`pymeasure.instruments.hp.hp856Xx.AmplitudeUnits` attribute), 277
- `DBMV` (`pymeasure.instruments.hp.hp856Xx.AmplitudeUnits` attribute), 277
- `DBUV` (`pymeasure.instruments.hp.hp856Xx.AmplitudeUnits` attribute), 277
- `DC` (`pymeasure.instruments.hp.hp856Xx.CouplingMode` attribute), 278
- `dc_mode()` (`pymeasure.instruments.lakeshore.LakeShore425` method), 358
- `dc_voltage` (`pymeasure.instruments.tcpowerconversion.CXN` property), 453
- `dcmode` (`pymeasure.instruments.srs.SR860` property), 447
- `DCXS` (class in `pymeasure.instruments.aja`), 196

`deactivate_marker()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* method), 267

`default_setup()` (*pymeasure.instruments.keysight.KeysightDSOX1102G* method), 339

`default_setup()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 366

`default_setup()` (*pymeasure.instruments.teledyne.TeledyneOscilloscope* method), 469

`define_arbitrary_waveform()` (*pymeasure.instruments.keithley.Keithley6221* method), 320

`define_position()` (*pymeasure.instruments.newport.esp300.Axis* method), 377

`delay` (*pymeasure.instruments.hp.HP3437A* property), 244

`delay` (*pymeasure.instruments.hp.HP6632A* property), 285

`delay_time` (*pymeasure.instruments.agilent.agilent4156.Agilent4156* property), 165

`delete_data_file()` (*pymeasure.instruments.anritsu.AnritsuMS464xB* method), 217

`delete_directory()` (*pymeasure.instruments.anritsu.AnritsuMS464xB* method), 217

`demand_current` (*pymeasure.instruments.oxfordinstruments.IPS120\_10* property), 400

`demand_field` (*pymeasure.instruments.oxfordinstruments.IPS120\_10* property), 400

`demodulation_agc_enabled` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 258

`demodulation_mode` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 258

`demodulation_time` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 258

`DemodulationMode` (class in *pymeasure.instruments.hp.hp856Xx*), 278

`deposition_time_min` (*pymeasure.instruments.aja.DCXS* property), 197

`deposition_time_sec` (*pymeasure.instruments.aja.DCXS* property), 197

`derivative_action_time` (*pymeasure.instruments.oxfordinstruments.ITC503* property), 396

`detectedfrequency` (*pymeasure.instruments.srs.SR860* property), 447

`DetectionModes` (class in *pymeasure.instruments.hp.hp856Xx*), 278

`detector_bandwidth` (*pymeasure.instruments.hp.HP34401A* property), 241

`detector_mode` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 252

`determine_valid_channels()` (*pymeasure.instruments.keithley.Keithley2700* method), 316

`deviation` (*pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel* property), 417

`device_address` (*pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2* property), 492

`device_operation_enable_register` (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 130

`device_operation_register` (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 131

`device_serial` (*pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2* property), 492

`device_type` (*pymeasure.instruments.thyracont.smartline\_v1.SmartlineV1* property), 490

`device_type` (*pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2* property), 492

`device_version` (*pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2* property), 492

`digital_out_enable_data` (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 130

`digitize()` (*pymeasure.instruments.keysight.KeysightDSOX1102G* method), 339

`diode` (*pymeasure.instruments.agilent.Agilent34450A* property), 161

`dip_search` (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209

`direction` (*pymeasure.instruments.anaheimautomation.DPSeriesMotorCo* property), 203

`DirectoryLineEdit` (class in *pymeasure.display.widgets.directory\_widget*), 90

`disable` (*pymeasure.instruments.agilent.agilent4156.SMU* property), 167

`disable` (*pymeasure.instruments.agilent.agilent4156.VMU* property), 169

`disable` (*pymeasure.instruments.agilent.agilent4156.VSU* property), 169

`disable()` (*pymeasure.instruments.agilent.Agilent8257D*

- method*), 153
- `disable()` (*pymeasure.instruments.agilent.agilentB1500.SMU* *method*), 188
- `disable()` (*pymeasure.instruments.anritsu.AnritsuMG3692A* *method*), 208
- `disable()` (*pymeasure.instruments.danfysik.Danfysik8500* *method*), 227
- `disable()` (*pymeasure.instruments.deltaelektronika.SM7045D* *method*), 230
- `disable()` (*pymeasure.instruments.keysight.KeysightN5767A* *method*), 341
- `disable()` (*pymeasure.instruments.newport.ESP300* *method*), 377
- `disable()` (*pymeasure.instruments.newport.esp300.Axis* *method*), 377
- `disable()` (*pymeasure.instruments.parker.ParkerGV6* *method*), 403
- `disable()` (*pymeasure.instruments.topica.ibeamsmart.IBeamSmart* *method*), 496
- `disable_all()` (*pymeasure.instruments.agilent.agilent4156.Agilent4156* *method*), 165
- `disable_amplitude_modulation()` (*pymeasure.instruments.agilent.Agilent8257D* *method*), 153
- `disable_averaging()` (*pymeasure.instruments.agilent.Agilent8722ES* *method*), 155
- `disable_bias()` (*pymeasure.instruments.srs.SR570* *method*), 441
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley2000* *method*), 289
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley2400* *method*), 302
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley2450* *method*), 310
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley2700* *method*), 317
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley6221* *method*), 321
- `disable_buffer()` (*pymeasure.instruments.keithley.Keithley6517B* *method*), 327
- `disable_control()` (*pymeasure.instruments.oxfordinstruments.IPS120\_10* *method*), 400
- `disable_emission()` (*pymeasure.instruments.novanta.Fpu60* *method*), 393
- `disable_filter()` (*pymeasure.instruments.keithley.Keithley2000* *method*), 289
- `disable_low_freq_out()` (*pymeasure.instruments.agilent.Agilent8257D* *method*), 153
- `disable_modulation()` (*pymeasure.instruments.agilent.Agilent8257D* *method*), 153
- `disable_offset_current()` (*pymeasure.instruments.srs.SR570* *method*), 441
- `disable_output_trigger()` (*pymeasure.instruments.keithley.Keithley2400* *method*), 302
- `disable_output_trigger()` (*pymeasure.instruments.keithley.Keithley6221* *method*), 321
- `disable_persistent_mode()` (*pymeasure.instruments.oxfordinstruments.IPS120\_10* *method*), 400
- `disable_persistent_switch()` (*pymeasure.instruments.ami.AMI430* *method*), 201
- `disable_pulse_modulation()` (*pymeasure.instruments.agilent.Agilent8257D* *method*), 153
- `disable_reference()` (*pymeasure.instruments.keithley.Keithley2000* *method*), 289
- `disable_rf()` (*pymeasure.instruments.anapico.APSIN12G* *method*), 205
- `disable_source()` (*pymeasure.instruments.keithley.Keithley2400* *method*), 302
- `disable_source()` (*pymeasure.instruments.keithley.Keithley2450* *method*), 310
- `disable_source()` (*pymeasure.instruments.keithley.Keithley6221* *method*), 321
- `disable_source()` (*pymeasure.instruments.keithley.Keithley6517B* *method*), 327
- `disable_source()` (*pymeasure.instruments.yokogawa.Yokogawa7651* *method*), 500
- `display` (*pymeasure.instruments.agilent.Agilent33500* *property*), 174
- `display` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* *property*), 457
- `display` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* *property*), 461
- `display` (*pymeasure.instruments.teledyne.teledyne\_oscilloscope.TeledyneC* *property*), 473

<code>display_active</code> ( <code>pymeasure.instruments.hp.HP6632A</code> property), 285	<code>display_parameter()</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> method), 469
<code>display_brightness</code> ( <code>pymeasure.instruments.keithley.Keithley2306</code> property), 298	<code>display_parameters</code> ( <code>pymeasure.instruments.hp.hp856Xx.HP856Xx</code> attribute), 257
<code>display_channel</code> ( <code>pymeasure.instruments.keithley.Keithley2306</code> property), 298	<code>display_reset()</code> ( <code>pymeasure.instruments.hp.HP3478A</code> method), 246
<code>display_closed_channels()</code> ( <code>pymeasure.instruments.keithley.Keithley2700</code> method), 317	<code>display_text</code> ( <code>pymeasure.instruments.hp.HP3478A</code> property), 246
<code>display_data</code> ( <code>pymeasure.instruments.thyracont.smartline_v2.SmartlineV2</code> property), 492	<code>display_text_data</code> ( <code>pymeasure.instruments.keithley.Keithley2200</code> property), 337
<code>display_enabled</code> ( <code>pymeasure.instruments.advantest.advantestR624X.AdvantestR624X</code> property), 131	<code>display_text_data</code> ( <code>pymeasure.instruments.keithley.Keithley2306</code> property), 298
<code>display_enabled</code> ( <code>pymeasure.instruments.hp.HP34401A</code> property), 241	<code>display_text_enabled</code> ( <code>pymeasure.instruments.keithley.Keithley2306</code> property), 298
<code>display_enabled</code> ( <code>pymeasure.instruments.keithley.Keithley2200</code> property), 337	<code>display_text_no_symbol</code> ( <code>pymeasure.instruments.hp.HP3478A</code> property), 246
<code>display_enabled</code> ( <code>pymeasure.instruments.keithley.Keithley2306</code> property), 298	<code>display_unit</code> ( <code>pymeasure.instruments.thyracont.smartline_v1.SmartlineV1</code> property), 490
<code>display_enabled</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 302	<code>display_unit</code> ( <code>pymeasure.instruments.thyracont.smartline_v2.SmartlineV2</code> property), 493
<code>display_enabled</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 321	<code>display_units</code> ( <code>pymeasure.instruments.lakeshore.LakeShore211</code> property), 352
<code>display_estimates()</code> ( <code>pymeasure.display.widgets.estimator_widget.EstimatorWidget</code> method), 90	<code>displayed_text</code> ( <code>pymeasure.instruments.hp.HP34401A</code> property), 241
<code>display_filter_enabled</code> ( <code>pymeasure.instruments.lakeshore.LakeShore421</code> property), 356	<code>do_fft()</code> ( <code>pymeasure.instruments.hp.hp856Xx.HP856Xx</code> method), 262
<code>display_layout</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS464xB</code> property), 217	<code>DockWidget</code> (class in <code>pymeasure.display.widgets.dock_widget</code> ), 95
<code>display_layout</code> ( <code>pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannelDOR</code> property), 221	<code>DOR</code> (class in <code>pymeasure.instruments.advantest.advantestR624X</code> ), 144
<code>display_line</code> ( <code>pymeasure.instruments.hp.hp856Xx.HP856Xx</code> attribute), 255	<code>download_data()</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> method), 339
<code>display_orientation</code> ( <code>pymeasure.instruments.thyracont.smartline_v2.SmartlineV2</code> property), 493	<code>download_image()</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> method), 339
<code>display_parameter</code> ( <code>pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscopeChannel</code> property), 474	<code>download_image()</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> method), 366
<code>display_parameter()</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> method), 366	

- method), 366
- download\_image() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 469
- download\_waveform() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 366
- download\_waveform() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 469
- DPSeriesMotorController (class in pymeasure.instruments.anaheimautomation), 202
- DriverChannel (class in pymeasure.instruments.toptica.ibeamsmart), 497
- DSP7225 (class in pymeasure.instruments.signalrecovery), 427
- DSP7265 (class in pymeasure.instruments.signalrecovery), 433
- dut\_constant (pymeasure.instruments.temptronic.ATSBBase property), 477
- dut\_mode (pymeasure.instruments.temptronic.ATSBBase property), 478
- dut\_temperature (pymeasure.instruments.temptronic.ATSBBase property), 478
- dut\_type (pymeasure.instruments.temptronic.ATSBBase property), 478
- duty\_cycle (pymeasure.instruments.hp.HP8116A property), 249
- dwelt\_time (pymeasure.instruments.agilent.Agilent8257D property), 153
- dynamic\_temperature\_setpoint (pymeasure.instruments.temptronic.ATSBBase property), 478
- ## E
- EC0560 (class in pymeasure.instruments.temptronic), 484
- elapsed\_time (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 257
- emergency\_off() (pymeasure.instruments.eurotest.EurotestHPP120256 method), 232
- emission (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart property), 496
- emission\_enabled (pymeasure.instruments.ipgphotonics.yar.YAR property), 286
- emission\_enabled (pymeasure.instruments.novanta.Fpu60 property), 393
- emit() (pymeasure.display.log.LogHandler method), 87
- emit() (pymeasure.experiment.workers.Worker method), 79
- enable (pymeasure.instruments.edwards.Nxds property), 231
- enable() (pymeasure.instruments.agilent.Agilent8257D method), 153
- enable() (pymeasure.instruments.agilent.agilentB1500.SMU method), 188
- enable() (pymeasure.instruments.anritsu.AnritsuMG3692C method), 208
- enable() (pymeasure.instruments.danfysik.Danfysik8500 method), 227
- enable() (pymeasure.instruments.deltaelektronika.SM7045D method), 230
- enable() (pymeasure.instruments.keysight.KeysightN5767A method), 341
- enable() (pymeasure.instruments.newport.ESP300 method), 377
- enable() (pymeasure.instruments.newport.esp300.Axis method), 377
- enable() (pymeasure.instruments.parker.ParkerGV6 method), 403
- enable\_4W\_mode() (pymeasure.instruments.siglentechnologies.siglent\_spdbase.SPDSingleC method), 425
- enable\_air\_flow (pymeasure.instruments.temptronic.ATSBBase property), 478
- enable\_amplitude\_modulation() (pymeasure.instruments.agilent.Agilent8257D method), 153
- enable\_averaging() (pymeasure.instruments.agilent.Agilent8722ES method), 155
- enable\_bias() (pymeasure.instruments.srs.SR570 method), 441
- enable\_continous() (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart method), 496
- enable\_control() (pymeasure.instruments.oxfordinstruments.IPS120\_10 method), 400
- enable\_filter() (pymeasure.instruments.keithley.Keithley2000 method), 290
- enable\_local\_interface() (pymeasure.instruments.siglentechnologies.siglent\_spdbase.SPDBase method), 424
- enable\_low\_freq\_out() (pymeasure.instruments.agilent.Agilent8257D method), 153
- enable\_offset\_current() (pymeasure.instruments.srs.SR570 method), 441
- enable\_output() (pymeasure-

`sure.instruments.siglenttechnologies.siglent_spdbase.SPDCChannel` property), 203  
`method`), 425  
`enable_persistent_mode()` (`pymea-`  
`sure.instruments.oxfordinstruments.IPS120_10`  
`method`), 400  
`enable_persistent_switch()` (`pymea-`  
`sure.instruments.ami.AMI430` `method`), 201  
`enable_pulse_modulation()` (`pymea-`  
`sure.instruments.agilent.Agilent8257D`  
`method`), 153  
`enable_pulsing()` (`pymea-`  
`sure.instruments.toptica.ibeamsmart.IBeamSmart`  
`method`), 496  
`enable_reference()` (`pymea-`  
`sure.instruments.keithley.Keithley2000`  
`method`), 290  
`enable_rf()` (`pymea.instruments.anapico.APSIN12G` `end_of_all_cycles()` (`pymea-`  
`method`), 205 `sure.instruments.temptronic.ATSBBase` `method`),  
`enable_source()` (`pymea-` 478  
`sure.instruments.advantest.advantestR624X.AdvantestR624X` `end_of_one_cycle()` (`pymea-`  
`method`), 124 `sure.instruments.temptronic.ATSBBase` `method`),  
`enable_source()` (`pymea-` 478  
`sure.instruments.advantest.advantestR624X.SMUChannel` `END_OF_SWEEP` (`pymea-`  
`method`), 140 `sure.instruments.hp.hp856Xx.StatusRegister`  
`enable_source()` (`pymea-` `attribute`), 279  
`sure.instruments.keithley.Keithley2400` `end_of_test()` (`pymea-`  
`method`), 302 `sure.instruments.temptronic.ATSBBase` `method`),  
`enable_source()` (`pymea-` 478  
`sure.instruments.keithley.Keithley2450` `end_sequence()` (`pymea-`  
`method`), 310 `sure.instruments.advantest.advantestR624X.AdvantestR624X`  
`enable_source()` (`pymea-` `method`), 125  
`sure.instruments.keithley.Keithley6221` `energy` (`pymea.instruments.thorlabs.ThorlabsPM100USB`  
`method`), 321 `property`), 489  
`enable_source()` (`pymea-` `enter_cycle()` (`pymea-`  
`sure.instruments.keithley.Keithley6517B` `sure.instruments.temptronic.ATSBBase` `method`),  
`method`), 327 479  
`enable_source()` (`pymea-` `enter_ramp()` (`pymea-`  
`sure.instruments.yokogawa.Yokogawa7651` `sure.instruments.temptronic.ATSBBase` `method`),  
`method`), 500 479  
`enable_timer()` (`pymea-` `entry_level_strategy` (`pymea-`  
`sure.instruments.siglenttechnologies.siglent_spdbase.SPDCChannel` `sure.instruments.activetechnologies.AWG401x_AWG`  
`method`), 425 `property`), 118  
`enabled` (`pymea.instruments.activetechnologies.AWG401x_AWG` `abi_456` (`pymea.instruments.prologix.PrologixAdapter` `property`), 58  
`property`), 117 `eos` (`pymea.instruments.prologix.PrologixAdapter` `property`), 58  
`enabled` (`pymea.instruments.activetechnologies.AWG401x_AWG` `channel_status` (`pymea.instruments.srs.SR830` `prop-`  
`property`), 118 `erty`), 443  
`enabled` (`pymea.instruments.aja.DCXS` `property`), `error` (`pymea.instruments.keithley.Keithley2260B`  
197 `property`), 296  
`enabled` (`pymea.instruments.newport.esp300.Axis` `error` (`pymea.instruments.keithley.Keithley2400`  
`property`), 377 `property`), 302  
`enabled` (`pymea.instruments.toptica.ibeamsmart.DriverChannel` `error` (`pymea.instruments.keithley.Keithley2450`  
`property`), 497 `property`), 310  
`encoder_autocorrect` (`pymea-` `error` (`pymea.instruments.keithley.Keithley2600`  
`sure.instruments.anaheimautomation.DPSeriesMotorController` `property`), 333

error (*pymeasure.instruments.keithley.Keithley2700* property), 317  
 error (*pymeasure.instruments.keithley.Keithley6221* property), 321  
 error (*pymeasure.instruments.keithley.Keithley6517B* property), 327  
 error (*pymeasure.instruments.newport.ESP300* property), 377  
 error (*pymeasure.instruments.siglenttechnologies.siglent\_spdbase.SPDBase* property), 424  
 error (*pymeasure.instruments.texio.TexioPSW360L30* property), 485  
 error\_code (*pymeasure.instruments.temptronic.ATSBASE* property), 479  
 ERROR\_PRESENT (*pymeasure.instruments.hp.hp856Xx.StatusRegister* attribute), 279  
 error\_reg (*pymeasure.instruments.anaheimautomation.DPSeriesMosaik* property), 203  
 error\_register (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 131  
 error\_status (*pymeasure.instruments.hp.HP3478A* property), 246  
 error\_status() (*pymeasure.instruments.temptronic.ATSBASE* method), 479  
 ErrorCode (class in *pymeasure.instruments.hp.hp856Xx*), 278  
 ErrorCode (class in *pymeasure.instruments.temptronic.temptronic\_base*), 483  
 errors (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 253  
 errors (*pymeasure.instruments.newport.ESP300* property), 377  
 ese2 (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209  
 ESP300 (class in *pymeasure.instruments.newport*), 377  
 esr2 (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209  
 EstimatorThread (class in *pymeasure.display.widgets.estimator\_widget*), 90  
 EstimatorWidget (class in *pymeasure.display.widgets.estimator\_widget*), 90  
 EurotestHPP120256 (class in *pymeasure.instruments.eurotest*), 231  
 EurotestHPP120256.EurotestHPP120256Status (class in *pymeasure.instruments.eurotest*), 232  
 evaluate\_metadata() (*pymeasure.experiment.procedure.Procedure* method), 73  
 event\_reg (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 410  
 event\_status\_enable (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 130  
 event\_status\_enable\_bits (*pymeasure.instruments.anritsu.AnritsuMS464xB* property), 217  
 event\_status\_register (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 131  
 exchange\_traces() (*pymeasure.instruments.hp.hp856Xx.HP856Xx* method), 265  
 exec() (*pymeasure.display.console.ManagedConsole* method), 84  
 execute() (*pymeasure.experiment.procedure.Procedure* method), 73  
 expand\_channel\_string() (*pymeasure.instruments.ni.virtualbench.VirtualBench* method), 391  
 expected\_protocol() (in module *pymeasure.test*), 66  
 Exporter (class in *pymeasure.display.manager*), 88  
 Experiment (class in *pymeasure.experiment.experiment*), 71  
 ExperimentQueue (class in *pymeasure.display.manager*), 88  
 export\_signal() (*pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput* method), 379  
 ExpressionValidator (class in *pymeasure.display.widgets.sequencer\_widget*), 92  
 ext\_ref\_base\_unit (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 410  
 ext\_ref\_extension (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 410  
 ext\_trig\_out (*pymeasure.instruments.agilent.Agilent33500* property), 174  
 ext\_vid\_connector (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 410  
 External (*pymeasure.instruments.hp.hp856Xx.FrequencyReference* attribute), 278  
 External (*pymeasure.instruments.hp.hp856Xx.MixerMode* attribute), 277  
 External (*pymeasure.instruments.hp.hp856Xx.SourceLevelingControlMode* attribute), 279  
 External (*pymeasure.instruments.hp.hp856Xx.TriggerMode* attribute), 280  
 external\_arming\_start\_slope (*pymeasure.instruments.pendulum.cnt91.CNT91* property), 405

`external_current` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`external_modulation_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 410

`external_modulation_power` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 411

`external_modulation_source` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 411

`external_start_arwing_source` (`pymeasure.instruments.pendulum.cnt91.CNT91` property), 405

`external_trigger_delay` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218

`external_trigger_edge` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218

`external_trigger_handshake` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218

`external_trigger_type` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218

`extfrequency` (`pymeasure.instruments.srs.SR860` property), 447

`extract_value()` (`pymeasure.instruments.Keithley.Keithley6517B` static method), 327

**F**

`factory_reset()` (`pymeasure.instruments.keysight.KeysightDSOX1102G` method), 340

`FakeAdapter` (class in `pymeasure.adapters`), 67

`FakeInstrument` (class in `pymeasure.instruments.fakes`), 112

`fast_mode` (`pymeasure.instruments.lakeshore.LakeShore425` property), 356

`fast_mode_enabled` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 133

`fault_code` (`pymeasure.instruments.aja.DCXS` property), 197

`Fav` (`pymeasure.instruments.hp.hp856Xx.SweepOut` attribute), 280

`fet` (`pymeasure.instruments.signalrecovery.DSP7225` property), 430

`fet` (`pymeasure.instruments.signalrecovery.DSP7265` property), 436

`fetch_control` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_density` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_eirpower` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_eirpower_data` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_eirpower_max` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_emf` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 211

`fetch_emf_meter` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_emf_meter_sample` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_interference_power` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_mimo_antenas` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_ocupied_bw` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_ota_mapping` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_pan` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_pbch_constellation` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_pci` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_pdsch` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_pdsch_constellation` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_peak` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_power` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_rrm` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

`fetch_scan` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 212

property), 212

fetch\_semaphore (pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property), 212

fetch\_ssb (pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property), 212

fetch\_sync\_evm (pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property), 212

fetch\_sync\_power (pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property), 212

fetch\_tae (pymea-  
sure.instruments.anritsu.AnritsuMS2090A  
property), 213

field (pymea-  
sure.instruments.ami.AMI430 property),  
201

field (pymea-  
sure.instruments.fwbell.FWBell5080 prop-  
erty), 235

field (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 356

field (pymea-  
sure.instruments.lakeshore.LakeShore425  
property), 359

field (pymea-  
sure.instruments.oxfordinstruments.IPS120\_10  
property), 400

field\_mode (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 356

field\_multiplier (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 356

field\_range (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 356

field\_range\_raw (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 357

field\_raw (pymea-  
sure.instruments.lakeshore.LakeShore421  
property), 357

field\_setpoint (pymea-  
sure.instruments.oxfordinstruments.IPS120\_10  
property), 400

fields() (pymea-  
sure.instruments.fwbell.FWBell5080  
method), 235

filer\_synchronous (pymea-  
sure.instruments.srs.SR860  
property), 447

filter (pymea-  
sure.instruments.agilent.agilentB1500.SMU  
property), 188

filter (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator  
property), 383

filter\_advanced (pymea-  
sure.instruments.srs.SR860  
property), 447

filter\_count (pymea-  
sure.instruments.keithley.Keithley2400 prop-  
erty), 302

filter\_slope (pymea-  
sure.instruments.srs.SR830 prop-  
erty), 444

filter\_slope (pymea-  
sure.instruments.srs.SR860 prop-  
erty), 447

filter\_state (pymea-  
sure.instruments.keithley.Keithley2400 prop-  
erty), 302

filter\_synchronous (pymea-  
sure.instruments.srs.SR830 property), 444

filter\_type (pymea-  
sure.instruments.keithley.Keithley2400  
property), 302

filter\_type (pymea-  
sure.instruments.srs.SR570 prop-  
erty), 441

find\_img\_index() (pymea-  
sure.display.curves.ResultsImage  
method),  
85

firmware (pymea-  
sure.instruments.ipgphotonics.yar.YAR  
property), 286

firmware\_revision (pymea-  
sure.instruments.hp.hp856Xx.HP856Xx at-  
tribute), 257

firmware\_version (pymea-  
sure.instruments.tcpowerconversion.CXN  
property), 453

firmware\_version (pymea-  
sure.instruments.thyracont.smartline\_v2.SmartlineV2  
property), 493

flags() (pymea-  
sure.display.widgets.sequencer\_widget.SequencerTreeMod  
method), 93

FlatTop (pymea-  
sure.instruments.hp.hp856Xx.WindowType  
attribute), 280

FloatParameter (class in pymea-  
sure.experiment.parameters), 75

Fluke7341 (class in pymea-  
sure.instruments.fluke), 234

flush\_read\_buffer() (pymea-  
sure.adapters.Adapter  
method), 48

flush\_read\_buffer() (pymea-  
sure.adapters.FakeAdapter method), 67

flush\_read\_buffer() (pymea-  
sure.adapters.PrologixAdapter  
method),  
58

flush\_read\_buffer() (pymea-  
sure.adapters.ProtocolAdapter  
method),  
67

flush\_read\_buffer() (pymea-  
sure.adapters.SerialAdapter method), 54

flush\_read\_buffer() (pymea-  
sure.adapters.TelnetAdapter method), 64

flush\_read\_buffer() (pymea-  
sure.adapters.VISAAdapter method), 51

flush\_read\_buffer() (pymea-  
sure.adapters.VXII1Adapter method), 61

fm\_deviation (pymea-  
sure.instruments.hp.HP8657B  
property), 282

fm\_source (pymea-  
sure.instruments.hp.HP8657B prop-  
erty), 282

`foldback_delay` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` property), 457

`foldback_delay` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` property), 462

`foldback_enabled` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` property), 457

`foldback_enabled` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` property), 462

`foldback_reset()` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` method), 457

`foldback_reset()` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` method), 462

`force()` (`pymeasure.instruments.agilent.agilentB1500.SMU` method), 189

`force_gnd()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 184

`force_gnd()` (`pymeasure.instruments.agilent.agilentB1500.SMU` method), 188

`FORCE_SIDE` (`pymeasure.instruments.agilent.agilentB1500.MeasOpMode` attribute), 194

`force_trigger()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope` method), 387

`format` (`pymeasure.instruments.pendulum.cnt91.CNT91` property), 405

`format()` (`pymeasure.display.widgets.log_widget.HTMLFormatter` method), 91

`format()` (`pymeasure.experiment.results.CSVFormatter` method), 79

`format()` (`pymeasure.experiment.results.Results` method), 80

`Fpu60` (class in `pymeasure.instruments.novanta`), 392

`frame` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113

`frame_format` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113

`frame_height` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113

`frame_width` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113

`Free` (`pymeasure.instruments.hp.hp856Xx.TriggerMode` attribute), 280

`freq_center` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` property), 421

`freq_span` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` property), 421

`freq_start` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` property), 421

`freq_stop` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` property), 421

`freq_sweep()` (`pymeasure.instruments.agilent.AgilentE4980` method), 157

`frequencies` (`pymeasure.instruments.agilent.Agilent8722ES` property), 155

`frequencies` (`pymeasure.instruments.agilent.AgilentE4408B` property), 156

`frequency` (`pymeasure.instruments.activetechnologies.AWG401x.ChannelA` property), 120

`frequency` (`pymeasure.instruments.agilent.Agilent33220A` property), 171

`frequency` (`pymeasure.instruments.agilent.Agilent33500` property), 174

`frequency` (`pymeasure.instruments.agilent.agilent33500.Agilent33500ChannelA` property), 177

`frequency` (`pymeasure.instruments.agilent.Agilent33521A` property), 176

`frequency` (`pymeasure.instruments.agilent.Agilent34450A` property), 161

`frequency` (`pymeasure.instruments.agilent.Agilent8257D` property), 153

`frequency` (`pymeasure.instruments.agilent.AgilentE4980` property), 157

`frequency` (`pymeasure.instruments.ametek.Ametek7270` property), 199

`frequency` (`pymeasure.instruments.anapico.APSIN12G` property), 205

`frequency` (`pymeasure.instruments.andeenhagerling.AH2700A` property), 206

`frequency` (`pymeasure.instruments.anritsu.AnritsuMG3692C` property), 208

`frequency` (`pymeasure.instruments.attocube.anc300.Axis` property), 225

`frequency` (`pymeasure.instruments.hp.HP33120A` property), 240

`frequency` (`pymeasure.instruments.hp.HP8116A` property), 249

`Frequency` (`pymeasure.instruments.hp.hp856Xx.DemodulationMode` attribute), 278

`frequency` (`pymeasure.instruments.hp.HP8657B` property), 282

`frequency` (`pymeasure.instruments.keithley.Keithley2000` property), 290

`frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 411

`frequency` (`pymeasure.instruments.rohdeschwarz.sfm.Sound_ChannelA` property), 417

`frequency` (`pymeasure.instruments.signalrecovery.DSP7225` property), 430

`frequency` (`pymeasure.instruments.signalrecovery.DSP7265` property), 430

- property*), 436
- `frequency` (`pymeasure.instruments.srs.SR510` *property*), 440
- `frequency` (`pymeasure.instruments.srs.SR830` *property*), 444
- `frequency` (`pymeasure.instruments.srs.SR860` *property*), 447
- `frequency` (`pymeasure.instruments.tcpowerconversion.CXN` *property*), 453
- `frequency` (`pymeasure.instruments.teledyne.teledyneT3AFG.SignalChannel` *property*), 467
- `frequency_aperture` (`pymeasure.instruments.keithley.Keithley2000` *property*), 290
- `frequency_aperture` (`pymeasure.instruments.agilent.Agilent34450A` *property*), 161
- `frequency_center` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_center` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` *property*), 221
- `frequency_counter_mode_enabled` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` *attribute*), 256
- `frequency_counter_resolution` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` *attribute*), 256
- `frequency_current_auto_range` (`pymeasure.instruments.agilent.Agilent34450A` *property*), 161
- `frequency_current_range` (`pymeasure.instruments.agilent.Agilent34450A` *property*), 161
- `frequency_CW` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` *property*), 221
- `frequency_digits` (`pymeasure.instruments.keithley.Keithley2000` *property*), 290
- `frequency_display_enabled` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` *attribute*), 260
- `frequency_max` (`pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG` *property*), 120
- `frequency_min` (`pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG` *property*), 120
- `frequency_mode` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` *property*), 411
- `frequency_offset` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_offset` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` *attribute*), 259
- `frequency_points` (`pymeasure.instruments.agilent.AgilentE4408B` *property*), 156
- `frequency_reference` (`pymeasure.instruments.keithley.Keithley2000` *property*), 290
- `frequency_reference_source` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` *attribute*), 260
- `frequency_span` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_span` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` *property*), 221
- `frequency_span_full` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_span_last` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_start` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_start` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` *property*), 221
- `frequency_step` (`pymeasure.instruments.agilent.AgilentE4408B` *property*), 156
- `frequency_step` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_stop` (`pymeasure.instruments.anritsu.AnritsuMS2090A` *property*), 213
- `frequency_stop` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` *property*), 221
- `frequency_threshold` (`pymeasure.instruments.keithley.Keithley2000` *property*), 290
- `frequency_voltage_auto_range` (`pymeasure.instruments.agilent.Agilent34450A` *property*), 162
- `frequency_voltage_range` (`pymeasure.instruments.agilent.Agilent34450A` *property*), 162
- `frequencypreset1` (`pymeasure.instruments.srs.SR860`

- property), 447
- frequencypreset2 (pymeasure.instruments.srs.SR860 property), 447
- frequencypreset3 (pymeasure.instruments.srs.SR860 property), 447
- frequencypreset4 (pymeasure.instruments.srs.SR860 property), 448
- FrequencyReference (class in pymeasure.instruments.hp.hp856Xx), 278
- front\_blanked (pymeasure.instruments.srs.SR570 property), 441
- front\_panel (pymeasure.instruments.srs.SR860 property), 448
- front\_panel\_brightness (pymeasure.instruments.lakeshore.LakeShore421 property), 357
- front\_panel\_display (pymeasure.instruments.oxfordinstruments.ITC503 property), 396
- front\_panel\_locked (pymeasure.instruments.lakeshore.LakeShore421 property), 357
- FSL (class in pymeasure.instruments.rohdeschwarz.fsl), 420
- function\_ (pymeasure.instruments.hp.HP34401A property), 241
- fw\_version (pymeasure.instruments.siglentechnologies.siglent\_spdbsense.SPDbsense static property), 424
- FWBell15080 (class in pymeasure.instruments.fwbell), 234
- ## G
- gain (pymeasure.instruments.signalrecovery.DSP7225 property), 430
- gain (pymeasure.instruments.signalrecovery.DSP7265 property), 436
- gain\_mode (pymeasure.instruments.srs.SR570 property), 441
- gasflow (pymeasure.instruments.oxfordinstruments.ITC503 property), 396
- gasflow\_configuration\_parameter (pymeasure.instruments.oxfordinstruments.ITC503 property), 396
- gasflow\_control\_status (pymeasure.instruments.oxfordinstruments.ITC503 property), 396
- gate\_time (pymeasure.instruments.hp.HP34401A property), 241
- gen\_measurement() (pymeasure.experiment.procedure.Procedure method), 73
- GeneralError (class in pymeasure.instruments.newport.esp300), 378
- Generator (class in pymeasure.generator), 69
- get() (pymeasure.instruments.agilent.agilentB1500.CustomIntEnum class method), 193
- get\_alarm\_status() (pymeasure.instruments.lakeshore.LakeShore211 method), 352
- get\_array() (in module pymeasure.experiment.experiment), 72
- get\_array\_steps() (in module pymeasure.experiment.experiment), 72
- get\_array\_zero() (in module pymeasure.experiment.experiment), 72
- get\_buffer() (pymeasure.instruments.signalrecovery.DSP7225 method), 430
- get\_buffer() (pymeasure.instruments.signalrecovery.DSP7265 method), 437
- get\_buffer() (pymeasure.instruments.srs.SR830 method), 444
- get\_calibration\_information() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 392
- get\_channel\_pairs() (pymeasure.instruments.common\_base.CommonBase static method), 107
- get\_channel\_pairs() (pymeasure.instruments.siglent.spdbsense.SPDbsense static method), 337
- get\_channel\_pairs() (pymeasure.instruments.keysight.KeysightE36312A static method), 348
- get\_channel\_pairs() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 static method), 367
- get\_channels() (pymeasure.instruments.common\_base.CommonBase static method), 107
- get\_channels() (pymeasure.instruments.keysight.KeysightE36312A static method), 348
- get\_channels() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 static method), 367
- get\_data() (pymeasure.instruments.agilent.agilent4156.Agilent4156 method), 165
- get\_estimates() (pymeasure.display.widgets.estimator\_widget.EstimatorWidget method), 90
- get\_estimates() (pymeasure.experiment.procedure.Procedure method), 73

`get_filename()` (*pymeasure.display.console.ManagedConsole method*), 84  
`get_library_version()` (*pymeasure.instruments.ni.virtualbench.VirtualBench method*), 392  
`get_noise_bandwidth` (*pymeasure.instruments.srs.SR860 property*), 448  
`get_operation_times()` (*pymeasure.instruments.novanta.Fpu60 method*), 393  
`get_power_bandwidth()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx method*), 262  
`get_procedure()` (*pymeasure.display.widgets.inputs\_widget.InputsWidget method*), 91  
`get_relay_mode()` (*pymeasure.instruments.lakeshore.LakeShore211 method*), 353  
`get_scaling()` (*pymeasure.instruments.srs.SR830 method*), 444  
`get_sensor_transition()` (*pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 method*), 493  
`get_signal_strength_indicator` (*pymeasure.instruments.srs.SR860 property*), 448  
`get_state_of_channels()` (*pymeasure.instruments.keithley.Keithley2700 method*), 317  
`get_trace_data_a()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx method*), 263  
`get_trace_data_b()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx method*), 263  
`get_wl_data()` (*pymeasure.instruments.keysight.KeysightN7776C method*), 342  
`getAI()` (*in module pymeasure.instruments.comedi*), 116  
`getAO()` (*in module pymeasure.instruments.comedi*), 116  
`gettimebase` (*pymeasure.instruments.srs.SR860 property*), 448  
`gpib()` (*pymeasure.adapters.PrologixAdapter method*), 58  
`gpib_address` (*pymeasure.instruments.rohdeschwarz.sfm.SFM property*), 411  
`gpib_read_timeout` (*pymeasure.adapters.PrologixAdapter property*), 58  
`GPIB_trigger()` (*pymeasure.instruments.hp.HP8116A method*), 248  
`GPIB_trigger()` (*pymeasure.instruments.hp.HPLegacyInstrument method*), 283  
`gps` (*pymeasure.instruments.anritsu.AnritsuMS2090A property*), 213  
`gps_all` (*pymeasure.instruments.anritsu.AnritsuMS2090A property*), 213  
`gps_full` (*pymeasure.instruments.anritsu.AnritsuMS2090A property*), 213  
`gps_last` (*pymeasure.instruments.anritsu.AnritsuMS2090A property*), 213  
`graticule_enabled` (*pymeasure.instruments.hp.hp856Xx.HP856Xx attribute*), 257  
`grid_display` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204 property*), 367  
`ground_all()` (*pymeasure.instruments.attocube.anc300.ANC300Controller method*), 224

## H

`handle_abort()` (*pymeasure.experiment.workers.Worker method*), 79  
`handle_deprecated_host_arg()` (*pymeasure.instruments.attocube.anc300.ANC300Controller method*), 224  
`handle_error()` (*pymeasure.experiment.workers.Worker method*), 79  
`Hanning` (*pymeasure.instruments.hp.hp856Xx.WindowType attribute*), 281  
`harmonic` (*pymeasure.instruments.ametek.Ametek7270 property*), 199  
`harmonic` (*pymeasure.instruments.signalrecovery.DSP7225 property*), 430  
`harmonic` (*pymeasure.instruments.signalrecovery.DSP7265 property*), 437  
`harmonic` (*pymeasure.instruments.srs.SR830 property*), 444  
`harmonic` (*pymeasure.instruments.srs.SR860 property*), 448  
`harmonic_number_lock` (*pymeasure.instruments.hp.HP8561B property*), 275  
`harmonicdual` (*pymeasure.instruments.srs.SR860 property*), 448  
`has_amplitude_modulation` (*pymeasure.instruments.agilent.Agilent8257D property*), 154  
`has_modulation` (*pymeasure.instruments.agilent.Agilent8257D property*), 154

[has\\_next\(\)](#) (`pymeasure.display.manager.ExperimentQueue` method), 88  
[has\\_persistent\\_switch\\_enabled\(\)](#) (`pymeasure.instruments.ami.AMI430` method), 201  
[has\\_pulse\\_modulation](#) (`pymeasure.instruments.agilent.Agilent8257D` property), 154  
[haversine\\_enabled](#) (`pymeasure.instruments.hp.HP8116A` property), 249  
[head](#) (`pymeasure.instruments.temptronic.ATSB` property), 479  
[head\\_temperature](#) (`pymeasure.instruments.novanta.Fpu60` property), 393  
[header\(\)](#) (`pymeasure.experiment.results.Results` method), 80  
[headerData\(\)](#) (`pymeasure.display.widgets.sequencer_widget.SequencerWidget` method), 93  
[headerData\(\)](#) (`pymeasure.display.widgets.table_widget.PandasModelBase` method), 96  
[heater](#) (`pymeasure.instruments.oxfordinstruments.ITC503` property), 396  
[heater\\_gas\\_mode](#) (`pymeasure.instruments.oxfordinstruments.ITC503` property), 396  
[heater\\_voltage](#) (`pymeasure.instruments.oxfordinstruments.ITC503` property), 396  
[High](#) (`pymeasure.instruments.hp.hp856Xx.PeakSearchMode` attribute), 279  
[high\\_freq](#) (`pymeasure.instruments.srs.SR570` property), 441  
[high\\_frequency\\_resolution](#) (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 411  
[high\\_level](#) (`pymeasure.instruments.hp.HP8116A` property), 249  
[HMP4040](#) (class in `pymeasure.instruments.rohdeschwarz.hmp`), 421  
[hold\(\)](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 256  
[hold\\_function](#) (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 222  
[hold\\_function\\_all\\_channels](#) (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218  
[hold\\_time](#) (`pymeasure.instruments.agilent.agilent4156A.Agilent4156A` property), 165  
[home\(\)](#) (`pymeasure.instruments.anaheimautomation.DPSeriesMotorController` method), 203  
[home\(\)](#) (`pymeasure.instruments.newport.esp300.Axis` method), 377  
[horizontal\\_time\\_div](#) (`pymeasure.instruments.srs.SR860` property), 448  
[hotcathode](#) (`pymeasure.instruments.thyracont.smartline_v2.VSH` attribute), 494  
[HP33120A](#) (class in `pymeasure.instruments.hp`), 239  
[HP3437A](#) (class in `pymeasure.instruments.hp`), 243  
[HP3437A.SRQ](#) (class in `pymeasure.instruments.hp`), 243  
[HP34401A](#) (class in `pymeasure.instruments.hp`), 240  
[HP3478A](#) (class in `pymeasure.instruments.hp`), 245  
[HP3478A.ERRORS](#) (class in `pymeasure.instruments.hp`), 245  
[HP6632A](#) (class in `pymeasure.instruments.hp`), 284  
[HP6632A.ERRORS](#) (class in `pymeasure.instruments.hp`), 284  
[HP6632A.ST\\_ERRORS](#) (class in `pymeasure.instruments.hp`), 284  
[HP6632A](#) (class in `pymeasure.instruments.hp`), 286  
[HP6634A](#) (class in `pymeasure.instruments.hp`), 286  
[HP8116A](#) (class in `pymeasure.instruments.hp`), 248  
[HP8116A.Digit](#) (class in `pymeasure.instruments.hp`), 248  
[HP8116A.Direction](#) (class in `pymeasure.instruments.hp`), 248  
[HP8560A](#) (class in `pymeasure.instruments.hp`), 273  
[HP8561B](#) (class in `pymeasure.instruments.hp`), 275  
[HP8567B](#) (class in `pymeasure.instruments.hp`), 281  
[HP8567B.Modulation](#) (class in `pymeasure.instruments.hp`), 281  
[HPLegacyInstrument](#) (class in `pymeasure.instruments.hp`), 283  
[HRADC](#) (`pymeasure.instruments.agilent.agilentB1500.ADCTYPE` attribute), 193  
[HSADC](#) (`pymeasure.instruments.agilent.agilentB1500.ADCTYPE` attribute), 193  
[HSADC\\_PULSED](#) (`pymeasure.instruments.agilent.agilentB1500.ADCTYPE` attribute), 193  
[HTMLFormatter](#) (class in `pymeasure.display.widgets.log_widget`), 91  
**I**  
[IBeamSmart](#) (class in `pymeasure.instruments.toptica.ibeamsmart`), 495  
[id](#) (`pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG` property), 122  
[id](#) (`pymeasure.instruments.aja.DCXS` property), 197  
[id](#) (`pymeasure.instruments.ametek.Ametek7270` property), 199  
[id](#) (`pymeasure.instruments.andeenhagerling.AH2700A` property), 206  
[id](#) (`pymeasure.instruments.danfysik.Danfysik8500` property), 228

- `id (pymeasure.instruments.eurotest.EurotestHPP120256 property)`, 233
- `id (pymeasure.instruments.fluke.Fluke7341 property)`, 234
- `id (pymeasure.instruments.fwbell.FWBell5080 property)`, 235
- `id (pymeasure.instruments.heidenhain.ND287 property)`, 237
- `id (pymeasure.instruments.hp.HP6632A property)`, 285
- `id (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute)`, 257
- `id (pymeasure.instruments.hp.HP8657B attribute)`, 283
- `id (pymeasure.instruments.Instrument property)`, 110
- `id (pymeasure.instruments.ipgphotonics.yar.YAR property)`, 286
- `id (pymeasure.instruments.keithley.Keithley2000 property)`, 290
- `id (pymeasure.instruments.keithley.Keithley2200 property)`, 337
- `id (pymeasure.instruments.keithley.Keithley2260B property)`, 296
- `id (pymeasure.instruments.keithley.Keithley2306 property)`, 298
- `id (pymeasure.instruments.keithley.Keithley2400 property)`, 302
- `id (pymeasure.instruments.keithley.Keithley2450 property)`, 310
- `id (pymeasure.instruments.keithley.Keithley2600 property)`, 333
- `id (pymeasure.instruments.keithley.Keithley2700 property)`, 317
- `id (pymeasure.instruments.keithley.Keithley2750 property)`, 331
- `id (pymeasure.instruments.keithley.Keithley6221 property)`, 321
- `id (pymeasure.instruments.keithley.Keithley6517B property)`, 327
- `id (pymeasure.instruments.keysight.KeysightE36312A property)`, 348
- `id (pymeasure.instruments.lecroy.LeCroyT3DSO1204 property)`, 367
- `id (pymeasure.instruments.signalrecovery.DSP7225 property)`, 430
- `id (pymeasure.instruments.signalrecovery.DSP7265 property)`, 437
- `id (pymeasure.instruments.tcpowerconversion.CXN property)`, 453
- `id (pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38 property)`, 457
- `id (pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65 property)`, 462
- `id (pymeasure.instruments.teledyne.TeledyneT3AFG property)`, 466
- `id (pymeasure.instruments.texio.TexioPSW360L30 property)`, 486
- `id (pymeasure.instruments.thermotron.Thermotron3800 property)`, 487
- `id (pymeasure.instruments.velleman.VellemanK8090 attribute)`, 498
- `id (pymeasure.instruments.yokogawa.Yokogawa7651 property)`, 500
- `ImageFrame (class in pymeasure.display.widgets.image_frame)`, 90
- `ImageWidget (class in pymeasure.display.widgets.image_widget)`, 90
- `imode (pymeasure.instruments.signalrecovery.DSP7225 property)`, 430
- `imode (pymeasure.instruments.signalrecovery.DSP7265 property)`, 437
- `impedance (pymeasure.instruments.agilent.AgilentE4980 property)`, 157
- `index() (pymeasure.display.widgets.sequencer_widget.SequencerTreeMod method)`, 93
- `information (pymeasure.instruments.hcp.TC038 property)`, 238
- `init_all_sweep() (pymeasure.instruments.anritsu.AnritsuMS2090A method)`, 213
- `init_continuous (pymeasure.instruments.anritsu.AnritsuMS2090A property)`, 213
- `init_curve_buffer() (pymeasure.instruments.signalrecovery.DSP7225 method)`, 430
- `init_curve_buffer() (pymeasure.instruments.signalrecovery.DSP7265 method)`, 437
- `init_sequence() (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X method)`, 125
- `init_spa_self (pymeasure.instruments.anritsu.AnritsuMS2090A property)`, 213
- `init_sweep() (pymeasure.instruments.anritsu.AnritsuMS2090A method)`, 213
- `init_trigger() (pymeasure.instruments.hp.HP34401A method)`, 241
- `italize_oven() (pymeasure.instruments.thermotron.Thermotron3800 method)`, 488
- `initialize_all_smus() (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 method)`, 184
- `initialize_smu() (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 method)`, 184

`Input` (class in `pymeasure.display.inputs`), 85  
`input_0` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 353  
`input_A` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 353  
`input_A` (`pymeasure.instruments.lakeshore.LakeShore331` attribute), 355  
`input_B` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 353  
`input_B` (`pymeasure.instruments.lakeshore.LakeShore331` attribute), 355  
`input_C1` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_C2` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_C3` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_C4` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_C5` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_config` (`pymeasure.instruments.srs.SR830` property), 444  
`input_coupling` (`pymeasure.instruments.srs.SR830` property), 444  
`input_coupling` (`pymeasure.instruments.srs.SR860` property), 448  
`input_current_gain` (`pymeasure.instruments.srs.SR860` property), 448  
`input_D1` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_D2` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_D3` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_D4` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_D5` (`pymeasure.instruments.lakeshore.LakeShore224` attribute), 354  
`input_grounding` (`pymeasure.instruments.srs.SR830` property), 444  
`input_notch_config` (`pymeasure.instruments.srs.SR830` property), 444  
`input_range` (`pymeasure.instruments.srs.SR860` property), 448  
`input_shields` (`pymeasure.instruments.srs.SR860` property), 448  
`input_signal` (`pymeasure.instruments.srs.SR860` property), 448  
`input_voltage_mode` (`pymeasure.instruments.srs.SR860` property), 448  
`InputsWidget` (class in `pymeasure.display.widgets.inputs_widget`), 91  
`insert_id()` (`pymeasure.instruments.advantest.advantestR624X.SMUCChannel` method), 132  
`insert_id()` (`pymeasure.instruments.attocube.anc300.Axis` method), 225  
`insert_id()` (`pymeasure.instruments.Channel` method), 112  
`insert_id()` (`pymeasure.instruments.keithley.keithley2200.PSChannel` method), 338  
`insert_id()` (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope` method), 474  
`instant_voltage_1` (`pymeasure.instruments.razorbill.razorbillRP100` property), 406  
`instant_voltage_2` (`pymeasure.instruments.razorbill.razorbillRP100` property), 406  
`instantiate()` (`pymeasure.generator.Generator` method), 69  
`Instrument` (class in `pymeasure.instruments`), 109  
`IntegerInput` (class in `pymeasure.display.inputs`), 85  
`IntegerParameter` (class in `pymeasure.experiment.parameters`), 75  
`integral_action_time` (`pymeasure.instruments.oxfordinstruments.ITC503` property), 396  
`integration_time` (`pymeasure.instruments.agilent.agilent4156.Agilent4156` property), 165  
`intensity` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 367  
`intensity` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 469  
`interlock_enabled` (`pymeasure.instruments.novanta.Fpu60` property), 393  
`Internal` (`pymeasure.instruments.hp.hp856Xx.FrequencyReference` attribute), 278  
`Internal` (`pymeasure.instruments.hp.hp856Xx.MixerMode` attribute), 277  
`Internal` (`pymeasure.instruments.hp.hp856Xx.SourceLevelingControlMode` attribute), 279  
`internal_frequency` (`pymeasure.instruments.agilent.Agilent8257D` property), 154  
`internal_shape` (`pymeasure.instruments.agilent.Agilent8257D` property), 154  
`internalfrequency` (`pymeasure.instruments.srs.SR860` property), 448  
`interpolator_autocalibrated` (`pymeasure.instruments.pendulum.cnt91.CNT91` property), 405  
`interrupt_sequence_command()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 26

`invert` (`pymeasure.instruments.lecroy.lecroyT3DSO1204.LecroyT3DSO1204Channel` property), 374  
`invert_signal_sign` (`pymeasure.instruments.srs.SR570` property), 441  
`ion_gauge_status` (`pymeasure.instruments.mksinst.mks937b.IonGaugeAndPressureChannel` property), 376  
`IonGaugeAndPressureChannel` (class in `pymeasure.instruments.mksinst.mks937b`), 376  
`IPS120_10` (class in `pymeasure.instruments.oxfordinstruments`), 399  
`is_averaging()` (`pymeasure.instruments.agilent.Agilent8722ES` method), 155  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley2000` method), 290  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley2400` method), 302  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley2450` method), 310  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley2700` method), 317  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley6221` method), 321  
`is_buffer_full()` (`pymeasure.instruments.keithley.Keithley6517B` method), 327  
`is_current_stable()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 228  
`is_enabled` (`pymeasure.instruments.agilent.Agilent8257D` property), 154  
`is_enabled()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 228  
`is_enabled()` (`pymeasure.instruments.keysight.KeysightN5767A` method), 342  
`is_moving()` (`pymeasure.instruments.parker.ParkerGV6` method), 403  
`is_out_of_range()` (`pymeasure.instruments.srs.SR830` method), 444  
`is_ready()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 228  
`is_running()` (`pymeasure.display.manager.BaseManager` method), 87  
`is_sequence_running()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 228  
`is_set()` (`pymeasure.experiment.parameters.Metadata` method), 77  
`is_set()` (`pymeasure.experiment.parameters.Parameter` method), 78  
`is_waiting_for_response()` (`pymeasure.instruments.oxfordinstruments.base.OxfordInstrumentsBase` method), 394  
`ITC503` (class in `pymeasure.instruments.oxfordinstruments`), 395  
`ITC503.FLOW_CONTROL_STATUS` (class in `pymeasure.instruments.oxfordinstruments`), 395

## J

`join()` (`pymeasure.display.thread.StoppableQThread` method), 89  
`join()` (`pymeasure.experiment.workers.Worker` method), 79

## K

`Keithley2000` (class in `pymeasure.instruments.keithley`), 287  
`Keithley2200` (class in `pymeasure.instruments.keithley`), 334  
`Keithley2200.BaseChannelCreator` (class in `pymeasure.instruments.keithley`), 334  
`Keithley2200.ChannelCreator` (class in `pymeasure.instruments.keithley`), 334  
`Keithley2200.MultiChannelCreator` (class in `pymeasure.instruments.keithley`), 335  
`Keithley2260B` (class in `pymeasure.instruments.keithley`), 295  
`Keithley2306` (class in `pymeasure.instruments.keithley`), 297  
`Keithley2400` (class in `pymeasure.instruments.keithley`), 300  
`Keithley2450` (class in `pymeasure.instruments.keithley`), 308  
`Keithley2600` (class in `pymeasure.instruments.keithley`), 332  
`Keithley2700` (class in `pymeasure.instruments.keithley`), 315  
`Keithley2750` (class in `pymeasure.instruments.keithley`), 330  
`Keithley6221` (class in `pymeasure.instruments.keithley`), 319  
`Keithley6517B` (class in `pymeasure.instruments.keithley`), 325  
`kelvin` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreTemp` property), 359  
`KeysightDSOX1102G` (class in `pymeasure.instruments.keysight`), 339  
`KeysightE36312A` (class in `pymeasure.instruments.keysight`), 343

- KeysightE36312A.BaseChannelCreator (class in *pymeasure.instruments.keysight*), 344
- KeysightE36312A.ChannelCreator (class in *pymeasure.instruments.keysight*), 344
- KeysightE36312A.MultiChannelCreator (class in *pymeasure.instruments.keysight*), 344
- KeysightN5767A (class in *pymeasure.instruments.keysight*), 341
- KeysightN7776C (class in *pymeasure.instruments.keysight*), 342
- kill() (*pymeasure.instruments.parker.ParkerGV6* method), 403
- kill\_enabled (*pymeasure.instruments.eurotest.EurotestHPP120256* property), 233
- ## L
- labels() (*pymeasure.experiment.results.Results* method), 80
- LakeShore211 (class in *pymeasure.instruments.lakeshore*), 351
- LakeShore211.AnalogMode (class in *pymeasure.instruments.lakeshore*), 351
- LakeShore211.AnalogRange (class in *pymeasure.instruments.lakeshore*), 351
- LakeShore211.RelayMode (class in *pymeasure.instruments.lakeshore*), 351
- LakeShore211.RelayNumber (class in *pymeasure.instruments.lakeshore*), 351
- LakeShore224 (class in *pymeasure.instruments.lakeshore*), 353
- LakeShore331 (class in *pymeasure.instruments.lakeshore*), 355
- LakeShore421 (class in *pymeasure.instruments.lakeshore*), 355
- LakeShore425 (class in *pymeasure.instruments.lakeshore*), 358
- LakeShoreHeaterChannel (class in *pymeasure.instruments.lakeshore.lakeshore\_base*), 359
- LakeShoreTemperatureChannel (class in *pymeasure.instruments.lakeshore.lakeshore\_base*), 359
- lam\_status (*pymeasure.instruments.eurotest.EurotestHPP120256* property), 233
- laser\_enabled (*pymeasure.instruments.toptica.ibeamsmart.IBeamSmart* property), 496
- last\_test\_date (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* property), 458
- last\_test\_date (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* property), 462
- LDCCurrent (*pymeasure.instruments.thorlabs.ThorlabsPro8000* property), 489
- LDCCurrentLimit (*pymeasure.instruments.thorlabs.ThorlabsPro8000* property), 489
- LDCPolarity (*pymeasure.instruments.thorlabs.ThorlabsPro8000* property), 489
- LDCStatus (*pymeasure.instruments.thorlabs.ThorlabsPro8000* property), 489
- learn\_mode (*pymeasure.instruments.temptronic.ATSBASE* property), 479
- LeCroyT3DS01204 (class in *pymeasure.instruments.lecroy*), 360
- LeCroyT3DS01204.BaseChannelCreator (class in *pymeasure.instruments.lecroy*), 361
- LeCroyT3DS01204.ChannelCreator (class in *pymeasure.instruments.lecroy*), 361
- LeCroyT3DS01204.MultiChannelCreator (class in *pymeasure.instruments.lecroy*), 361
- LeCroyT3DS01204Channel (class in *pymeasure.instruments.lecroy.lecroyT3DS01204*), 374
- left\_limit (*pymeasure.instruments.newport.esp300.Axis* property), 378
- level (*pymeasure.instruments.hp.HP8657B* property), 283
- level (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 411
- level\_lin (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209
- level\_log (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209
- level\_mode (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 412
- level\_offset (*pymeasure.instruments.hp.HP8657B* property), 283
- level\_opt\_attn (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209
- level\_scale (*pymeasure.instruments.anritsu.AnritsuMS9710C* property), 209
- lia\_status (*pymeasure.instruments.srs.SR830* property), 444
- limit\_enabled (*pymeasure.instruments.hp.HP8116A* property), 249
- Line (*pymeasure.instruments.hp.hp856Xx.TriggerMode* attribute), 280
- line\_frequency (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* property), 131
- line\_frequency (*pymeasure.instruments.keithley.Keithley2400* property), 302
- line\_frequency\_auto (*pymeasure.instruments.keithley.Keithley2400* property), 302

- `sure.instruments.keithley.Keithley2400` (property), 302
- `LINEAR` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LINEAR_DOUBLE` (`pymeasure.instruments.agilent.agilentB1500.SweepMode` attribute), 194
- `LINEAR_SINGLE` (`pymeasure.instruments.agilent.agilentB1500.SweepMode` attribute), 194
- `LineEditDelegate` (class in `pymeasure.display.widgets.sequencer_widget`), 92
- `list_files()` (`pymeasure.instruments.activetechnologies.AWG401x.AWG` method), 118
- `list_resources()` (in module `pymeasure.instruments`), 116
- `Listener` (class in `pymeasure.experiment.listeners`), 72
- `ListInput` (class in `pymeasure.display.inputs`), 86
- `ListParameter` (class in `pymeasure.experiment.parameters`), 76
- `lo_frequency` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 269
- `load()` (`pymeasure.display.manager.BaseManager` method), 87
- `load()` (`pymeasure.display.manager.Manager` method), 88
- `load()` (`pymeasure.display.widgets.image_widget.ImageWidget` method), 91
- `load()` (`pymeasure.display.widgets.plot_widget.PlotWidget` method), 92
- `load()` (`pymeasure.display.widgets.tab_widget.TabWidget` method), 95
- `load()` (`pymeasure.display.widgets.table_widget.TableWidget` method), 97
- `load()` (`pymeasure.experiment.results.Results` static method), 80
- `load_capacity` (`pymeasure.instruments.tcpowerconversion.CXN` property), 453
- `load_capacity` (`pymeasure.instruments.tcpowerconversion.tccxn.PresetChannel` property), 455
- `load_config` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 131
- `load_data_file()` (`pymeasure.instruments.anritsu.AnritsuMS464xB` method), 218
- `load_data_file_to_memory()` (`pymeasure.instruments.anritsu.AnritsuMS464xB` method), 218
- `load_impedance` (`pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG` property), 120
- `load_sequence()` (`pymeasure.display.widgets.sequencer_widget.SequencerWidget` method), 94
- `load_sequence()` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 422
- `load_setup_file` (`pymeasure.instruments.temptronic.ATSBBase` property), 479
- `local()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 228
- `local()` (`pymeasure.instruments.keithley.Keithley2000` method), 290
- `local_lockout` (`pymeasure.instruments.temptronic.ATSBBase` property), 479
- `locked` (`pymeasure.instruments.keysight.KeysightN7776C` property), 342
- `LOG_10` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_100` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_25` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_250` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_50` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_5000` (`pymeasure.instruments.agilent.agilentB1500.SamplingMode` attribute), 195
- `LOG_DOUBLE` (`pymeasure.instruments.agilent.agilentB1500.SweepMode` attribute), 194
- `log_magnitude()` (`pymeasure.instruments.agilent.Agilent8722ES` method), 155
- `log_ratio` (`pymeasure.instruments.signalrecovery.DSP7225` property), 430
- `log_ratio` (`pymeasure.instruments.signalrecovery.DSP7265` property), 437
- `LOG_SINGLE` (`pymeasure.instruments.agilent.agilentB1500.SweepMode` attribute), 194
- `logarithmic_scale` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 269
- `LogHandler` (class in `pymeasure.display.log`), 87
- `LogHandler.Emitter` (class in `pymeasure.display.log`), 87
- `LogWidget` (class in `pymeasure.display.widgets.log_widget`), 91
- `low_freq` (`pymeasure.instruments.srs.SR570` property), 441
- `low_freq_out_amplitude` (`pymeasure.instruments.keithley.Keithley2400` property), 302

`sure.instruments.agilent.Agilent8257D` (property), 154

`low_freq_out_source` (`pymea-`  
`sure.instruments.agilent.Agilent8257D` prop-  
erty), 154

`low_level` (`pymea-`  
`sure.instruments.hp.HP8116A` prop-  
erty), 249

`lower_sideband_enabled` (`pymea-`  
`sure.instruments.rohdeschwarz.sfm.SFM`  
property), 412

## M

`mag` (`pymea-`  
`sure.instruments.ametek.Ametek7270` prop-  
erty), 199

`mag` (`pymea-`  
`sure.instruments.signalrecovery.DSP7225`  
property), 431

`mag` (`pymea-`  
`sure.instruments.signalrecovery.DSP7265`  
property), 437

`magnet_current` (`pymea-`  
`sure.instruments.ami.AMI430`  
property), 201

`MagnetError` (class in `pymea-`  
`sure.instruments.oxfordinstruments.ips120_10`),  
401

`magnitude` (`pymea-`  
`sure.instruments.srs.SR830` property),  
445

`magnitude` (`pymea-`  
`sure.instruments.srs.SR860` property),  
449

`magnitude()` (`pymea-`  
`sure.instruments.agilent.Agilent8722B`  
method), 155

`main_air_flow_rate` (`pymea-`  
`sure.instruments.temptronic.ATSB` prop-  
erty), 479

`ManagedConsole` (class in `pymea-`  
`sure.display.console`),  
83

`ManagedDockWindow` (class in `pymea-`  
`sure.display.windows.managed_dock_window`),  
101

`ManagedImageWindow` (class in `pymea-`  
`sure.display.windows.managed_image_window`),  
98

`ManagedWindow` (class in `pymea-`  
`sure.display.windows.managed_window`),  
98

`ManagedWindowBase` (class in `pymea-`  
`sure.display.windows.managed_window`),  
99

`Manager` (class in `pymea-`  
`sure.display.manager`), 88

`MANUAL` (`pymea-`  
`sure.instruments.agilent.agilentB1500.ADCMode`  
attribute), 193

`MANUAL` (`pymea-`  
`sure.instruments.agilent.agilentB1500.AutoManual`  
attribute), 194

`MANUAL` (`pymea-`  
`sure.instruments.agilent.agilentB1500.CompliancePolarity`  
attribute), 196

`MANUAL` (`pymea-`  
`sure.instruments.hp.hp856Xx.AmplitudeUnits`  
attribute), 277

`manual_mode` (`pymea-`  
`sure.instruments.tcpowerconversion.CXN`  
property), 453

`manual_trigger_type` (`pymea-`  
`sure.instruments.anritsu.AnritsuMS464xB`  
property), 218

`marker_amplitude` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 266

`marker_delta` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 266

`marker_frequency` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 267

`marker_noise_mode_enabled` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 267

`marker_signal_tracking_enabled` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 268

`marker_threshold` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 267

`marker_time` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx`  
attribute), 268

`master_slave_setting` (`pymea-`  
`sure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38`  
property), 458

`master_slave_setting` (`pymea-`  
`sure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65`  
property), 462

`material` (`pymea-`  
`sure.instruments.aja.DCXS` property),  
197

`math_define` (`pymea-`  
`sure.instruments.lecroy.LeCroyT3DSO1204`  
property), 367

`math_vdiv` (`pymea-`  
`sure.instruments.lecroy.LeCroyT3DSO1204`  
property), 367

`math_vpos` (`pymea-`  
`sure.instruments.lecroy.LeCroyT3DSO1204`  
property), 367

`max_amplitude` (`pymea-`  
`sure.instruments.hp.HP33120A`  
property), 240

`max_current` (`pymea-`  
`sure.instruments.deltaelektronika.SM7045D`  
property), 230

`max_current` (`pymea-`  
`sure.instruments.keithley.Keithley2400`  
property), 302

`max_current` (`pymea-`  
`sure.instruments.keithley.Keithley2450`  
property), 310

`max_current` (`pymea-`  
`sure.instruments.rohdeschwarz.hmp.HMP4040`  
property), 422

`max_frequency` (`pymea-`  
`sure.instruments.hp.HP33120A`  
property), 240

`max_hold_enabled` (`pymea-`

<code>sure.instruments.lakeshore.LakeShore421</code> (property), 357	<code>mean_resistance</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 303
<code>max_hold_field</code> ( <code>pymeasure.instruments.lakeshore.LakeShore421</code> property), 357	<code>mean_resistance</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 310
<code>max_hold_field_raw</code> ( <code>pymeasure.instruments.lakeshore.LakeShore421</code> property), 357	<code>mean_voltage</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 303
<code>max_hold_multiplier</code> ( <code>pymeasure.instruments.lakeshore.LakeShore421</code> property), 357	<code>mean_voltage</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 311
<code>max_hold_reset()</code> ( <code>pymeasure.instruments.lakeshore.LakeShore421</code> method), 357	<code>means</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 303
<code>max_number_of_points</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS464xB</code> property), 218	<code>means</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 311
<code>max_offset</code> ( <code>pymeasure.instruments.hp.HP33120A</code> property), 240	<code>meas_acpower</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 213
<code>max_output_amplitude</code> ( <code>pymeasure.instruments.teledyne.teledyneT3AFG.SignalChannel</code> property), 467	<code>meas_emf_meter_clear_all</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 213
<code>max_resistance</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 302	<code>meas_emf_meter_clear_sample</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>max_resistance</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 310	<code>meas_emf_meter_sample</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>max_voltage</code> ( <code>pymeasure.instruments.deltaelektronika.SM7450</code> property), 230	<code>meas_int_power</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>max_voltage</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 302	<code>meas_iq_capture</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>max_voltage</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 310	<code>meas_iq_capture_fail</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>max_voltage</code> ( <code>pymeasure.instruments.rohdeschwarz.hmp.HMP4046</code> property), 422	<code>meas_mode()</code> ( <code>pymeasure.instruments.agilent.agilentB1500.AgilentB1500</code> method), 185
<code>maximum_case_temperature</code> ( <code>pymeasure.instruments.ipgphotonics.yar.YAR</code> property), 287	<code>meas_op_mode</code> ( <code>pymeasure.instruments.agilent.agilentB1500.SMU</code> property), 188
<code>maximum_test_time</code> ( <code>pymeasure.instruments.temptronic.ATSB</code> property), 479	<code>meas_ota_mapp</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>maximums</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 303	<code>meas_ota_run</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>maximums</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 310	<code>meas_console</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>maxspeed</code> ( <code>pymeasure.instruments.anaheimautomation.DPSeriesMotorController</code> property), 203	<code>meas_power</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>mean_current</code> ( <code>pymeasure.instruments.keithley.Keithley2400</code> property), 303	<code>meas_power_all</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS2090A</code> property), 214
<code>mean_current</code> ( <code>pymeasure.instruments.keithley.Keithley2450</code> property), 310	<code>meas_range_current</code> ( <code>pymeasure</code> property), 310

*sure.instruments.agilent.agilentB1500.SMU*  
 property), 189  
 meas\_range\_current\_auto() (*pymeasure.instruments.agilent.agilentB1500.SMU*  
 method), 189  
 meas\_range\_voltage (*pymeasure.instruments.agilent.agilentB1500.SMU*  
 property), 189  
 MeasMode (class in *pymeasure.instruments.agilent.agilentB1500*), 194  
 MeasOpMode (class in *pymeasure.instruments.agilent.agilentB1500*), 194  
 Measurable (class in *pymeasure.experiment.parameters*), 76  
 measure() (*pymeasure.instruments.agilent.agilent4156.Agilent4156*  
 method), 166  
 measure() (*pymeasure.instruments.lakeshore.LakeShore425*  
 method), 359  
 measure\_ACI (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_ACV (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_capacity() (*pymeasure.instruments.attocube.anc300.Axis*  
 method), 225  
 measure\_concurrent\_functions (*pymeasure.instruments.keithley.Keithley2400* prop-  
 erty), 303  
 measure\_continuity() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 290  
 measure\_current (*pymeasure.instruments.deltaelektronika.SM7045D*  
 property), 230  
 measure\_current() (*pymeasure.instruments.advantest.advantestR624X.SMUChannel*  
 method), 139  
 measure\_current() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 290  
 measure\_current() (*pymeasure.instruments.keithley.Keithley2400*  
 method), 303  
 measure\_current() (*pymeasure.instruments.keithley.Keithley2450*  
 method), 311  
 measure\_current() (*pymeasure.instruments.keithley.Keithley6517B*  
 method), 327  
 measure\_DCI (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_DCV (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_delay (*pymeasure.instruments.lecroy.LeCroyT3DSO1204*  
 property), 367  
 measure\_diode() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 291  
 measure\_frequency() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 291  
 measure\_mode (*pymeasure.instruments.anritsu.AnritsuMS9710C*  
 property), 209  
 measure\_parameter() (*pymeasure.instruments.lecroy.LeCroyT3DSO1204*  
 method), 368  
 measure\_parameter() (*pymeasure.instruments.teledyne.teledyne\_oscilloscope.TeledyneOscilloscope*  
 method), 474  
 measure\_parameter() (*pymeasure.instruments.teledyne.TeledyneOscilloscope*  
 method), 469  
 measure\_peak() (*pymeasure.instruments.anritsu.AnritsuMS9710C*  
 method), 209  
 measure\_period() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 291  
 measure\_R2W (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_R4W (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_resistance() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 291  
 measure\_resistance() (*pymeasure.instruments.keithley.Keithley2400*  
 method), 303  
 measure\_resistance() (*pymeasure.instruments.keithley.Keithley2450*  
 method), 311  
 measure\_resistance() (*pymeasure.instruments.keithley.Keithley6517B*  
 method), 327  
 measure\_Rext (*pymeasure.instruments.hp.HP3478A*  
 property), 246  
 measure\_temperature() (*pymeasure.instruments.keithley.Keithley2000*  
 method), 291  
 measure\_voltage (*pymeasure.instruments.deltaelektronika.SM7045D*  
 property), 230  
 measure\_voltage() (*pymeasure.instruments.advantest.advantestR624X.SMUChannel*  
 method), 134  
 measure\_voltage() (*pymeasure.instruments.agilent.agilentB1500.SMU*  
 property), 189

*sure.instruments.keithley.Keithley2000*  
*method*), 291  
**measure\_voltage()** (*pymeasure.instruments.keithley.Keithley2400*  
*method*), 303  
**measure\_voltage()** (*pymeasure.instruments.keithley.Keithley2450*  
*method*), 311  
**measure\_voltage()** (*pymeasure.instruments.keithley.Keithley6517B*  
*method*), 327  
**measured\_current** (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040*  
*property*), 422  
**measured\_voltage** (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040*  
*property*), 422  
**measurement()** (*pymeasure.instruments.common\_base.CommonBase*  
*static method*), 107  
**measurement()** (*pymeasure.instruments.keysight.KeysightE36312A*  
*static method*), 348  
**measurement()** (*pymeasure.instruments.lecroy.LeCroyT3DSO1204*  
*static method*), 368  
**measurement\_count** (*pymeasure.instruments.advantest.advantestR624X.SMUChannel*  
*property*), 141  
**measurement\_event\_enabled** (*pymeasure.instruments.keithley.Keithley6221*  
*property*), 321  
**measurement\_events** (*pymeasure.instruments.keithley.Keithley6221*  
*property*), 321  
**measurement\_parameter** (*pymeasure.instruments.anritsu.anritsuMS464xB.Trace*  
*property*), 223  
**measurement\_time** (*pymeasure.instruments.pendulum.cnt91.CNT91*  
*property*), 405  
**MeasurementChannel** (*class in pymeasure.instruments.anritsu.anritsuMS464xB*),  
220  
**MeasurementType** (*class in pymeasure.instruments.advantest.advantestR624X*),  
144  
**memory\_size** (*pymeasure.instruments.lecroy.LeCroyT3DSO1204*  
*property*), 369  
**memory\_size** (*pymeasure.instruments.teledyne.TeledyneOscilloscope*  
*property*), 469  
**menu** (*pymeasure.instruments.lecroy.LeCroyT3DSO1204*  
*property*), 369  
**MESSAGE** (*pymeasure.instruments.hp.hp856Xx.StatusRegister*  
*attribute*), 279  
**message\_waiting()** (*pymeasure.experiment.listeners.Listener*  
*method*),  
72  
**Metadata** (*class in pymeasure.experiment.parameters*),  
76  
**metadata()** (*pymeasure.experiment.results.Results*  
*method*), 80  
**metadata\_objects()** (*pymeasure.experiment.procedure.Procedure*  
*method*),  
73  
**min\_amplitude** (*pymeasure.instruments.hp.HP33120A*  
*property*), 240  
**min\_current** (*pymeasure.instruments.keithley.Keithley2400*  
*property*), 303  
**min\_current** (*pymeasure.instruments.keithley.Keithley2450*  
*property*), 311  
**min\_current** (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040*  
*property*), 422  
**min\_frequency** (*pymeasure.instruments.hp.HP33120A*  
*property*), 240  
**min\_offset** (*pymeasure.instruments.hp.HP33120A*  
*property*), 240  
**min\_resistance** (*pymeasure.instruments.keithley.Keithley2400*  
*property*), 303  
**min\_resistance** (*pymeasure.instruments.keithley.Keithley2450*  
*property*), 311  
**min\_voltage** (*pymeasure.instruments.keithley.Keithley2400*  
*property*), 303  
**min\_voltage** (*pymeasure.instruments.keithley.Keithley2450*  
*property*), 311  
**min\_voltage** (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040*  
*property*), 422  
**minimum\_display\_power** (*pymeasure.instruments.ipgphotonics.yar.YAR*  
*property*), 287  
**minimums** (*pymeasure.instruments.keithley.Keithley2400*  
*property*), 304  
**minimums** (*pymeasure.instruments.keithley.Keithley2450*  
*property*), 311  
**mixer\_bias** (*pymeasure.instruments.hp.HP8561B*  
*property*), 275  
**mixer\_level** (*pymeasure.instruments.hp.hp856Xx.HP856Xx*  
*attribute*), 255  
**mixer\_mode** (*pymeasure.instruments.hp.HP8561B*  
*property*), 275  
**MixerMode** (*class in pymeasure.instruments.hp.hp856Xx*), 277  
**MKS937B** (*class in pymeasure.instruments.mksinst.mks937b*), 375  
**mode** (*pymeasure.instruments.agilent.AgilentE4980*  
*property*), 157

`mode` (*pymeasure.instruments.attocube.anc300.Axis property*), 225

`mode` (*pymeasure.instruments.hp.HP3478A property*), 246

`mode` (*pymeasure.instruments.keithley.Keithley2000 property*), 291

`mode` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property*), 458

`mode` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property*), 462

`mode` (*pymeasure.instruments.temptronic.ATSBBase property*), 480

`mode` (*pymeasure.instruments.thermotron.Thermotron3800 property*), 488

`modulation_degree` (*pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel property*), 417

`modulation_enabled` (*pymeasure.instruments.rohdeschwarz.sfm.SFM property*), 412

`modulation_enabled` (*pymeasure.instruments.rohdeschwarz.sfm.Sound\_Channel property*), 418

`module`

- `pymeasure.display.browser`, 83
- `pymeasure.display.console`, 83
- `pymeasure.display.curves`, 84
- `pymeasure.display.inputs`, 85
- `pymeasure.display.listeners`, 87
- `pymeasure.display.log`, 87
- `pymeasure.display.manager`, 87
- `pymeasure.display.plotter`, 89
- `pymeasure.display.thread`, 89
- `pymeasure.display.widgets.browser_widget`, 90
- `pymeasure.display.widgets.directory_widget`, 90
- `pymeasure.display.widgets.dock_widget`, 95
- `pymeasure.display.widgets.estimator_widget`, 90
- `pymeasure.display.widgets.image_frame`, 90
- `pymeasure.display.widgets.image_widget`, 90
- `pymeasure.display.widgets.inputs_widget`, 91
- `pymeasure.display.widgets.log_widget`, 91
- `pymeasure.display.widgets.plot_frame`, 91
- `pymeasure.display.widgets.plot_widget`, 92
- `pymeasure.display.widgets.results_dialog`, 92
- `pymeasure.display.widgets.sequencer_widget`, 92
- `pymeasure.display.widgets.tab_widget`, 94
- `pymeasure.display.widgets.table_widget`, 95
- `pymeasure.display.windows.managed_dock_window`, 101
- `pymeasure.display.windows.managed_image_window`, 98
- `pymeasure.display.windows.managed_window`, 98
- `pymeasure.display.windows.plotter_window`, 100
- `pymeasure.experiment.experiment`, 71
- `pymeasure.experiment.listeners`, 72
- `pymeasure.experiment.parameters`, 74
- `pymeasure.experiment.procedure`, 73
- `pymeasure.experiment.results`, 79
- `pymeasure.experiment.workers`, 79
- `pymeasure.instruments`, 101
- `pymeasure.instruments.activetechnologies`, 116
- `pymeasure.instruments.advantest`, 121
- `pymeasure.instruments.advantest.advantestR3767CG`, 122
- `pymeasure.instruments.advantest.advantestR624X`, 143
- `pymeasure.instruments.agilent`, 152
- `pymeasure.instruments.agilent.agilent4156`, 163
- `pymeasure.instruments.agilent.agilentB1500`, 193
- `pymeasure.instruments.aja`, 196
- `pymeasure.instruments.ametek`, 198
- `pymeasure.instruments.ami`, 200
- `pymeasure.instruments.anaheimautomation`, 202
- `pymeasure.instruments.anapico`, 204
- `pymeasure.instruments.andeenhagerling`, 205
- `pymeasure.instruments.anritsu`, 208
- `pymeasure.instruments.attocube`, 223
- `pymeasure.instruments.bkprecision`, 226
- `pymeasure.instruments.comedi`, 116
- `pymeasure.instruments.danfysik`, 226
- `pymeasure.instruments.deltaelektronika`, 229
- `pymeasure.instruments.edwards`, 231
- `pymeasure.instruments.eurotest`, 231
- `pymeasure.instruments.fluke`, 233
- `pymeasure.instruments.fwbell`, 234
- `pymeasure.instruments.hcp`, 237
- `pymeasure.instruments.heidenhain`, 237
- `pymeasure.instruments.hp`, 239
- `pymeasure.instruments.ipgphotonics`, 286
- `pymeasure.instruments.keithley`, 287
- `pymeasure.instruments.keysight`, 338
- `pymeasure.instruments.lakeshore`, 351

- `pymeasure.instruments.lecroy`, 360
- `pymeasure.instruments.mksinst`, 375
- `pymeasure.instruments.newport`, 376
- `pymeasure.instruments.ni`, 378
- `pymeasure.instruments.novanta`, 392
- `pymeasure.instruments.oxfordinstruments`, 393
- `pymeasure.instruments.parker`, 403
- `pymeasure.instruments.pendulum`, 404
- `pymeasure.instruments.razorbill`, 405
- `pymeasure.instruments.rohdeschwarz`, 406
- `pymeasure.instruments.siglenttechnologies`, 423
- `pymeasure.instruments.signalrecovery`, 427
- `pymeasure.instruments.srs`, 440
- `pymeasure.instruments.tcpowerconversion`, 452
- `pymeasure.instruments.tdk`, 456
- `pymeasure.instruments.tektronix`, 464
- `pymeasure.instruments.teledyne`, 465
- `pymeasure.instruments.temptronic`, 476
- `pymeasure.instruments.texio`, 484
- `pymeasure.instruments.thermotron`, 487
- `pymeasure.instruments.thorlabs`, 488
- `pymeasure.instruments.thyracont`, 489
- `pymeasure.instruments.toptica`, 494
- `pymeasure.instruments.validators`, 113
- `pymeasure.instruments.velleman`, 497
- `pymeasure.instruments.yokogawa`, 499
- `pymeasure.test`, 66
- `Monitor` (class in `pymeasure.display.listeners`), 87
- `Monitor` (class in `pymeasure.experiment.listeners`), 72
- `monitored_value` (`pymeasure.instruments.hcp.TC038` property), 238
- `motion_done` (`pymeasure.instruments.newport.esp300.Axis` property), 378
- `mouseMoved()` (`pymeasure.display.curves.Crosshairs` method), 84
- `mout` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreHeaterChannel` property), 360
- `move()` (`pymeasure.instruments.anaheimautomation.DPSeriesMotorController` method), 203
- `move()` (`pymeasure.instruments.attocube.anc300.Axis` method), 225
- `move()` (`pymeasure.instruments.parker.ParkerGV6` method), 403
- `move_raw()` (`pymeasure.instruments.attocube.anc300.Axis` method), 225
- `mroll_frequency` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 269
- `multidrop_capability` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` property), 458
- `multidrop_capability` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` property), 462
- N**
- `NA` (`pymeasure.instruments.hp.hp856Xx.StatusRegister` attribute), 279
- `ND287` (class in `pymeasure.instruments.heidenhain`), 237
- `NegativePeak` (`pymeasure.instruments.hp.hp856Xx.DetectionModes` attribute), 278
- `new_curve()` (`pymeasure.display.widgets.dock_widget.DockWidget` method), 95
- `new_curve()` (`pymeasure.display.widgets.image_widget.ImageWidget` method), 91
- `new_curve()` (`pymeasure.display.widgets.plot_widget.PlotWidget` method), 92
- `new_curve()` (`pymeasure.display.widgets.tab_widget.TabWidget` method), 95
- `new_curve()` (`pymeasure.display.widgets.table_widget.TableWidget` method), 97
- `next()` (`pymeasure.display.manager.BaseManager` method), 87
- `next()` (`pymeasure.display.manager.ExperimentQueue` method), 88
- `next_setpoint()` (`pymeasure.instruments.temptronic.ATS545` method), 484
- `next_setpoint()` (`pymeasure.instruments.temptronic.ATSBASE` method), 480
- `next_step()` (`pymeasure.instruments.keysight.KeysightN7776C` method), 342
- `NextHigh` (`pymeasure.instruments.hp.hp856Xx.PeakSearchMode` attribute), 279
- `NextLeft` (`pymeasure.instruments.hp.hp856Xx.PeakSearchMode` attribute), 279
- `NextRight` (`pymeasure.instruments.hp.hp856Xx.PeakSearchMode` attribute), 279
- `nicam_additional_bits` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412
- `nicam_audio_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412
- `nicam_audio_volume` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412
- `nicam_bit_error_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412
- `nicam_bit_error_rate` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412

`nicam_carrier_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412

`nicam_carrier_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412

`nicam_carrier_level` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_control_bits` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_data` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_intercarrier_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_IQ_inverted` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 412

`nicam_mode` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_preemphasis_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_source` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`nicam_test_signal` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 413

`NONE` (`pymeasure.instruments.hp.hp856Xx.StatusRegister` attribute), 279

`Normal` (`pymeasure.instruments.hp.hp856Xx.DetectionModes` attribute), 278

`normal_channel` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 414

`normalize_trace_data_enabled` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 270

`normalized_reference_level` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 270

`normalized_reference_position` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 271

`not_at_temperature()` (`pymeasure.instruments.temptronic.ATSBASE` method), 480

`nozzle_air_flow_rate` (`pymeasure.instruments.temptronic.ATSBASE` property), 480

`nplc` (`pymeasure.instruments.hp.HP34401A` property), 241

`null_operation_enabled` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 141

`num_ch` (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 118

`num_dch` (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 118

`number_of_channels` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 218

`number_of_points` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 222

`number_of_ports` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 219

`number_of_traces` (`pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel` property), 222

`Number_readings` (`pymeasure.instruments.hp.HP3437A` property), 244

`Nxds` (class in `pymeasure.instruments.edwards`), 231

## O

`OCP_enabled` (`pymeasure.instruments.hp.HP6632A` property), 284

`Off` (`pymeasure.instruments.hp.hp856Xx.DemodulationMode` attribute), 278

`offset` (`pymeasure.instruments.agilent.Agilent33220A` property), 171

`offset` (`pymeasure.instruments.agilent.Agilent33500` property), 174

`offset` (`pymeasure.instruments.agilent.agilent33500.Agilent33500Channel` property), 177

`offset` (`pymeasure.instruments.agilent.agilent4156.VARD` property), 168

`offset` (`pymeasure.instruments.hp.HP33120A` property), 240

`offset` (`pymeasure.instruments.hp.HP8116A` property), 249

`offset` (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOs` property), 474

`offset` (`pymeasure.instruments.teledyne.teledyneT3AFG.SignalChannel` property), 467

`offset_current` (`pymeasure.instruments.srs.SR570` property), 442

`offset_current_enabled` (`pymeasure.instruments.srs.SR570` property), 442

`offset_current_sign` (`pymeasure.instruments.srs.SR570` property), 442

`offset_voltage` (`pymeasure.instruments.attocube.anc300.Axis` property), 231

erty), 225

open() (pymeasure.instruments.keithley.Keithley2750 method), 331

open\_all() (pymeasure.instruments.keithley.Keithley2750 method), 331

open\_all\_channels() (pymeasure.instruments.keithley.Keithley2700 method), 317

open\_channels (pymeasure.instruments.keithley.Keithley2700 property), 317

open\_file\_externally() (pymeasure.display.windows.managed\_window.ManagedWindowBase method), 100

open\_rows\_to\_columns() (pymeasure.instruments.keithley.Keithley2700 method), 317

operating\_hours (pymeasure.instruments.thyracont.smartline\_v2.SmartlineV2 property), 493

operating\_mode (pymeasure.instruments.hp.HP8116A property), 249

operation\_enable\_reg (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 414

operation\_event\_enabled (pymeasure.instruments.keithley.Keithley6221 property), 321

operation\_events (pymeasure.instruments.keithley.Keithley6221 property), 321

operation\_mode (pymeasure.instruments.tcpowerconversion.CXN property), 453

operation\_register (pymeasure.instruments.advantest.advantestR624X.SMUChannel property), 142

options (pymeasure.instruments.andeenhagerling.AH2700A property), 207

options (pymeasure.instruments.fwbell.FWBell5080 property), 235

options (pymeasure.instruments.hp.HP8116A property), 249

options (pymeasure.instruments.Instrument property), 110

options (pymeasure.instruments.keithley.Keithley2000 property), 291

options (pymeasure.instruments.keithley.Keithley2200 property), 337

options (pymeasure.instruments.keithley.Keithley2260B property), 296

options (pymeasure.instruments.keithley.Keithley2306 property), 298

options (pymeasure.instruments.keithley.Keithley2400 property), 304

options (pymeasure.instruments.keithley.Keithley2450 property), 311

options (pymeasure.instruments.keithley.Keithley2600 property), 333

options (pymeasure.instruments.keithley.Keithley2700 property), 317

options (pymeasure.instruments.keithley.Keithley2750 property), 331

options (pymeasure.instruments.keithley.Keithley6221 property), 321

options (pymeasure.instruments.keithley.Keithley6517B property), 328

options (pymeasure.instruments.keysight.KeysightE36312A property), 349

options (pymeasure.instruments.lecroy.LeCroyT3DSO1204 property), 369

options (pymeasure.instruments.signalrecovery.DSP7225 property), 431

options (pymeasure.instruments.signalrecovery.DSP7265 property), 437

options (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property), 458

options (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property), 462

options (pymeasure.instruments.teledyne.TeledyneT3AFG property), 466

options (pymeasure.instruments.texio.TexioPSW360L30 property), 486

oroll\_frequency (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 269

output (pymeasure.instruments.agilent.Agilent33220A property), 171

output (pymeasure.instruments.agilent.Agilent33500 property), 174

output (pymeasure.instruments.agilent.agilent33500.Agilent33500Channel property), 177

output (pymeasure.instruments.anritsu.AnritsuMG3692C property), 208

output (pymeasure.instruments.lakeshore.lakeshore\_base.LakeShoreHeater property), 360

output (pymeasure.instruments.srs.SR510 property), 440

output\_1 (pymeasure.instruments.lakeshore.LakeShore331 attribute), 355

output\_1 (pymeasure.instruments.razorbill.razorbillRP100 property), 406

output\_2 (pymeasure.instruments.lakeshore.LakeShore331 attribute), 355

output\_2 (pymeasure.instruments.razorbill.razorbillRP100 property), 406

output\_all\_measurements() (pymeasure.instruments.advantest.advantestR624X.SMUChannel property), 142

- method), 140
- output\_conversion() (pymea-  
sure.instruments.srs.SR830 method), 445
- output\_enable\_register (pymea-  
sure.instruments.advantest.advantestR624X.SMUChannel  
property), 142
- output\_enabled (pymea-  
sure.instruments.eurotest.EurotestHPP120256  
property), 233
- output\_enabled (pymea-  
sure.instruments.hp.HP6632A  
property), 285
- output\_enabled (pymea-  
sure.instruments.hp.HP8116A  
property), 249
- output\_enabled (pymea-  
sure.instruments.hp.HP8657B  
property), 283
- output\_enabled (pymea-  
sure.instruments.keithley.keithley2200.PSChannel  
property), 338
- output\_enabled (pymea-  
sure.instruments.keithley.Keithley2260B  
property), 296
- output\_enabled (pymea-  
sure.instruments.keysight.keysightE36312A.VoltageSource  
property), 351
- output\_enabled (pymea-  
sure.instruments.keysight.KeysightN7776C  
property), 342
- output\_enabled (pymea-  
sure.instruments.rohdeschwarz.hmp.HMP4040  
property), 422
- output\_enabled (pymea-  
sure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38  
property), 458
- output\_enabled (pymea-  
sure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65  
property), 462
- output\_enabled (pymea-  
sure.instruments.teledyne.teledyneT3AFG.SignalChannel  
property), 467
- output\_enabled (pymea-  
sure.instruments.texio.TexioPSW360L30  
property), 486
- output\_impedance (pymea-  
sure.instruments.activetechnologies.AWG401x.ChannelAFG  
property), 120
- output\_load (pymea-  
sure.instruments.agilent.Agilent33500.ChannelAFG  
property), 174
- output\_load (pymea-  
sure.instruments.agilent.agilent33500.Agilent33500Channel  
property), 177
- output\_low\_grounded (pymea-  
sure.instruments.keithley.Keithley6221 prop-  
erty), 321
- output\_off\_state (pymea-  
sure.instruments.keithley.Keithley2400 prop-  
erty), 304
- output\_trigger\_on\_external() (pymea-  
sure.instruments.keithley.Keithley2400  
method), 304
- output\_trigger\_on\_external() (pymea-  
sure.instruments.keithley.Keithley6221  
method), 321
- output\_voltage (pymea-  
sure.instruments.attocube.anc300.Axis prop-  
erty), 225
- output\_voltage (pymea-  
sure.instruments.fakes.SwissArmyFake prop-  
erty), 113
- output\_voltage (pymea-  
sure.instruments.rohdeschwarz.sfm.SFM  
property), 414
- outputs\_enabled (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.PowerSupply  
property), 389
- OutputType (class in pymea-  
sure.instruments.advantest.advantestR624X),  
143
- over\_voltage (pymea-  
sure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38  
property), 458
- over\_voltage (pymea-  
sure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65  
property), 462
- over\_voltage\_limit (pymea-  
sure.instruments.hp.HP6632A property),  
285
- OxfordInstrumentsBase (class in pymea-  
sure.instruments.oxfordinstruments.base),  
393
- OxfordVISAError (class in pymea-  
sure.instruments.oxfordinstruments.base),  
394
- pandas\_column\_count() (pymea-  
sure.display.widgets.table\_widget.PandasModelBase  
method), 96
- pandas\_column\_count() (pymea-  
sure.display.widgets.table\_widget.PandasModelByColumn  
method), 96
- pandas\_column\_count() (pymea-  
sure.display.widgets.table\_widget.PandasModelByRow  
method), 96
- pandas\_row\_count() (pymea-  
sure.display.widgets.table\_widget.PandasModelBase  
method), 96
- pandas\_row\_count() (pymea-  
sure.display.widgets.table\_widget.PandasModelByColumn  
method), 96

**pandas\_row\_count()** (pymea-  
 sure.display.widgets.table\_widget.PandasModelByColumn  
 method), 97  
**PandasModelBase** (class in pymea-  
 sure.display.widgets.table\_widget), 95  
**PandasModelByColumn** (class in pymea-  
 sure.display.widgets.table\_widget), 96  
**PandasModelByRow** (class in pymea-  
 sure.display.widgets.table\_widget), 97  
**parallel\_meas** (pymea-  
 sure.instruments.agilent.agilentB1500.AgilentB1500  
 property), 185  
**Parameter** (class in pymeasure.experiment.parameters),  
 77  
**parameter** (pymeasure.display.inputs.Input property),  
 85  
**parameter\_DAT1** (pymeasure.instruments.srs.SR860  
 property), 449  
**parameter\_DAT2** (pymeasure.instruments.srs.SR860  
 property), 449  
**parameter\_DAT3** (pymeasure.instruments.srs.SR860  
 property), 449  
**parameter\_DAT4** (pymeasure.instruments.srs.SR860  
 property), 449  
**parameter\_objects()** (pymea-  
 sure.experiment.procedure.Procedure method),  
 73  
**parameter\_values()** (pymea-  
 sure.experiment.procedure.Procedure method),  
 73  
**parameters\_are\_set()** (pymea-  
 sure.experiment.procedure.Procedure method),  
 74  
**parent()** (pymeasure.display.widgets.sequencer\_widget.SequencerWidget  
 method), 93  
**ParkerGV6** (class in pymeasure.instruments.parker), 403  
**parse()** (pymeasure.experiment.results.Results method),  
 80  
**parse\_axis()** (pymea-  
 sure.display.widgets.plot\_frame.PlotFrame  
 method), 91  
**parse\_columns()** (pymea-  
 sure.experiment.procedure.Procedure static  
 method), 74  
**parse\_header()** (pymeasure.experiment.results.Results  
 static method), 80  
**parse\_stream()** (pymeasure.generator.Generator  
 method), 70  
**pass\_filter** (pymeasure.instruments.tdk.tdk\_gen40\_38.TDKGen40\_38  
 property), 458  
**pass\_filter** (pymeasure.instruments.tdk.tdk\_gen80\_65.TDKGen80\_65  
 property), 462  
**pattern\_down** (pymea-  
 sure.instruments.attocube.anc300.Axis prop-  
 erty), 225  
**pattern\_up** (pymeasure.instruments.attocube.anc300.Axis  
 property), 226  
**pause()** (pymeasure.instruments.agilent.agilentB1500.AgilentB1500  
 method), 184  
**pause()** (pymeasure.instruments.ami.AMI430 method),  
 201  
**pb\_desc** (pymeasure.instruments.hp.HP3437A at-  
 tribute), 244  
**peak\_excursion** (pymea-  
 sure.instruments.hp.hp856Xx.HP856Xx at-  
 tribute), 268  
**peak\_preselector()** (pymea-  
 sure.instruments.hp.HP8561B method), 275  
**peak\_search** (pymeasure.instruments.anritsu.AnritsuMS9710C  
 property), 210  
**PeakSearchMode** (class in pymea-  
 sure.instruments.hp.hp856Xx), 279  
**period** (pymeasure.instruments.keithley.Keithley2000  
 property), 291  
**period\_aperature** (pymea-  
 sure.instruments.keithley.Keithley2000 prop-  
 erty), 291  
**period\_digits** (pymea-  
 sure.instruments.keithley.Keithley2000 prop-  
 erty), 291  
**period\_reference** (pymea-  
 sure.instruments.keithley.Keithley2000 prop-  
 erty), 291  
**period\_threshold** (pymea-  
 sure.instruments.keithley.Keithley2000 prop-  
 erty), 291  
**persistent\_field** (pymea-  
 sure.instruments.oxfordinstruments.IPS120\_10  
 property), 400  
**phase** (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG  
 property), 120  
**phase** (pymeasure.instruments.agilent.Agilent33500  
 property), 174  
**phase** (pymeasure.instruments.agilent.agilent33500.Agilent33500Channel  
 property), 177  
**phase** (pymeasure.instruments.ametek.Ametek7270 prop-  
 erty), 199  
**phase** (pymeasure.instruments.signalrecovery.DSP7225  
 property), 431  
**phase** (pymeasure.instruments.signalrecovery.DSP7265  
 property), 437  
**phase** (pymeasure.instruments.srs.SR510 property), 440  
**phase** (pymeasure.instruments.srs.SR830 property), 445  
**phase** (pymeasure.instruments.srs.SR860 property), 449  
**phase** (pymeasure.instruments.agilent.Agilent8722ES  
 method), 156  
**phase\_max** (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG  
 property), 120

`phase_min` (`pymeasure.instruments.activetechnologies.AWG6400` property), 120

`phase_sync` (`pymeasure.instruments.agilent.Agilent33500` method), 174

`PhysicalParameter` (class in `pymeasure.experiment.parameters`), 78

`piezo` (`pymeasure.instruments.thyracont.smartline_v2.VSR` attribute), 494

`ping` (`pymeasure.instruments.hcp.TC038D` method), 239

`ping` (`pymeasure.instruments.tcpowerconversion.CXN` method), 453

`pirani` (`pymeasure.instruments.thyracont.smartline_v2.VSH` attribute), 494

`pirani` (`pymeasure.instruments.thyracont.smartline_v2.VSR` attribute), 494

`PLC` (`pymeasure.instruments.agilent.agilentB1500.ADCMode` attribute), 193

`plot` (`pymeasure.experiment.experiment.Experiment` method), 72

`plot_live` (`pymeasure.experiment.experiment.Experiment` method), 72

`PlotFrame` (class in `pymeasure.display.widgets.plot_frame`), 91

`Plotter` (class in `pymeasure.display.plotter`), 89

`PlotterWindow` (class in `pymeasure.display.windows.plotter_window`), 100

`PlotWidget` (class in `pymeasure.display.widgets.plot_widget`), 92

`pointer` (`pymeasure.instruments.oxfordinstruments.ITC503` property), 397

`points` (`pymeasure.instruments.agilent.agilent4156.VAR2` property), 168

`polarity` (`pymeasure.instruments.danfysik.Danfysik8500` property), 228

`Port` (class in `pymeasure.instruments.anritsu.anritsuMS464xB`), 223

`position` (`pymeasure.instruments.heidenhain.ND287` property), 237

`position` (`pymeasure.instruments.newport.esp300.Axis` property), 378

`position` (`pymeasure.instruments.parker.ParkerGV6` property), 403

`position_error` (`pymeasure.instruments.parker.ParkerGV6` property), 403

`PositivePeak` (`pymeasure.instruments.hp.hp856Xx.DetectionModes` attribute), 278

`power` (`pymeasure.instruments.agilent.Agilent8257D` property), 154

`power` (`pymeasure.instruments.aja.DCXS` property), 197

`power` (`pymeasure.instruments.analog4FCinstruments.anapico.APSIN12G` property), 205

`power` (`pymeasure.instruments.anritsu.AnritsuMG3692C` property), 208

`power` (`pymeasure.instruments.ipgphotonics.yar.YAR` property), 287

`power` (`pymeasure.instruments.keithley.keithley2200.PSChannel` property), 338

`power` (`pymeasure.instruments.keithley.Keithley2260B` property), 296

`power` (`pymeasure.instruments.novanta.Fpu60` property), 393

`power` (`pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDCh` property), 425

`power` (`pymeasure.instruments.tcpowerconversion.CXN` property), 454

`power` (`pymeasure.instruments.texio.TexioPSW360L30` property), 486

`power` (`pymeasure.instruments.thorlabs.ThorlabsPM100USB` property), 489

`power` (`pymeasure.instruments.toptica.ibeamsmart.DriverChannel` property), 497

`power` (`pymeasure.instruments.toptica.ibeamsmart.IBeamSmart` property), 496

`power_density` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 214

`power_enabled` (`pymeasure.instruments.mksinst.mks937b.PressureChannel` property), 376

`power_level` (`pymeasure.instruments.anritsu.anritsuMS464xB.Port` property), 223

`power_limit` (`pymeasure.instruments.tcpowerconversion.CXN` property), 454

`power_on_clear` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 130

`power_range` (`pymeasure.instruments.ipgphotonics.yar.YAR` property), 287

`power_setpoint` (`pymeasure.instruments.ipgphotonics.yar.YAR` property), 287

`power_setpoint` (`pymeasure.instruments.novanta.Fpu60` property), 393

`preamp` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 214

`preemphasis_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.Sound_Channel` property), 418

`preemphasis_time` (`pymeasure.instruments.rohdeschwarz.sfm.Sound_Channel` property), 418

`prepare` (`pymeasure.display.curves.BufferCurve`

`method`), 84  
`preselector_dac_number` (`pymea-`  
`sure.instruments.hp.HP8561B` `property`), 275  
`preset()` (`pymea.instruments.hp.hp856Xx.HP856Xx`  
`method`), 251  
`preset_1` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 452  
`preset_2` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 452  
`preset_3` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 452  
`preset_4` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 452  
`preset_5` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 453  
`preset_6` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 453  
`preset_7` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 453  
`preset_8` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 453  
`preset_9` (`pymea.instruments.tcpowerconversion.CXN`  
`attribute`), 453  
`preset_slot` (`pymea.instruments.tcpowerconversion.CXN`  
`property`), 454  
`PresetChannel` (`class in pymea-`  
`sure.instruments.tcpowerconversion.tccxn`), 455  
`pressure` (`pymea.instruments.mksinst.mks937b.Pressure`  
`property`), 376  
`pressure` (`pymea.instruments.thyracont.smartline_v1.SmartlineV1`  
`property`), 490  
`pressure` (`pymea.instruments.thyracont.smartline_v2.SmartlineV2`  
`property`), 493  
`PressureChannel` (`class in pymea-`  
`sure.instruments.mksinst.mks937b`), 376  
`preview_widget()` (`pymea-`  
`sure.display.widgets.plot_widget.PlotWidget`  
`method`), 92  
`preview_widget()` (`pymea-`  
`sure.display.widgets.tab_widget.TabWidget`  
`method`), 95  
`preview_widget()` (`pymea-`  
`sure.display.widgets.table_widget.TableWidget`  
`method`), 98  
`previous_step()` (`pymea-`  
`sure.instruments.keysight.KeysightN7776C`  
`method`), 342  
`probe_attenuation` (`pymea-`  
`sure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope`  
`property`), 474  
`probe_type` (`pymea.instruments.lakeshore.LakeShore440`  
`property`), 357  
`Procedure` (`class in pymea.experiment.procedure`), 73  
`product_name` (`pymea-`  
`sure.instruments.thyracont.smartline_v2.SmartlineV2`  
`property`), 493  
`program_sweep()` (`pymea-`  
`sure.instruments.oxfordinstruments.ITC503`  
`method`), 397  
`PrologixAdapter` (`class in pymea.adapters`), 56  
`proportional_band` (`pymea-`  
`sure.instruments.oxfordinstruments.ITC503`  
`property`), 397  
`protect_state_enabled` (`pymea-`  
`sure.instruments.hp.hp856Xx.HP856Xx` `at-`  
`tribute`), 255  
`ProtocolAdapter` (`class in pymea.adapters`), 66  
`NPS120_10` (`class in pymea-`  
`sure.instruments.oxfordinstruments`), 402  
`NPSChannel` (`class in pymea-`  
`sure.instruments.keithley.keithley2200`), 338  
`psu_temperature` (`pymea-`  
`sure.instruments.novanta.Fpu60` `property`), 393  
`pulse_dutycycle` (`pymea-`  
`sure.instruments.agilent.Agilent33220A`  
`property`), 171  
`pulse_dutycycle` (`pymea-`  
`sure.instruments.agilent.Agilent33500` `prop-`  
`erty`), 174  
`pulse_dutycycle` (`pymea-`  
`sure.instruments.agilent.agilent33500.Agilent33500Channel`  
`property`), 178  
`pulse_frequency` (`pymea-`  
`sure.instruments.agilent.Agilent8257D` `prop-`  
`erty`), 154  
`pulse_hold` (`pymea.instruments.agilent.Agilent33220A`  
`property`), 171  
`pulse_hold` (`pymea.instruments.agilent.Agilent33500`  
`property`), 174  
`pulse_hold` (`pymea.instruments.agilent.agilent33500.Agilent33500Ch`  
`property`), 178  
`pulse_input` (`pymea.instruments.agilent.Agilent8257D`  
`property`), 154  
`pulse_params` (`pymea-`  
`sure.instruments.tcpowerconversion.CXN`  
`property`), 454  
`pulse_period` (`pymea-`  
`sure.instruments.agilent.Agilent33220A`  
`property`), 171  
`pulse_period` (`pymea-`  
`sure.instruments.agilent.Agilent33500` `prop-`  
`erty`), 175  
`pulse_period` (`pymea-`  
`sure.instruments.agilent.agilent33500.Agilent33500Channel`  
`property`), 175

*property*), 178

`pulse_source` (*pymeasure.instruments.agilent.Agilent8257D* *property*), 154

`pulse_transition` (*pymeasure.instruments.agilent.Agilent33220A* *property*), 171

`pulse_transition` (*pymeasure.instruments.agilent.Agilent33500* *property*), 175

`pulse_transition` (*pymeasure.instruments.agilent.agilent33500.Agilent33500* *property*), 178

`pulse_width` (*pymeasure.instruments.agilent.Agilent33220A* *property*), 171

`pulse_width` (*pymeasure.instruments.agilent.Agilent33500* *property*), 175

`pulse_width` (*pymeasure.instruments.agilent.agilent33500.Agilent33500* *property*), 178

`pulse_width` (*pymeasure.instruments.hp.HP8116A* *property*), 249

`pymeasure.display.browser`  
module, 83

`pymeasure.display.console`  
module, 83

`pymeasure.display.curves`  
module, 84

`pymeasure.display.inputs`  
module, 85

`pymeasure.display.listeners`  
module, 87

`pymeasure.display.log`  
module, 87

`pymeasure.display.manager`  
module, 87

`pymeasure.display.plotter`  
module, 89

`pymeasure.display.thread`  
module, 89

`pymeasure.display.widgets.browser_widget`  
module, 90

`pymeasure.display.widgets.directory_widget`  
module, 90

`pymeasure.display.widgets.dock_widget`  
module, 95

`pymeasure.display.widgets.estimator_widget`  
module, 90

`pymeasure.display.widgets.image_frame`  
module, 90

`pymeasure.display.widgets.image_widget`  
module, 90

`pymeasure.display.widgets.inputs_widget`  
module, 91

`pymeasure.display.widgets.log_widget`  
module, 91

`pymeasure.display.widgets.plot_frame`  
module, 91

`pymeasure.display.widgets.plot_widget`  
module, 92

`pymeasure.display.widgets.results_dialog`  
module, 92

`pymeasure.display.widgets.sequencer_widget`  
module, 92

`pymeasure.display.widgets.tab_widget`  
module, 94

`pymeasure.display.widgets.table_widget`  
module, 95

`pymeasure.display.windows.managed_dock_window`  
module, 101

`pymeasure.display.windows.managed_image_window`  
module, 98

`pymeasure.display.windows.managed_window`  
module, 98

`pymeasure.display.windows.plotter_window`  
module, 100

`pymeasure.experiment.experiment`  
module, 71

`pymeasure.experiment.listeners`  
module, 72

`pymeasure.experiment.parameters`  
module, 74

`pymeasure.experiment.procedure`  
module, 73

`pymeasure.experiment.results`  
module, 79

`pymeasure.experiment.workers`  
module, 79

`pymeasure.instruments`  
module, 101

`pymeasure.instruments.activetechnologies`  
module, 116

`pymeasure.instruments.advantest`  
module, 121

`pymeasure.instruments.advantest.advantestR3767CG`  
module, 122

`pymeasure.instruments.advantest.advantestR624X`  
module, 143

`pymeasure.instruments.agilent`  
module, 152

`pymeasure.instruments.agilent.agilent4156`  
module, 163

`pymeasure.instruments.agilent.agilentB1500`  
module, 193

`pymeasure.instruments.aja`  
module, 196

`pymeasure.instruments.ametek`  
module, 198

`pymeasure.instruments.ami`

[module](#), 200  
[pymeasure.instruments.anaheimautomation](#)  
[module](#), 202  
[pymeasure.instruments.anapico](#)  
[module](#), 204  
[pymeasure.instruments.andeenhagerling](#)  
[module](#), 205  
[pymeasure.instruments.anritsu](#)  
[module](#), 208  
[pymeasure.instruments.attocube](#)  
[module](#), 223  
[pymeasure.instruments.bkprecision](#)  
[module](#), 226  
[pymeasure.instruments.comedi](#)  
[module](#), 116  
[pymeasure.instruments.danfysik](#)  
[module](#), 226  
[pymeasure.instruments.deltaelektronika](#)  
[module](#), 229  
[pymeasure.instruments.edwards](#)  
[module](#), 231  
[pymeasure.instruments.eurotest](#)  
[module](#), 231  
[pymeasure.instruments.fluke](#)  
[module](#), 233  
[pymeasure.instruments.fwbell](#)  
[module](#), 234  
[pymeasure.instruments.hcp](#)  
[module](#), 237  
[pymeasure.instruments.heidenhain](#)  
[module](#), 237  
[pymeasure.instruments.hp](#)  
[module](#), 239  
[pymeasure.instruments.ipgphotonics](#)  
[module](#), 286  
[pymeasure.instruments.keithley](#)  
[module](#), 287  
[pymeasure.instruments.keysight](#)  
[module](#), 338  
[pymeasure.instruments.lakeshore](#)  
[module](#), 351  
[pymeasure.instruments.lecroy](#)  
[module](#), 360  
[pymeasure.instruments.mksinst](#)  
[module](#), 375  
[pymeasure.instruments.newport](#)  
[module](#), 376  
[pymeasure.instruments.ni](#)  
[module](#), 378  
[pymeasure.instruments.novanta](#)  
[module](#), 392  
[pymeasure.instruments.oxfordinstruments](#)  
[module](#), 393  
[pymeasure.instruments.parker](#)

[module](#), 403  
[pymeasure.instruments.pendulum](#)  
[module](#), 404  
[pymeasure.instruments.razorbill](#)  
[module](#), 405  
[pymeasure.instruments.rohdeschwarz](#)  
[module](#), 406  
[pymeasure.instruments.siglenttechnologies](#)  
[module](#), 423  
[pymeasure.instruments.signalrecovery](#)  
[module](#), 427  
[pymeasure.instruments.srs](#)  
[module](#), 440  
[pymeasure.instruments.tcpowerconversion](#)  
[module](#), 452  
[pymeasure.instruments.tdk](#)  
[module](#), 456  
[pymeasure.instruments.tektronix](#)  
[module](#), 464  
[pymeasure.instruments.teledyne](#)  
[module](#), 465  
[pymeasure.instruments.temptronic](#)  
[module](#), 476  
[pymeasure.instruments.texio](#)  
[module](#), 484  
[pymeasure.instruments.thermotron](#)  
[module](#), 487  
[pymeasure.instruments.thorlabs](#)  
[module](#), 488  
[pymeasure.instruments.thyracont](#)  
[module](#), 489  
[pymeasure.instruments.toptica](#)  
[module](#), 494  
[pymeasure.instruments.validators](#)  
[module](#), 113  
[pymeasure.instruments.velleman](#)  
[module](#), 497  
[pymeasure.instruments.yokogawa](#)  
[module](#), 499  
[pymeasure.test](#)  
[module](#), 66

## Q

[QListener](#) (*class in pymeasure.display.listeners*), 87

[query\\_ac\\_current\(\)](#) (*pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method*), 381

[query\\_acquisition\\_status\(\)](#) (*pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method*), 387

[query\\_adc\\_setup\(\)](#) (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500 method*), 185

<code>query_analog_channel()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_line_configuration()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput module method), 380</i>	<code>query_line_configuration()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput module method), 380</i>
<code>query_analog_channel_characteristics()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_meas_mode()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 185</i>	<code>query_meas_mode()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 185</i>
<code>query_analog_edge_trigger()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_meas_op_mode()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 188</i>	<code>query_meas_op_mode()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 188</i>
<code>query_analog_pulse_width_trigger()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_meas_range_current_auto()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 187</i>	<code>query_meas_range_current_auto()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 187</i>
<code>query_arbitrary_waveform()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 383</i>	<code>query_meas_ranges()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 188</i>	<code>query_meas_ranges()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 188</i>
<code>query_arbitrary_waveform_gain_and_offset()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 383</i>	<code>query_meas_settings()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 185</i>	<code>query_meas_settings()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 185</i>
<code>query_current_output()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply module method), 389</i>	<code>query_measurement()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 381</i>	<code>query_measurement()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 381</i>
<code>query_dc_current()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 381</i>	<code>query_modules()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 183</i>	<code>query_modules()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 183</i>
<code>query_dc_voltage()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 381</i>	<code>query_sampling_settings()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 187</i>	<code>query_sampling_settings()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 187</i>
<code>query_enabled_analog_channels()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_series_resistor()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 187</i>	<code>query_series_resistor()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 187</i>
<code>query_event_status_register()</code> <i>(pymeasure.instruments.anritsu.AnritsuMS464xB module method), 219</i>	<code>query_staircase_sweep_settings()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter module method), 186</i>	<code>query_staircase_sweep_settings()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 186</i>
<code>query_export_signal()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput module method), 380</i>	<code>query_standard_waveform()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 384</i>	<code>query_standard_waveform()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 384</i>
<code>query_generation_status()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 384</i>	<code>query_time_stamp_setting()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 186</i>	<code>query_time_stamp_setting()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 186</i>
<code>query_learn()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 183</i>	<code>query_timing()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>	<code>query_timing()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 387</i>
<code>query_learn()</code> <i>(pymeasure.instruments.agilent.agilentB1500.QueryLearn static method), 191</i>	<code>query_trigger_delay()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 388</i>	<code>query_trigger_delay()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 388</i>
<code>query_learn()</code> <i>(pymeasure.instruments.agilent.agilentB1500.SMU module method), 188</i>	<code>query_trigger_type()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 388</i>	<code>query_trigger_type()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 388</i>
<code>query_learn_header()</code> <i>(pymeasure.instruments.agilent.agilentB1500.AgilentB1500 module method), 183</i>	<code>query_voltage_output()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply module method), 389</i>	<code>query_voltage_output()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply module method), 389</i>
<code>query_learn_header()</code> <i>(pymeasure.instruments.agilent.agilentB1500.QueryLearn class method), 191</i>	<code>query_waveform_mode()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 384</i>	<code>query_waveform_mode()</code> <i>(pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator module method), 384</i>

QueryLearn (class in *pymeasure.instruments.agilent.agilentB1500*), 191  
 questionable\_event\_enabled (*pymeasure.instruments.keithley.Keithley6221* property), 322  
 questionable\_event\_reg (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 414  
 questionable\_events (*pymeasure.instruments.keithley.Keithley6221* property), 322  
 questionable\_operation\_enable\_reg (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 414  
 questionanble\_status\_reg (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 414  
 queue() (*pymeasure.display.manager.BaseManager* method), 87  
 queue() (*pymeasure.display.windows.managed\_window.ManagedWindowBase* method), 100  
 queue\_sequence() (*pymeasure.display.widgets.sequencer\_widget.SequencerWidget* method), 94  
 quick\_range() (*pymeasure.instruments.srs.SR830* method), 445

## R

R75\_out (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 407  
 Ramp (*pymeasure.instruments.hp.hp856Xx.SweepOut* attribute), 280  
 ramp() (*pymeasure.instruments.ami.AMI430* method), 201  
 ramp\_rate (*pymeasure.instruments.tcpowerconversion.CXN* property), 454  
 ramp\_rate (*pymeasure.instruments.temptronic.ATSB* property), 480  
 ramp\_rate\_current (*pymeasure.instruments.ami.AMI430* property), 201  
 ramp\_rate\_field (*pymeasure.instruments.ami.AMI430* property), 201  
 ramp\_source() (*pymeasure.instruments.agilent.agilentB1500.SMU* method), 189  
 ramp\_start\_power (*pymeasure.instruments.tcpowerconversion.CXN* property), 454  
 ramp\_symmetry (*pymeasure.instruments.agilent.Agilent33220A* property), 171  
 ramp\_symmetry (*pymeasure.instruments.agilent.Agilent33500* property), 175  
 ramp\_symmetry (*pymeasure.instruments.agilent.agilent33500.Agilent33500Channel* property), 178  
 ramp\_time (*pymeasure.instruments.aja.DCXS* property), 197  
 ramp\_to\_current() (*pymeasure.instruments.ami.AMI430* method), 201  
 ramp\_to\_current() (*pymeasure.instruments.danfysik.Danfysik8500* method), 228  
 ramp\_to\_current() (*pymeasure.instruments.deltaelektronika.SM7045D* method), 230  
 ramp\_to\_current() (*pymeasure.instruments.keithley.Keithley2400* method), 304  
 ramp\_to\_current() (*pymeasure.instruments.keithley.Keithley2450* method), 311  
 ramp\_to\_current() (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* method), 458  
 ramp\_to\_current() (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* method), 462  
 ramp\_to\_current() (*pymeasure.instruments.yokogawa.Yokogawa7651* method), 500  
 ramp\_to\_field() (*pymeasure.instruments.ami.AMI430* method), 201  
 ramp\_to\_voltage() (*pymeasure.instruments.keithley.Keithley2400* method), 304  
 ramp\_to\_voltage() (*pymeasure.instruments.keithley.Keithley2450* method), 312  
 ramp\_to\_voltage() (*pymeasure.instruments.keithley.Keithley6517B* method), 328  
 ramp\_to\_voltage() (*pymeasure.instruments.yokogawa.Yokogawa7651* method), 500  
 ramp\_to\_zero() (*pymeasure.instruments.deltaelektronika.SM7045D* method), 230  
 ramp\_to\_zero() (*pymeasure.instruments.eurotest.EurotestHPP120256* method), 233  
 range (*pymeasure.instruments.fwbell.FWBell5080* property), 236  
 range (*pymeasure.instruments.hp.HP3437A* property), 244  
 range (*pymeasure.instruments.hp.HP3478A* property), 244

247

`range` (`pymeasure.instruments.lakeshore.LakeShore425` property), 359

`range` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreBase` property), 360

`range` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` property), 493

`range_` (`pymeasure.instruments.hp.HP34401A` property), 242

`Ranging` (class in `pymeasure.instruments.agilent.agilentB1500`), 191

`ratio` (`pymeasure.instruments.agilent.agilent4156.VARD` property), 168

`ratio` (`pymeasure.instruments.signalrecovery.DSP7225` property), 431

`ratio` (`pymeasure.instruments.signalrecovery.DSP7265` property), 438

`razorbillRP100` (class in `pymeasure.instruments.razorbill`), 405

`read`() (`pymeasure.adapters.Adapter` method), 48

`read`() (`pymeasure.adapters.FakeAdapter` method), 68

`read`() (`pymeasure.adapters.PrologixAdapter` method), 58

`read`() (`pymeasure.adapters.SerialAdapter` method), 54

`read`() (`pymeasure.adapters.TelnetAdapter` method), 64

`read`() (`pymeasure.adapters.VISAAdapter` method), 51

`read`() (`pymeasure.adapters.VXI11Adapter` method), 61

`read`() (`pymeasure.instruments.aja.DCXS` method), 197

`read`() (`pymeasure.instruments.andeenhagerling.AH2700A` method), 207

`read`() (`pymeasure.instruments.attocube.anc300.ANC300Controller` method), 224

`read`() (`pymeasure.instruments.Channel` method), 112

`read`() (`pymeasure.instruments.danfysik.Danfysik8500` method), 228

`read`() (`pymeasure.instruments.fluke.Fluke7341` method), 234

`read`() (`pymeasure.instruments.fwbell.FWBell5080` method), 236

`read`() (`pymeasure.instruments.hcp.TC038` method), 238

`read`() (`pymeasure.instruments.hcp.TC038D` method), 239

`read`() (`pymeasure.instruments.Instrument` method), 110

`read`() (`pymeasure.instruments.ipgphotonics.yar.YAR` method), 287

`read`() (`pymeasure.instruments.keithley.Keithley2000` method), 291

`read`() (`pymeasure.instruments.keithley.Keithley2260B` method), 296

`read`() (`pymeasure.instruments.keithley.Keithley2306` method), 298

`read`() (`pymeasure.instruments.keithley.Keithley2400` method), 304

`read`() (`pymeasure.instruments.keithley.Keithley2450` method), 312

`read`() (`pymeasure.instruments.keithley.Keithley2600` method), 333

`read`() (`pymeasure.instruments.keithley.Keithley2700` method), 317

`read`() (`pymeasure.instruments.keithley.Keithley2750` method), 331

`read`() (`pymeasure.instruments.keithley.Keithley6221` method), 322

`read`() (`pymeasure.instruments.keithley.Keithley6517B` method), 328

`read`() (`pymeasure.instruments.keysight.KeysightE36312A` method), 349

`read`() (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` method), 369

`read`() (`pymeasure.instruments.mksinst.mks937b.MKS937B` method), 376

`read`() (`pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInput` method), 380

`read`() (`pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMulti` method), 382

`read`() (`pymeasure.instruments.parker.ParkerGV6` method), 404

`read`() (`pymeasure.instruments.signalrecovery.DSP7225` method), 431

`read`() (`pymeasure.instruments.signalrecovery.DSP7265` method), 438

`read`() (`pymeasure.instruments.tcpowerconversion.CXN` method), 454

`read`() (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` method), 458

`read`() (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` method), 463

`read`() (`pymeasure.instruments.teledyne.TeledyneT3AFG` method), 466

`read`() (`pymeasure.instruments.texio.TexioPSW360L30` method), 486

`read`() (`pymeasure.instruments.thyracont.smartline_v1.SmartlineV1` method), 490

`read`() (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 493

`read`() (`pymeasure.instruments.toptica.ibeamsmart.IBeamSmart` method), 496

`read`() (`pymeasure.instruments.velleman.VellemanK8090` method), 498

`read_analog_digital_dataframe`() (`pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscill` method), 388

`read_analog_digital_u64`() (`pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscill` method), 388

`read_binary_values`() (`pymeasure.adapters.Adapter` method), 48

`read_binary_values`() (`pymeasure`

- sure.adapters.FakeAdapter* method), 68
- `read_binary_values()` (*pymea-  
sure.adapters.PrologixAdapter* method), 59
- `read_binary_values()` (*pymea-  
sure.adapters.SerialAdapter* method), 54
- `read_binary_values()` (*pymea-  
sure.adapters.TelnetAdapter* method), 64
- `read_binary_values()` (*pymea-  
sure.adapters.VISAAdapter* method), 51
- `read_binary_values()` (*pymea-  
sure.adapters.VXIIAdapter* method), 62
- `read_binary_values()` (*pymea-  
sure.instruments.andenhagerling.AH2700A* method), 207
- `read_binary_values()` (*pymea-  
sure.instruments.Channel* method), 112
- `read_binary_values()` (*pymea-  
sure.instruments.fwbell.FWBell5080* method), 236
- `read_binary_values()` (*pymea-  
sure.instruments.Instrument* method), 110
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2000* method), 292
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2200* method), 337
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2260B* method), 296
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2306* method), 299
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2400* method), 304
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2450* method), 312
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2600* method), 333
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2700* method), 317
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley2750* method), 331
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley6221* method), 322
- `read_binary_values()` (*pymea-  
sure.instruments.keithley.Keithley6517B* method), 328
- `read_binary_values()` (*pymea-  
sure.instruments.keysight.KeysightE36312A* method), 349
- `read_binary_values()` (*pymea-  
sure.instruments.lecroy.LeCroyT3DSO1204* method), 369
- `read_binary_values()` (*pymea-  
sure.instruments.signalrecovery.DSP7225* method), 431
- `read_binary_values()` (*pymea-  
sure.instruments.signalrecovery.DSP7265* method), 438
- `read_binary_values()` (*pymea-  
sure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* method), 458
- `read_binary_values()` (*pymea-  
sure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* method), 463
- `read_binary_values()` (*pymea-  
sure.instruments.teledyne.TeledyneT3AFG* method), 466
- `read_binary_values()` (*pymea-  
sure.instruments.texio.TexioPSW360L30* method), 486
- `read_buffer()` (*pymea-  
sure.instruments.pendulum.cnt91.CNT91* method), 405
- `read_bytes()` (*pymea-  
sure.adapters.Adapter* method), 48
- `read_bytes()` (*pymea-  
sure.adapters.FakeAdapter* method), 68
- `read_bytes()` (*pymea-  
sure.adapters.PrologixAdapter* method), 59
- `read_bytes()` (*pymea-  
sure.adapters.SerialAdapter* method), 54
- `read_bytes()` (*pymea-  
sure.adapters.TelnetAdapter* method), 65
- `read_bytes()` (*pymea-  
sure.adapters.VISAAdapter* method), 51
- `read_bytes()` (*pymea-  
sure.adapters.VXIIAdapter* method), 62
- `read_bytes()` (*pymea-  
sure.instruments.andenhagerling.AH2700A* method), 207
- `read_bytes()` (*pymea-  
sure.instruments.Channel* method), 112
- `read_bytes()` (*pymea-  
sure.instruments.fwbell.FWBell5080* method), 236
- `read_bytes()` (*pymea-  
sure.instruments.Instrument* method), 110
- `read_bytes()` (*pymea-  
sure.instruments.keithley.Keithley2000* method), 292

- method*), 292
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2200* *method*), 337
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2260B* *method*), 296
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2306* *method*), 299
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2400* *method*), 304
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2450* *method*), 312
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2600* *method*), 333
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2700* *method*), 317
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley2750* *method*), 331
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley6221* *method*), 322
- `read_bytes()` (*pymeasure.instruments.keithley.Keithley6517B* *method*), 328
- `read_bytes()` (*pymeasure.instruments.keysight.KeysightE36312A* *method*), 349
- `read_bytes()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* *method*), 369
- `read_bytes()` (*pymeasure.instruments.signalrecovery.DSP7225* *method*), 431
- `read_bytes()` (*pymeasure.instruments.signalrecovery.DSP7265* *method*), 438
- `read_bytes()` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* *method*), 458
- `read_bytes()` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* *method*), 463
- `read_bytes()` (*pymeasure.instruments.teledyne.TeledyneT3AFG* *method*), 466
- `read_bytes()` (*pymeasure.instruments.texio.TexioPSW360L30* *method*), 486
- `read_channels()` (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500* *method*), 187
- `read_data()` (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500* *method*), 187
- `read_data()` (*pymeasure.instruments.hp.HP3437A* *method*), 244
- `read_datafile()` (*pymeasure.instruments.anritsu.AnritsuMS464xB* *method*), 219
- `read_measurement()` (*pymeasure.instruments.advantest.advantestR624X.AdvantestR624X* *method*), 132
- `read_measurement_from_addr()` (*pymeasure.instruments.advantest.advantestR624X.SMUChannel* *method*), 141
- `read_memory()` (*pymeasure.instruments.anritsu.AnritsuMS9710C* *method*), 210
- `read_output()` (*pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply* *method*), 389
- `read_random_memory()` (*pymeasure.instruments.advantest.advantestR624X.SMUChannel* *method*), 140
- `read_raw()` (*pymeasure.adapters.VXI11Adapter* *method*), 62
- `read_trace()` (*pymeasure.instruments.rohdeschwarz.fsl.FSL* *method*), 421
- `readAI()` (in module *pymeasure.instruments.comedi*), 116
- `reading` (*pymeasure.instruments.hp.HP34401A* *property*), 242
- `recall()` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* *method*), 459
- `recall()` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* *method*), 463
- `recall_config()` (*pymeasure.instruments.siglentechnologies.siglent\_spdbase.SPDBase* *method*), 424
- `recall_open_short_average()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* *method*), 271
- `recall_state()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* *method*), 254
- `recall_thru()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* *method*), 272
- `recall_trace()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* *method*), 264

receive() (pymeasure.experiment.listeners.Listener method), 72  
 Recorder (class in pymeasure.experiment.listeners), 72  
 reference (pymeasure.instruments.signalrecovery.DSP7225 property), 431  
 reference (pymeasure.instruments.signalrecovery.DSP7265 property), 438  
 reference\_externalinput (pymeasure.instruments.srs.SR860 property), 449  
 reference\_level (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 255  
 reference\_level\_calibration (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 254  
 reference\_offset (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 255  
 reference\_output (pymeasure.instruments.anapico.APSIN12G property), 205  
 reference\_phase (pymeasure.instruments.signalrecovery.DSP7225 property), 431  
 reference\_phase (pymeasure.instruments.signalrecovery.DSP7265 property), 438  
 reference\_source (pymeasure.instruments.srs.SR830 property), 445  
 reference\_source (pymeasure.instruments.srs.SR860 property), 449  
 reference\_source\_trigger (pymeasure.instruments.srs.SR830 property), 445  
 reference\_triggermode (pymeasure.instruments.srs.SR860 property), 449  
 refresh\_parameters() (pymeasure.experiment.procedure.Procedure method), 74  
 regulation\_mode (pymeasure.instruments.aja.DCXS property), 197  
 relative\_field (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_field\_raw (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_mode\_enabled (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_multiplier (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_setpoint (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_setpoint\_multiplier (pymeasure.instruments.lakeshore.LakeShore421 property), 357  
 relative\_setpoint\_raw (pymeasure.instruments.lakeshore.LakeShore421 property), 358  
 relay() (pymeasure.instruments.keithley.Keithley2306 method), 299  
 relay\_mode (pymeasure.instruments.advantest.advantestR624X.SMUChannel property), 142  
 release\_control() (pymeasure.instruments.tcpowerconversion.CXN method), 454  
 reload() (pymeasure.experiment.results.Results method), 80  
 remaining\_deposition\_time\_min (pymeasure.instruments.aja.DCXS property), 197  
 remaining\_deposition\_time\_sec (pymeasure.instruments.aja.DCXS property), 197  
 remote (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property), 459  
 remote (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property), 463  
 remote() (pymeasure.instruments.danfysik.Danfysik8500 method), 228  
 remote() (pymeasure.instruments.keithley.Keithley2000 method), 292  
 remote\_interfaces (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 414  
 remote\_local\_state (pymeasure.instruments.agilent.Agilent33220A property), 171  
 remote\_lock() (pymeasure.instruments.keithley.Keithley2000 method), 292  
 remote\_mode (pymeasure.instruments.temptronic.ATSBBase property), 480  
 remote\_trigger\_type (pymeasure.instruments.anritsu.AnritsuMS464xB property), 219  
 remove() (pymeasure.display.manager.BaseManager method), 88  
 remove() (pymeasure.display.manager.Manager method), 88  
 remove() (pymeasure.display.widgets.image\_widget.ImageWidget method), 91  
 remove() (pymeasure.display.widgets.plot\_widget.PlotWidget method), 92  
 remove() (pymeasure.display.widgets.tab\_widget.TabWidget method), 95  
 remove() (pymeasure.display.widgets.table\_widget.TableWidget method), 98

- `remove_child()` (*pymeasure.instruments.common\_base.CommonBase* method), 108
- `remove_child()` (*pymeasure.instruments.keithley.Keithley2200* method), 337
- `remove_child()` (*pymeasure.instruments.keysight.KeysightE36312A* method), 349
- `remove_child()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 369
- `remove_file()` (*pymeasure.instruments.activetechnologies.AWG401x\_AWG* method), 118
- `remove_node()` (*pymeasure.display.widgets.sequencer\_widget.SequencerWidget* method), 94
- `repeat` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* property), 459
- `repeat` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* property), 463
- `repeat_sweep()` (*pymeasure.instruments.anritsu.AnritsuMS9740A* method), 211
- `repetition_rate` (*pymeasure.instruments.hp.HP8116A* property), 250
- `repetitions` (*pymeasure.instruments.rohdeschwarz.hmp.HMP4040* property), 422
- `replace_placeholders()` (in module *pymeasure.experiment.results*), 80
- `request_control()` (*pymeasure.instruments.tcpowerconversion.CXN* method), 454
- `request_service()` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* method), 253
- `request_service_conditions` (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 254
- `res_bandwidth` (*pymeasure.instruments.rohdeschwarz.fsl.FSL* property), 421
- `reset()` (*pymeasure.adapters.PrologixAdapter* method), 59
- `reset()` (*pymeasure.instruments.activetechnologies.AWG401x\_AWG* method), 118
- `reset()` (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500* method), 183
- `reset()` (*pymeasure.instruments.andeenhagerling.AH2700A* method), 207
- `reset()` (*pymeasure.instruments.fwbell.FWBell5080* method), 236
- `reset()` (*pymeasure.instruments.hp.HP8116A* method), 250
- `reset()` (*pymeasure.instruments.hp.HP8657B* method), 283
- `reset()` (*pymeasure.instruments.hp.HPLegacyInstrument* method), 283
- `reset()` (*pymeasure.instruments.Instrument* method), 110
- `reset()` (*pymeasure.instruments.keithley.Keithley2000* method), 292
- `reset()` (*pymeasure.instruments.keithley.Keithley2200* method), 337
- `reset()` (*pymeasure.instruments.keithley.Keithley2260B* method), 296
- `reset()` (*pymeasure.instruments.keithley.Keithley2306* method), 299
- `reset()` (*pymeasure.instruments.keithley.Keithley2400* method), 305
- `reset()` (*pymeasure.instruments.keithley.Keithley2450* method), 312
- `reset()` (*pymeasure.instruments.keithley.Keithley2600* method), 333
- `reset()` (*pymeasure.instruments.keithley.Keithley2700* method), 318
- `reset()` (*pymeasure.instruments.keithley.Keithley2750* method), 331
- `reset()` (*pymeasure.instruments.keithley.Keithley6221* method), 322
- `reset()` (*pymeasure.instruments.keithley.Keithley6517B* method), 328
- `reset()` (*pymeasure.instruments.keysight.KeysightE36312A* method), 349
- `reset()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 369
- `reset()` (*pymeasure.instruments.parker.ParkerGV6* method), 404
- `reset()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 431
- `reset()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 438
- `reset()` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* method), 459
- `reset()` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* method), 463
- `reset()` (*pymeasure.instruments.teledyne.TeledyneT3AFG* method), 467
- `reset()` (*pymeasure.instruments.waveform\_generation.WaveformGenerator* method), 480
- `reset()` (*pymeasure.instruments.temptronic.ATSBBase* method), 480
- `reset()` (*pymeasure.instruments.texio.TexioPSW360L30* method), 486
- `reset_alarm()` (*pymeasure.instruments.lakeshore.LakeShore211* method), 353
- `reset_buffer()` (*pymeasure.instruments.rohdeschwarz.fsl.FSL* method), 421

`sure.instruments.keithley.Keithley2000`  
`method)`, 292

`reset_buffer()` (`pymea-`  
`sure.instruments.keithley.Keithley2400`  
`method)`, 305

`reset_buffer()` (`pymea-`  
`sure.instruments.keithley.Keithley2450`  
`method)`, 312

`reset_buffer()` (`pymea-`  
`sure.instruments.keithley.Keithley2700`  
`method)`, 318

`reset_buffer()` (`pymea-`  
`sure.instruments.keithley.Keithley6221`  
`method)`, 322

`reset_buffer()` (`pymea-`  
`sure.instruments.keithley.Keithley6517B`  
`method)`, 328

`reset_instrument()` (`pymea-`  
`sure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput`  
`method)`, 380

`reset_instrument()` (`pymea-`  
`sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter`  
`method)`, 382

`reset_instrument()` (`pymea-`  
`sure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator`  
`method)`, 384

`reset_instrument()` (`pymea-`  
`sure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope`  
`method)`, 388

`reset_instrument()` (`pymea-`  
`sure.instruments.ni.virtualbench.VirtualBench.PowerSupply`  
`method)`, 389

`reset_interlocks()` (`pymea-`  
`sure.instruments.danfysik.Danfysik8500`  
`method)`, 228

`reset_OVP_OCP()` (`pymea-`  
`sure.instruments.hp.HP6632A` `method)`, 285

`reset_position()` (`pymea-`  
`sure.instruments.anaheimautomation.DPSeriesMasterController`  
`method)`, 204

`resistance` (`pymeasure.instruments.agilent.Agilent34410A`  
`property)`, 159

`resistance` (`pymeasure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance` (`pymeasure.instruments.hp.HP34401A`  
`property)`, 242

`resistance` (`pymeasure.instruments.keithley.Keithley2000`  
`property)`, 292

`resistance` (`pymeasure.instruments.keithley.Keithley2400`  
`property)`, 305

`resistance` (`pymeasure.instruments.keithley.Keithley2450`  
`property)`, 312

`resistance` (`pymeasure.instruments.keithley.Keithley2450`  
`property)`, 312

`resistance` (`pymeasure.instruments.keithley.Keithley6517B`  
`property)`, 328

`resistance_4w` (`pymea-`  
`sure.instruments.agilent.Agilent34410A`  
`property)`, 159

`resistance_4w` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_4w` (`pymeasure.instruments.hp.HP34401A`  
`property)`, 242

`resistance_4w_auto_range` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_4W_digits` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_4W_nplc` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_4w_range` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_4W_range` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_4W_reference` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_4w_resolution` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_auto_range` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_digits` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_nplc` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 292

`resistance_nplc` (`pymea-`  
`sure.instruments.keithley.Keithley2400` `prop-`  
`erty)`, 305

`resistance_nplc` (`pymea-`  
`sure.instruments.keithley.Keithley2450` `prop-`  
`erty)`, 312

`resistance_nplc` (`pymea-`  
`sure.instruments.keithley.Keithley6517B`  
`property)`, 328

`resistance_range` (`pymea-`  
`sure.instruments.agilent.Agilent34450A`  
`property)`, 162

`resistance_range` (`pymea-`  
`sure.instruments.keithley.Keithley2000` `prop-`  
`erty)`, 293

`resistance_range` (`pymea-`  
`sure.instruments.keithley.Keithley2400` `prop-`  
`erty)`, 305

*sure.instruments.keithley.Keithley2400* property), 305

*resistance\_range* (*pymea-*  
*sure.instruments.keithley.Keithley2450* prop-  
erty), 312

*resistance\_range* (*pymea-*  
*sure.instruments.keithley.Keithley6517B*  
property), 328

*resistance\_reference* (*pymea-*  
*sure.instruments.keithley.Keithley2000* prop-  
erty), 293

*resistance\_resolution* (*pymea-*  
*sure.instruments.agilent.Agilent34450A*  
property), 162

*resolution* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS9710C*  
property), 210

*resolution* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS9740A*  
property), 211

*resolution* (*pymea-*  
*sure.instruments.hp.HP34401A*  
property), 242

*resolution* (*pymea-*  
*sure.instruments.hp.HP3478A* prop-  
erty), 247

*resolution\_actual* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS9710C*  
property), 210

*resolution\_bandwidth* (*pymea-*  
*sure.instruments.hp.hp856Xx.HP856Xx* at-  
tribute), 260

*resolution\_bandwidth\_to\_span\_ratio* (*pymea-*  
*sure.instruments.hp.hp856Xx.HP856Xx* at-  
tribute), 260

*resolution\_vbw* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS9710C*  
property), 210

*resolution\_vbw* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS9740A*  
property), 211

*Results* (class in *pymea-*  
*sure.experiment.results*), 79

*ResultsClass* (*pymea-*  
*sure.display.widgets.image\_frame.ImageFrame*  
attribute), 90

*ResultsClass* (*pymea-*  
*sure.display.widgets.plot\_frame.PlotFrame*  
attribute), 91

*ResultsCurve* (class in *pymea-*  
*sure.display.curves*), 84

*ResultsDialog* (class in *pymea-*  
*sure.display.widgets.results\_dialog*), 92

*ResultsImage* (class in *pymea-*  
*sure.display.curves*), 84

*ResultsTable* (class in *pymea-*  
*sure.display.widgets.table\_widget*), 97

*resume()* (*pymea-*  
*sure.display.manager.BaseManager*  
method), 88

*return\_to\_local()* (*pymea-*  
*sure.instruments.anritsu.AnritsuMS464xB*  
method), 219

*reverse\_power\_limit* (*pymea-*  
*sure.instruments.tcpowerconversion.CXN*  
property), 454

*rf\_enabled* (*pymea-*  
*sure.instruments.tcpowerconversion.CXN*  
property), 454

*rf\_out\_enabled* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 414

*rf\_sweep\_center* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 414

*rf\_sweep\_span* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 414

*rf\_sweep\_start* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 414

*rf\_sweep\_step* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 415

*rf\_sweep\_stop* (*pymea-*  
*sure.instruments.rohdeschwarz.sfm.SFM*  
property), 415

*right\_limit* (*pymea-*  
*sure.instruments.newport.esp300.Axis*  
property), 378

*rom\_version* (*pymea-*  
*sure.instruments.hp.HP6632A*  
property), 285

*round\_up()* (*pymea-*  
*sure.display.curves.ResultsImage*  
method), 85

*rowCount()* (*pymea-*  
*sure.display.widgets.sequencer\_widget.SequencerTree*  
method), 94

*rowCount()* (*pymea-*  
*sure.display.widgets.table\_widget.PandasModelBase*  
method), 96

*RQS* (*pymea-*  
*sure.instruments.hp.hp856Xx.StatusRegister*  
attribute), 280

*rsd* (*pymea-*  
*sure.instruments.deltaelektronika.SM7045D*  
property), 231

*run()* (*pymea-*  
*sure.display.listeners.Monitor* method), 87

*run()* (*pymea-*  
*sure.display.plotter.Plotter* method), 89

*run()* (*pymea-*  
*sure.display.widgets.estimator\_widget.EstimatorThread*  
method), 90

*run()* (*pymea-*  
*sure.experiment.workers.Worker* method),  
79

*run()* (*pymea-*  
*sure.instruments.keysight.KeysightDSOX1102G*  
method), 340

*run()* (*pymea-*  
*sure.instruments.lecroy.LeCroyT3DSO1204*  
method), 369

*run()* (*pymea-*  
*sure.instruments.ni.virtualbench.VirtualBench.FunctionGene*  
method), 384

*run()* (*pymea-*  
*sure.instruments.ni.virtualbench.VirtualBench.MixedSignalO*  
method), 388

*run()* (*pymea-*  
*sure.instruments.teledyne.TeledyneOscilloscope*  
method), 469

[run\(\)](#) (`pymeasure.instruments.thermotron.Thermotron3800` property), 119  
[method](#)), 488  
[run\\_mode](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 118  
[run\\_status](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
**S**  
[Sample](#) (`pymeasure.instruments.hp.hp856Xx.DetectionModes` attribute), 278  
[sample\\_continuously\(\)](#) (`pymeasure.instruments.keithley.Keithley2400` method), 305  
[sample\\_count](#) (`pymeasure.instruments.hp.HP34401A` property), 242  
[sample\\_decreasing\\_strategy](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[sample\\_frequency](#) (`pymeasure.instruments.srs.SR830` property), 445  
[sample\\_hold\\_mode](#) (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 133  
[sample\\_increasing\\_strategy](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[SampleHold](#) (class in `pymeasure.instruments.advantest.advantestR624X`), 143  
[SampleMode](#) (class in `pymeasure.instruments.advantest.advantestR624X`), 143  
[sampler\\_harmonic\\_number](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 269  
[SAMPLING](#) (`pymeasure.instruments.agilent.agilentB1500.MeasMode` attribute), 194  
[sampling\\_auto\\_abort\(\)](#) (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 187  
[sampling\\_frequency](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 269  
[sampling\\_mode](#) (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` property), 187  
[sampling\\_points](#) (`pymeasure.instruments.anritsu.AnritsuMS9710C` property), 210  
[sampling\\_points](#) (`pymeasure.instruments.anritsu.AnritsuMS9740A` property), 211  
[sampling\\_rate](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[sampling\\_rate\\_max](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[sampling\\_rate\\_min](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[sampling\\_source\(\)](#) (`pymeasure.instruments.agilent.agilentB1500.SMU` method), 190  
[sampling\\_timing\(\)](#) (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 187  
[SamplingMode](#) (class in `pymeasure.instruments.agilent.agilentB1500`), 194  
[SamplingPostOutput](#) (class in `pymeasure.instruments.agilent.agilentB1500`), 195  
[save\(\)](#) (`pymeasure.instruments.agilent.agilent4156.Agilent4156` method), 166  
[save\(\)](#) (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` method), 459  
[save\(\)](#) (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` method), 463  
[save\\_config\(\)](#) (`pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDBase` method), 424  
[save\\_dock\\_layout\(\)](#) (`pymeasure.display.widgets.dock_widget.DockWidget` method), 95  
[save\\_file\(\)](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` method), 119  
[save\\_sequence\(\)](#) (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 422  
[save\\_state\(\)](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 254  
[save\\_trace\(\)](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 264  
[save\\_var\(\)](#) (`pymeasure.instruments.agilent.agilent4156.Agilent4156` method), 166  
[scale](#) (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope` property), 474  
[scale\\_volt](#) (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 415  
[scan\(\)](#) (`pymeasure.instruments.agilent.Agilent8722ES` method), 156  
[scan\\_continuous\(\)](#) (`pymeasure.instruments.agilent.Agilent8722ES` method), 156  
[scan\\_points](#) (`pymeasure.instruments.agilent.Agilent8722ES` property), 156  
[scan\\_single\(\)](#) (`pymeasure.instruments.agilent.Agilent8722ES` method), 156

`sure.instruments.agilent.Agilent8722ES`  
(method), 156

`ScientificInput` (class in `pymeasure.display.inputs`), 86

`scpi_version` (`pymeasure.instruments.hp.HP34401A` property), 242

`screen_layout` (`pymeasure.instruments.srs.SR860` property), 449

`screenshot()` (`pymeasure.instruments.srs.SR860` method), 449

`search_peak()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 266

`select_for_output()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 134

`selected_channel` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` property), 422

`selected_channel` (`pymeasure.instruments.siglenttechnologies.siglent_spdbase.SPDBase` property), 424

`selected_channel_active` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` property), 422

`self_calibrate()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.Firmware` method), 384

`self_test` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 131

`self_test_result` (`pymeasure.instruments.hp.HP34401A` property), 242

`send_trigger()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 184

`sense_mode` (`pymeasure.instruments.anritsu.AnritsuMS2090A` property), 214

`sensitivity` (`pymeasure.instruments.ametek.Ametek7270` property), 199

`sensitivity` (`pymeasure.instruments.signalrecovery.DSP7225` property), 431

`sensitivity` (`pymeasure.instruments.signalrecovery.DSP7265` property), 438

`sensitivity` (`pymeasure.instruments.srs.SR510` property), 440

`sensitivity` (`pymeasure.instruments.srs.SR570` property), 442

`sensitivity` (`pymeasure.instruments.srs.SR830` property), 445

`sensitvity` (`pymeasure.instruments.srs.SR860` property), 449

`sensor` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreTemperatureChannel` property), 359

`sensor_serial` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` property), 493

`sequence` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` property), 422

`sequence_program_listing()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 126

`sequence_program_number` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 126

`sequence_wait()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 125

`SequenceDialog` (class in `pymeasure.display.widgets.sequencer_widget`), 93

`SequenceInterruptType` (class in `pymeasure.instruments.advantest.advantestR624X`), 144

`SequenceBaseTreeModel` (class in `pymeasure.display.widgets.sequencer_widget`), 93

`SequencerTreeView` (class in `pymeasure.display.widgets.sequencer_widget`), 94

`SequencerWidget` (class in `pymeasure.display.widgets.sequencer_widget`), 94

`serial` (`pymeasure.instruments.mksinst.mks937b.MKS937B` property), 376

`serial` (`pymeasure.instruments.tcpowerconversion.CXN` property), 454

`serial` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` property), 459

`serial` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` property), 463

`serial` (`pymeasure.instruments.toptica.ibeamsmart.IBeamSmart` property), 496

`serial_baud` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 415

`serial_bits` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 415

`serial_flowcontrol` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 415

`serial_nr` (`pymeasure.instruments.attocube.anc300.Axis` property), 226

`serial_number` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 257

`serial_number` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreTemperatureChannel` property), 358

- `serial_number` (`pymeasure.instruments.novanta.Fpu60` property), 393
- `serial_parity` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 415
- `serial_stopbits` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 416
- `SerialAdapter` (class in `pymeasure.adapters`), 53
- `series_resistance` (`pymeasure.instruments.agilent.agilent4156.SMU` property), 167
- `series_resistor` (`pymeasure.instruments.agilent.agilentB1500.SMU` property), 188
- `service_request_enable_bits` (`pymeasure.instruments.anritsu.AnritsuMS464xB` property), 219
- `service_request_enable_register` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 130
- `SESR` (class in `pymeasure.instruments.advantest.advantestR624X`), 145
- `set_auto_couple()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 252
- `set_averaging()` (`pymeasure.instruments.agilent.Agilent8722ES` method), 156
- `set_buffer()` (`pymeasure.instruments.signalrecovery.DSP7225` method), 432
- `set_buffer()` (`pymeasure.instruments.signalrecovery.DSP7265` method), 438
- `set_channel_A_mode()` (`pymeasure.instruments.ametek.Ametek7270` method), 199
- `set_channel_state()` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` method), 422
- `set_color()` (`pymeasure.display.widgets.plot_widget.PlotWidget` method), 92
- `set_color()` (`pymeasure.display.widgets.tab_widget.TabWidget` method), 95
- `set_color()` (`pymeasure.display.widgets.table_widget.TableWidget` method), 98
- `set_comparison_limits()` (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` method), 142
- `set_continuous_sensor_transition()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 493
- `set_continuous_sweep` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 270
- `set_crt_adjustment_pattern()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 257
- `set_current_mode()` (`pymeasure.instruments.ametek.Ametek7270` method), 199
- `set_default_sensor_transition()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 493
- `set_defaults()` (`pymeasure.instruments.parker.ParkerGV6` method), 404
- `set_differential_mode()` (`pymeasure.instruments.ametek.Ametek7270` method), 199
- `set_digital_output()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 124
- `set_direct_sensor_transition()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 493
- `set_field()` (`pymeasure.instruments.oxfordinstruments.IPS120_10` method), 400
- `set_fixed_frequency()` (`pymeasure.instruments.agilent.Agilent8722ES` method), 156
- `set_full_span()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 260
- `set_fullband()` (`pymeasure.instruments.hp.HP8561B` method), 276
- `set_hardware_limits()` (`pymeasure.instruments.parker.ParkerGV6` method), 404
- `set_high()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 494
- `set_IF_bandwidth()` (`pymeasure.instruments.agilent.Agilent8722ES` method), 156
- `set_linear_scale()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 252
- `set_lo_common_connection_relay()` (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 131
- `set_low()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2` method), 494
- `set_marker_delta_to_span()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` method), 268
- `set_marker_minimum()` (`pymeasure`

- `sure.instruments.hp.hp856Xx.HP856Xx` (pymea-  
method), 267
- `set_marker_to_center_frequency()` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx`  
method), 266
- `set_marker_to_center_frequency_step_size()`  
(`pymea-  
sure.instruments.hp.hp856Xx.HP856Xx`  
method), 268
- `set_marker_to_reference_level()` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx`  
method), 268
- `set_max_over_voltage()` (pymea-  
`sure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38`  
method), 459
- `set_max_over_voltage()` (pymea-  
`sure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65`  
method), 463
- `set_maximum_hold` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 254
- `set_minimum_hold` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 254
- `set_model()` (`pymea-  
sure.display.widgets.table_widget.TableWidget`  
method), 97
- `set_monitored_quantity()` (pymea-  
`sure.instruments.hcp.TC038` method), 238
- `set_output_format()` (pymea-  
`sure.instruments.advantest.advantestR624X.AdvantestR624X`  
method), 127
- `set_output_type()` (pymea-  
`sure.instruments.advantest.advantestR624X.SMUChannel`  
method), 132
- `set_parameter()` (pymea-  
`sure.display.inputs.BooleanInput` method),  
85
- `set_parameter()` (`pymea-  
sure.display.inputs.Input`  
method), 85
- `set_parameter()` (pymea-  
`sure.display.inputs.IntegerInput` method),  
85
- `set_parameter()` (`pymea-  
sure.display.inputs.ListInput`  
method), 86
- `set_parameter()` (pymea-  
`sure.display.inputs.ScientificInput` method),  
86
- `set_parameters()` (pymea-  
`sure.display.windows.managed_window.ManagedWindowBase`  
method), 100
- `set_parameters()` (pymea-  
`sure.experiment.procedure.Procedure` method),  
74
- `set_point` (`pymea-  
sure.instruments.fluke.Fluke7341`  
property), 234
- `set_point_number` (pymea-  
`sure.instruments.temptronic.ATSBBase` prop-  
erty), 480
- `set_ramp_delay()` (pymea-  
`sure.instruments.danfysik.Danfysik8500`  
method), 228
- `set_ramp_to_current()` (pymea-  
`sure.instruments.danfysik.Danfysik8500`  
method), 228
- `set_reference_mode()` (pymea-  
`sure.instruments.ametek.Ametek7270` method),  
199
- `set_sample_mode()` (pymea-  
`sure.instruments.advantest.advantestR624X.SMUChannel`  
method), 134
- `set_scaling()` (`pymea-  
sure.instruments.srs.SR830`  
method), 445
- `set_scanner_control()` (pymea-  
`sure.instruments.advantest.advantestR624X.SMUChannel`  
method), 132
- `set_sequence()` (pymea-  
`sure.instruments.danfysik.Danfysik8500`  
method), 229
- `set_signal_identification_to_center_frequency()`  
(`pymea-  
sure.instruments.hp.HP8561B` method),  
276
- `set_software_limits()` (pymea-  
`sure.instruments.parker.ParkerGV6` method),  
404
- `set_temperature()` (pymea-  
`sure.instruments.temptronic.ATSBBase` method),  
480
- `set_timed_arm()` (pymea-  
`sure.instruments.keithley.Keithley2400`  
method), 305
- `set_timed_arm()` (pymea-  
`sure.instruments.keithley.Keithley6221`  
method), 322
- `set_timing_parameters()` (pymea-  
`sure.instruments.advantest.advantestR624X.SMUChannel`  
method), 134
- `set_title()` (`pymea-  
sure.instruments.hp.hp856Xx.HP856Xx`  
method), 253
- `set_trace_data_a` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 263
- `set_trace_data_b` (pymea-  
`sure.instruments.hp.hp856Xx.HP856Xx` at-  
tribute), 263
- `set_trigger_counts()` (pymea-  
`sure.instruments.keithley.Keithley2400`  
method), 305
- `set_voltage_mode()` (pymea-  
`sure.instruments.ametek.Ametek7270` method),

- 199
- `set_voltage_mode()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 432
- `set_voltage_mode()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 439
- `set_wire_mode()` (*pymeasure.instruments.advantest.advantestR624X.SMUChannel* method), 141
- `setChannelAMode()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 431
- `setChannelAMode()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 438
- `setData()` (*pymeasure.display.widgets.sequencer\_widget.SequencerTableModel* method), 94
- `setDifferentialMode()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 432
- `setDifferentialMode()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 438
- `setEditorData()` (*pymeasure.display.widgets.sequencer\_widget.ComboBoxDelegate* method), 92
- `setEditorData()` (*pymeasure.display.widgets.sequencer\_widget.LineEditDelegate* method), 93
- `setModel()` (*pymeasure.display.widgets.sequencer\_widget.SequencerTableModel* method), 94
- `setModel()` (*pymeasure.display.widgets.table\_widget.Table* method), 97
- `setModelData()` (*pymeasure.display.widgets.sequencer\_widget.ComboBoxDelegate* method), 92
- `setModelData()` (*pymeasure.display.widgets.sequencer\_widget.LineEditDelegate* method), 93
- `setpoint` (*pymeasure.instruments.aja.DCXS* property), 197
- `setpoint` (*pymeasure.instruments.hcp.TC038* property), 238
- `setpoint` (*pymeasure.instruments.hcp.TC038D* property), 239
- `setpoint` (*pymeasure.instruments.lakeshore.lakeshore\_base.LakeShoreChannel* property), 360
- `setpoint` (*pymeasure.instruments.tcpowerconversion.CXN* property), 454
- `setpoint` (*pymeasure.instruments.thermotron.Thermotron3800* property), 488
- `setting()` (*pymeasure.instruments.common\_base.CommonBase* static method), 108
- `setting()` (*pymeasure.instruments.keysight.KeysightE36312A* static method), 349
- `setting()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* static method), 369
- `setup()` (*pymeasure.instruments.teledyne.teledyne\_oscilloscope.TeledyneC* method), 475
- `setup_parser()` (*pymeasure.display.console.ConsoleArgumentParser* method), 83
- `setup_plot()` (*pymeasure.display.plotter.Plotter* method), 89
- SFM (class in *pymeasure.instruments.rohdeschwarz.sfm*), 407
- `shape` (*pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG* property), 120
- `shape` (*pymeasure.instruments.agilent.Agilent33220A* property), 171
- `shape` (*pymeasure.instruments.agilent.Agilent33500* property), 175
- `shape` (*pymeasure.instruments.agilent.agilent33500.Agilent33500Channel* property), 178
- `shape` (*pymeasure.instruments.hp.HP33120A* property), 240
- `shape` (*pymeasure.instruments.hp.HP8116A* property), 250
- `shield` (*pymeasure.instruments.signalrecovery.DSP7225* property), 432
- `shield` (*pymeasure.instruments.signalrecovery.DSP7265* property), 439
- `shutdown()` (*pymeasure.experiment.procedure.Procedure* method), 74
- `shutdown()` (*pymeasure.experiment.workers.Worker* method), 79
- `shutdown()` (*pymeasure.instruments.agilent.Agilent8257D* method), 154
- `shutdown()` (*pymeasure.instruments.ametek.Ametek7270* method), 199
- `shutdown()` (*pymeasure.instruments.ami.AMI430* method), 201
- `shutdown()` (*pymeasure.instruments.andeenhagerling.AH2700A* method), 207
- `shutdown()` (*pymeasure.instruments.anritsu.AnritsuMG3692C* method), 208
- `shutdown()` (*pymeasure.instruments.deltalelektronika.SM7045D* method), 231
- `shutdown()` (*pymeasure.instruments.eurotest.EurotestHPP120256* method), 236
- `shutdown()` (*pymeasure.instruments.fwbell.FWBell5080* method), 236
- `shutdown()` (*pymeasure.instruments.hp.HP8116A* method), 250
- `shutdown()` (*pymeasure.instruments.hp.HP8657B* method), 283
- `shutdown()` (*pymeasure.instruments.hp.HPLegacyInstrument* method), 283

method), 283  
shutdown() (pymeasure.instruments.Instrument method), 110  
shutdown() (pymeasure.instruments.keithley.Keithley2000 method), 293  
shutdown() (pymeasure.instruments.keithley.Keithley2200 method), 337  
shutdown() (pymeasure.instruments.keithley.Keithley2260 method), 296  
shutdown() (pymeasure.instruments.keithley.Keithley2306 method), 299  
shutdown() (pymeasure.instruments.keithley.Keithley2400 method), 305  
shutdown() (pymeasure.instruments.keithley.Keithley2450 method), 312  
shutdown() (pymeasure.instruments.keithley.Keithley2600 method), 333  
shutdown() (pymeasure.instruments.keithley.Keithley2700 method), 318  
shutdown() (pymeasure.instruments.keithley.Keithley2750 method), 332  
shutdown() (pymeasure.instruments.keithley.Keithley6221 method), 322  
shutdown() (pymeasure.instruments.keithley.Keithley6517 method), 328  
shutdown() (pymeasure.instruments.keysight.KeysightE3652 method), 350  
shutdown() (pymeasure.instruments.lakeshore.LakeShore421 method), 358  
shutdown() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 370  
shutdown() (pymeasure.instruments.newport.ESP300 method), 377  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 392  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput method), 380  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeters method), 382  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerators method), 384  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 388  
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupplies method), 390  
shutdown() (pymeasure.instruments.siglentechnologies.siglent\_spdbase method), 424  
shutdown() (pymeasure.instruments.signalrecovery.DSP7225 method), 432  
shutdown() (pymeasure.instruments.signalrecovery.DSP7265 method), 439  
shutdown() (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40 method), 459  
shutdown() (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80 method), 470  
method), 463  
shutdown() (pymeasure.instruments.teledyne.TeledyneT3AFG method), 467  
shutdown() (pymeasure.instruments.temptronic.ATSBASE method), 480  
shutdown() (pymeasure.instruments.texio.TexioPSW360L30 method), 486  
shutdown() (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart method), 496  
shutdown() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 500  
shutter\_delay (pymeasure.instruments.aja.DCXS property), 197  
shutter\_open (pymeasure.instruments.novanta.Fpu60 property), 393  
shutter\_state (pymeasure.instruments.aja.DCXS property), 197  
signal\_identification (pymeasure.instruments.hp.HP8561B property), 276  
signal\_identification\_frequency (pymeasure.instruments.hp.HP8561B property), 276  
Signal\_inverted (pymeasure.instruments.srs.SR570 property), 442  
SignalChannel (class in pymeasure.instruments.teledyne.teledyneT3AFG), 467  
sine\_amplitudepreset1 (pymeasure.instruments.srs.SR860 property), 449  
sine\_amplitudepreset2 (pymeasure.instruments.srs.SR860 property), 450  
sine\_amplitudepreset3 (pymeasure.instruments.srs.SR860 property), 450  
sine\_amplitudepreset4 (pymeasure.instruments.srs.SR860 property), 450  
sine\_dclevelpreset1 (pymeasure.instruments.srs.SR860 property), 450  
sine\_dclevelpreset2 (pymeasure.instruments.srs.SR860 property), 450  
sine\_dclevelpreset3 (pymeasure.instruments.srs.SR860 property), 450  
sine\_dclevelpreset4 (pymeasure.instruments.srs.SR860 property), 450  
sine\_voltage (pymeasure.instruments.srs.SR830 property), 450  
single() (pymeasure.instruments.keysight.KeysightDSOX1102G method), 340  
single() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 370  
single() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 470

`single_sweep()` (pymeasure.instruments.anritsu.AnritsuMS9710C method), 210  
`single_sweep()` (pymeasure.instruments.rohdeschwarz.fsl.FSL method), 421  
`sizeHint()` (pymeasure.display.widgets.image\_widget.ImageWidget method), 91  
`sizeHint()` (pymeasure.display.widgets.plot\_widget.PlotWidget method), 92  
`skew_factor` (pymeasure.instruments.lecroy.lecroyT3DSO1204.LecroyT3DSO1204Channel property), 374  
`slew_rate` (pymeasure.instruments.danfysik.Danfysik8500 property), 229  
`slew_rate_1` (pymeasure.instruments.razorbill.razorbillRP100 property), 406  
`slew_rate_2` (pymeasure.instruments.razorbill.razorbillRP100 property), 406  
`slope` (pymeasure.instruments.ametek.Ametek7270 property), 199  
`slope` (pymeasure.instruments.signalrecovery.DSP7225 property), 432  
`slope` (pymeasure.instruments.signalrecovery.DSP7265 property), 439  
`slot` (pymeasure.instruments.thorlabs.ThorlabsPro8000 property), 489  
`SM7045D` (class in pymeasure.instruments.deltaelektronika), 230  
`SmartlineV1` (class in pymeasure.instruments.thyracont.smartline\_v1), 490  
`SmartlineV2` (class in pymeasure.instruments.thyracont.smartline\_v2), 491  
`SmartlineV2.Sources` (class in pymeasure.instruments.thyracont.smartline\_v2), 492  
`SMU` (class in pymeasure.instruments.agilent.agilent4156), 166  
`SMU` (class in pymeasure.instruments.agilent.agilentB1500), 188  
`SMU_MEASUREMENT` (pymeasure.instruments.agilent.agilentB1500.WaitTimeType attribute), 196  
`smu_names` (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 property), 183  
`smu_references` (pymeasure.instruments.agilent.agilentB1500.AgilentB1500 property), 183  
`SMU_SOURCE` (pymeasure.instruments.agilent.agilentB1500.WaitTimeType attribute), 196  
`SMUChannel` (class in pymeasure.instruments.advantest.advantestR624X), 132  
`SMUCurrentRanging` (class in pymeasure.instruments.agilent.agilentB1500), 192  
`SMUVoltageRanging` (class in pymeasure.instruments.agilent.agilentB1500), 192  
`snap()` (pymeasure.instruments.srs.SR830 method), 445  
`snap()` (pymeasure.instruments.srs.SR860 method), 450  
`software_version` (pymeasure.instruments.aja.DCXS property), 198  
`software_version` (pymeasure.instruments.novanta.Fpu60 property), 397  
`Sound_Channel` (class in pymeasure.instruments.rohdeschwarz.sfm), 417  
`sound_mode` (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 416  
`source_auto_range` (pymeasure.instruments.keithley.Keithley6221 property), 322  
`source_compliance` (pymeasure.instruments.keithley.Keithley6221 property), 322  
`source_current` (pymeasure.instruments.keithley.Keithley2400 property), 305  
`source_current` (pymeasure.instruments.keithley.Keithley2450 property), 312  
`source_current` (pymeasure.instruments.keithley.Keithley6221 property), 322  
`source_current` (pymeasure.instruments.yokogawa.Yokogawa7651 property), 500  
`source_current_delay` (pymeasure.instruments.keithley.Keithley2450 property), 312  
`source_current_delay_auto` (pymeasure.instruments.keithley.Keithley2450 property), 313  
`source_current_range` (pymeasure.instruments.keithley.Keithley2400 property), 305  
`source_current_range` (pymeasure.instruments.keithley.Keithley2450 property), 313  
`source_current_range` (pymeasure.instruments.yokogawa.Yokogawa7651 property), 500  
`source_current_resistance_limit` (pymeasure.instruments.keithley.Keithley6517B property), 329  
`source_delay` (pymeasure.instruments.keithley.Keithley2400 property), 305

`source_delay` (`pymeasure.instruments.keithley.Keithley6221` property), 323

`source_delay_auto` (`pymeasure.instruments.keithley.Keithley2400` property), 305

`source_enabled` (`pymeasure.instruments.bkprecision.BKPrecision9130B` property), 226

`source_enabled` (`pymeasure.instruments.keithley.Keithley2400` property), 305

`source_enabled` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_enabled` (`pymeasure.instruments.keithley.Keithley6221` property), 323

`source_enabled` (`pymeasure.instruments.keithley.Keithley6517B` property), 329

`source_enabled` (`pymeasure.instruments.yokogawa.Yokogawa7651` property), 500

`source_enabled` (`pymeasure.instruments.yokogawa.YokogawaGS200` property), 501

`source_level` (`pymeasure.instruments.yokogawa.YokogawaGS200` property), 501

`source_leveling_control` (`pymeasure.instruments.hp.HP8560A` property), 273

`source_mode` (`pymeasure.instruments.keithley.Keithley2400` property), 305

`source_mode` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_mode` (`pymeasure.instruments.yokogawa.Yokogawa7651` property), 500

`source_mode` (`pymeasure.instruments.yokogawa.YokogawaGS200` property), 501

`source_power` (`pymeasure.instruments.hp.HP8560A` property), 273

`source_power_offset` (`pymeasure.instruments.hp.HP8560A` property), 274

`source_power_step` (`pymeasure.instruments.hp.HP8560A` property), 274

`source_power_sweep` (`pymeasure.instruments.hp.HP8560A` property), 274

`source_range` (`pymeasure.instruments.keithley.Keithley6221` property), 323

`source_range` (`pymeasure.instruments.yokogawa.YokogawaGS200` property), 501

`source_voltage` (`pymeasure.instruments.keithley.Keithley2400` property), 306

`source_voltage` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_voltage` (`pymeasure.instruments.keithley.Keithley6517B` property), 329

`source_voltage` (`pymeasure.instruments.yokogawa.Yokogawa7651` property), 500

`source_voltage_delay` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_voltage_delay_auto` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_voltage_range` (`pymeasure.instruments.keithley.Keithley2400` property), 306

`source_voltage_range` (`pymeasure.instruments.keithley.Keithley2450` property), 313

`source_voltage_range` (`pymeasure.instruments.keithley.Keithley6517B` property), 329

`source_voltage_range` (`pymeasure.instruments.yokogawa.Yokogawa7651` property), 500

`SourceLevelingControlMode` (class in `pymeasure.instruments.hp.hp856Xx`), 279

`spacing` (`pymeasure.instruments.agilent.agilent4156.VARI` property), 167

`span` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 260

`span_frequency` (`pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767C` property), 122

`SPD1168X` (class in `pymeasure.instruments.siglenttechnologies`), 426

`SPD1305X` (class in `pymeasure.instruments.siglenttechnologies`), 427

`SPDBase` (class in `pymeasure.instruments.siglenttechnologies.siglent_spdbase`), 424

`SPDChannel` (class in `pymeasure.instruments.siglenttechnologies.siglent_spdbase`), 425

`SPDSingleChannelBase` (class in `pymeasure`)

- sure.instruments.siglenttechnologies.siglent\_spdbase*), 424
- special\_channel** (*pymea-  
sure.instruments.rohdeschwarz.sfm.SFM  
property*), 416
- SpectrumAnalyzer** (*pymea-  
sure.instruments.hp.hp856Xx.SweepCoupleMode  
attribute*), 280
- SPOT** (*pymea-  
sure.instruments.agilent.agilentB1500.MeasMode  
attribute*), 194
- square\_dutycycle** (*pymea-  
sure.instruments.agilent.Agilent33220A  
property*), 171
- square\_dutycycle** (*pymea-  
sure.instruments.agilent.Agilent33500  
property*), 175
- square\_dutycycle** (*pymea-  
sure.instruments.agilent.agilent33500.Agilent33500A  
property*), 178
- squelch** (*pymea-  
sure.instruments.hp.hp856Xx.HP856Xx  
attribute*), 258
- SR510** (*class in pymea-  
sure.instruments.srs*), 440
- SR570** (*class in pymea-  
sure.instruments.srs*), 441
- SR830** (*class in pymea-  
sure.instruments.srs*), 442
- SR860** (*class in pymea-  
sure.instruments.srs*), 446
- SRER** (*class in pymea-  
sure.instruments.advantest.advantestR624X*), 145
- srq\_enabled** (*pymea-  
sure.instruments.advantest.advantestR624X  
property*), 124
- SRQ\_enabled** (*pymea-  
sure.instruments.hp.HP6632A  
property*), 284
- srq\_event\_enabled** (*pymea-  
sure.instruments.keithley.Keithley6221  
property*), 323
- SRQ\_mask** (*pymea-  
sure.instruments.hp.HP3437A  
property*), 243
- SRQ\_mask** (*pymea-  
sure.instruments.hp.HP3478A  
property*), 245
- STAIRCASE\_SWEEP** (*pymea-  
sure.instruments.agilent.agilentB1500.MeasMode  
attribute*), 194
- staircase\_sweep\_source()** (*pymea-  
sure.instruments.agilent.agilentB1500.SMU  
method*), 190
- StaircaseSweepPostOutput** (*class in pymea-  
sure.instruments.agilent.agilentB1500*), 195
- standard\_devs** (*pymea-  
sure.instruments.keithley.Keithley2400  
property*), 306
- standard\_devs** (*pymea-  
sure.instruments.keithley.Keithley2450  
property*), 313
- standard\_event\_enabled** (*pymea-  
sure.instruments.keithley.Keithley6221  
property*), 323
- standard\_events** (*pymea-  
sure.instruments.keithley.Keithley6221  
property*), 323
- standby()** (*pymea-  
sure.instruments.advantest.advantestR624X.AdvantestR624X  
method*), 124
- standby()** (*pymea-  
sure.instruments.advantest.advantestR624X.SMUChannel  
method*), 140
- start** (*pymea-  
sure.instruments.agilent.agilent4156.VARX  
property*), 168
- START** (*pymea-  
sure.instruments.agilent.agilentB1500.StaircaseSweepPostOutput  
attribute*), 195
- start()** (*pymea-  
sure.experiment.experiment.Experiment  
method*), 72
- start()** (*pymea-  
sure.instruments.temptronic.ATSBASE  
method*), 481
- start\_autovernier()** (*pymea-  
sure.instruments.hp.HP8116A  
method*), 250
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley2000  
method*), 293
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley2400  
method*), 306
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley2450  
method*), 313
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley2700  
method*), 318
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley6221  
method*), 323
- start\_buffer()** (*pymea-  
sure.instruments.keithley.Keithley6517B  
method*), 329
- start\_buffer()** (*pymea-  
sure.instruments.signalrecovery.DSP7225  
method*), 432
- start\_buffer()** (*pymea-  
sure.instruments.signalrecovery.DSP7265  
method*), 439
- start\_frequency** (*pymea-  
sure.instruments.advantest.advantestR3767CG.AdvantestR3767CG  
property*), 122
- start\_frequency** (*pymea-  
sure.instruments.agilent.Agilent8257D  
property*), 154
- start\_frequency** (*pymea-  
sure.instruments.agilent.Agilent8722ES  
property*), 156
- start\_frequency** (*pymea-  
sure.instruments.agilent.AgilentE4408B  
property*), 156

property), 156

start\_frequency (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 259

start\_power (pymeasure.instruments.agilent.Agilent8257D property), 154

start\_ramp() (pymeasure.instruments.danfysik.Danfysik8500 method), 229

start\_sequence() (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X method), 125

start\_sequence() (pymeasure.instruments.danfysik.Danfysik8500 method), 229

start\_sequence() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423

start\_sequence\_program() (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X method), 125

start\_step\_sweep() (pymeasure.instruments.agilent.Agilent8257D method), 154

startup() (pymeasure.experiment.procedure.Procedure method), 74

startup() (pymeasure.experiment.procedure.UnknownProcedure method), 74

state (pymeasure.instruments.ami.AMI430 property), 201

status (pymeasure.instruments.agilent.agilentB1500.SMU property), 188

status (pymeasure.instruments.andeenhagerling.AH2700A property), 207

status (pymeasure.instruments.danfysik.Danfysik8500 property), 229

status (pymeasure.instruments.eurotest.EurotestHPP12025C property), 233

status (pymeasure.instruments.fwbell.FWBell5080 property), 236

status (pymeasure.instruments.heidenhain.ND287 property), 237

status (pymeasure.instruments.hp.HP6632A property), 285

status (pymeasure.instruments.hp.HP8116A property), 250

status (pymeasure.instruments.hp.hp856Xx.HP856Xx attribute), 253

status (pymeasure.instruments.hp.HPLegacyInstrument property), 283

status (pymeasure.instruments.Instrument property), 110

status (pymeasure.instruments.ipgphotonics.yar.YAR property), 287

status (pymeasure.instruments.keithley.Keithley2000 property), 293

status (pymeasure.instruments.keithley.Keithley2200 property), 338

status (pymeasure.instruments.keithley.Keithley2260B property), 296

status (pymeasure.instruments.keithley.Keithley2306 property), 299

status (pymeasure.instruments.keithley.Keithley2450 property), 313

status (pymeasure.instruments.keithley.Keithley2600 property), 333

status (pymeasure.instruments.keithley.Keithley2700 property), 318

status (pymeasure.instruments.keithley.Keithley2750 property), 332

status (pymeasure.instruments.keithley.Keithley6221 property), 323

status (pymeasure.instruments.keithley.Keithley6517B property), 329

status (pymeasure.instruments.keysight.KeysightE36312A property), 350

status (pymeasure.instruments.lecroy.LeCroyT3DSO1204 property), 370

status (pymeasure.instruments.parker.ParkerGV6 property), 404

status (pymeasure.instruments.signalrecovery.DSP7225 property), 432

status (pymeasure.instruments.signalrecovery.DSP7265 property), 439

status (pymeasure.instruments.srs.SR510 property), 440

status (pymeasure.instruments.tcpowerconversion.CXN property), 454

status (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property), 459

status (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property), 464

status (pymeasure.instruments.teledyne.TeledyneT3AFG property), 467

status (pymeasure.instruments.texio.TexioPSW360L30 property), 486

status (pymeasure.instruments.velleman.VellemanK8090 property), 498

status() (pymeasure.instruments.keithley.Keithley2400 method), 306

status\_byte\_register (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X property), 130

status\_desc (pymeasure.instruments.hp.HP3437A attribute), 244

status\_desc (pymeasure.instruments.hp.HP3478A attribute), 247

status\_desc (pymeasure.instruments.hp.HP6632A attribute), 287

tribute), 285

status\_desc (pymeasure.instruments.hp.HPLegacyInstrument.attribute), 283

status\_hex (pymeasure.instruments.danfysik.Danfysik8500 property), 229

status\_info\_shown (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 416

status\_preset() (pymeasure.instruments.rohdeschwarz.sfm.SFM method), 416

status\_reg (pymeasure.instruments.rohdeschwarz.sfm.SFM property), 416

StatusRegister (class in pymeasure.instruments.hp.hp856Xx), 279

std\_current (pymeasure.instruments.keithley.Keithley2400 property), 306

std\_current (pymeasure.instruments.keithley.Keithley2450 property), 313

std\_resistance (pymeasure.instruments.keithley.Keithley2400 property), 306

std\_resistance (pymeasure.instruments.keithley.Keithley2450 property), 313

std\_voltage (pymeasure.instruments.keithley.Keithley2400 property), 306

std\_voltage (pymeasure.instruments.keithley.Keithley2450 property), 313

step (pymeasure.instruments.agilent.agilent4156.VARX property), 168

step\_current\_down() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423

step\_current\_up() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423

step\_points (pymeasure.instruments.agilent.Agilent8257B property), 154

step\_position (pymeasure.instruments.anaheimautomation.DPSeriesMotorController property), 204

step\_voltage\_down() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423

step\_voltage\_up() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423

stepd (pymeasure.instruments.attocube.anc300.Axis property), 226

stepEnabled() (pymeasure.display.inputs.IntegerInput method), 86

stepEnabled() (pymeasure.display.inputs.ScientificInput method), 86

steps\_to\_absolute() (pymeasure.instruments.anaheimautomation.DPSeriesMotorController method), 204

stepu (pymeasure.instruments.attocube.anc300.Axis property), 226

StimulusResponse (pymeasure.instruments.hp.hp856Xx.SweepCoupleMode attribute), 280

stop (pymeasure.instruments.agilent.agilent4156.VARX property), 169

stop (pymeasure.instruments.agilent.agilentB1500.StaircaseSweepPostOutput attribute), 195

stop() (pymeasure.experiment.listeners.Recorder method), 73

stop() (pymeasure.instruments.advantest.advantestR624X.AdvantestR624X method), 124

stop() (pymeasure.instruments.advantest.advantestR624X.SMUChannel method), 140

stop() (pymeasure.instruments.agilent.agilent4156.Agilent4156 method), 166

stop() (pymeasure.instruments.anaheimautomation.DPSeriesMotorController method), 204

stop() (pymeasure.instruments.attocube.anc300.Axis method), 226

stop() (pymeasure.instruments.keysight.KeysightDSOX1102G method), 340

stop() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 370

stop() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 384

stop() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalGenerator method), 389

stop() (pymeasure.instruments.parker.ParkerGV6 method), 404

stop() (pymeasure.instruments.teledyne.TeledyneOscilloscope method), 470

stop() (pymeasure.instruments.thermotron.Thermotron3800 method), 488

stop\_all() (pymeasure.instruments.attocube.anc300.ANC300Controller method), 224

stop\_buffer() (pymeasure.instruments.keithley.Keithley2000 method), 293

stop\_buffer() (pymeasure.instruments.keithley.Keithley2400 method), 306

stop\_buffer() (pymeasure.instruments.keithley.Keithley2450 method), 313

stop\_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 318

stop\_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 318

<code>sure.instruments.keithley.Keithley6221</code> <code>method</code> ), 323	<code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>method</code> ), 272
<code>stop_buffer()</code> ( <code>pymea-</code> <code>sure.instruments.keithley.Keithley6517B</code> <code>method</code> ), 329	<code>stored_reading</code> ( <code>pymea-</code> <code>sure.instruments.hp.HP34401A</code> <code>property</code> ), 242
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.advantest.advantestR3767CG.AdvantestR3767CG</code> <code>property</code> ), 122	<code>stored_readings_count</code> ( <code>pymea-</code> <code>sure.instruments.hp.HP34401A</code> <code>property</code> ), 242
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.agilent.Agilent8257D</code> <code>prop-</code> <code>erty</code> ), 155	<code>StringInput</code> ( <code>class in pymeasure.display.inputs</code> ), 86
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.agilent.Agilent8722ES</code> <code>property</code> ), 156	<code>strip_chart_dat1</code> ( <code>pymeasure.instruments.srs.SR860</code> <code>property</code> ), 451
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.agilent.AgilentE4408B</code> <code>property</code> ), 157	<code>strip_chart_dat2</code> ( <code>pymeasure.instruments.srs.SR860</code> <code>property</code> ), 451
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.agilent.AgilentE4408B</code> <code>property</code> ), 157	<code>strip_chart_dat3</code> ( <code>pymeasure.instruments.srs.SR860</code> <code>property</code> ), 451
<code>stop_frequency</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>at-</code> <code>tribute</code> ), 259	<code>strip_chart_dat4</code> ( <code>pymeasure.instruments.srs.SR860</code> <code>property</code> ), 451
<code>stop_power</code> ( <code>pymeasure.instruments.agilent.Agilent8257D</code> <code>property</code> ), 155	<code>subsystem_info</code> ( <code>pymea-</code> <code>sure.instruments.rohdeschwarz.sfm.SFM</code> <code>property</code> ), 416
<code>stop_ramp()</code> ( <code>pymeasure.instruments.danfysik.Danfysik8500</code> <code>method</code> ), 229	<code>subtract_display_line_from_trace_b()</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>method</code> ), 265
<code>stop_sequence()</code> ( <code>pymea-</code> <code>sure.instruments.danfysik.Danfysik8500</code> <code>method</code> ), 229	<code>supply_current</code> ( <code>pymeasure.instruments.ami.AMI430</code> <code>property</code> ), 201
<code>stop_sequence()</code> ( <code>pymea-</code> <code>sure.instruments.rohdeschwarz.hmp.HMP4040</code> <code>method</code> ), 423	<code>sweep_auto_abort()</code> ( <code>pymea-</code> <code>sure.instruments.agilent.agilentB1500.AgilentB1500</code> <code>method</code> ), 186
<code>stop_step_sweep()</code> ( <code>pymea-</code> <code>sure.instruments.agilent.Agilent8257D</code> <code>method</code> ), 155	<code>sweep_couple</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>at-</code> <code>tribute</code> ), 270
<code>StoppableQThread</code> ( <code>class in pymeasure.display.thread</code> ), 89	<code>sweep_delay_time</code> ( <code>pymea-</code> <code>sure.instruments.advantest.advantestR624X.AdvantestR624X</code> <code>property</code> ), 125
<code>store_config</code> ( <code>pymea-</code> <code>sure.instruments.advantest.advantestR624X.AdvantestR624X</code> <code>property</code> ), 131	<code>sweep_marker_frequency</code> ( <code>pymea-</code> <code>sure.instruments.hp.HP8116A</code> <code>property</code> ), 250
<code>store_image()</code> ( <code>pymea-</code> <code>sure.instruments.anritsu.AnritsuMS464xB</code> <code>method</code> ), 219	<code>sweep_mode</code> ( <code>pymeasure.instruments.anritsu.anritsuMS464xB.Measurement</code> <code>property</code> ), 222
<code>store_metadata()</code> ( <code>pymea-</code> <code>sure.experiment.results.Results</code> <code>method</code> ), 80	<code>sweep_mode</code> ( <code>pymeasure.instruments.keysight.KeysightN7776C</code> <code>property</code> ), 342
<code>store_open()</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>method</code> ), 272	<code>sweep_output</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>at-</code> <code>tribute</code> ), 270
<code>store_sequence_command()</code> ( <code>pymea-</code> <code>sure.instruments.advantest.advantestR624X.AdvantestR624X</code> <code>method</code> ), 126	<code>sweep_points</code> ( <code>pymea-</code> <code>sure.instruments.keysight.KeysightN7776C</code> <code>property</code> ), 342
<code>store_short()</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>method</code> ), 272	<code>sweep_rate</code> ( <code>pymeasure.instruments.oxfordinstruments.IPS120_10</code> <code>property</code> ), 401
<code>store_thru()</code> ( <code>pymea-</code>	<code>sweep_single</code> ( <code>pymea-</code> <code>sure.instruments.hp.hp856Xx.HP856Xx</code> <code>at-</code> <code>tribute</code> ), 270
	<code>sweep_speed</code> ( <code>pymeasure.instruments.keysight.KeysightN7776C</code> <code>property</code> ), 342

**sweep\_start** (*pymeasure.instruments.hp.HP8116A* property), 250  
**sweep\_state** (*pymeasure.instruments.keysight.KeysightN7776C* property), 343  
**sweep\_status** (*pymeasure.instruments.oxfordinstruments.IPS120\_10* property), 401  
**sweep\_status** (*pymeasure.instruments.oxfordinstruments.ITC503* property), 397  
**sweep\_step** (*pymeasure.instruments.keysight.KeysightN7776C* property), 343  
**sweep\_stop** (*pymeasure.instruments.hp.HP8116A* property), 250  
**sweep\_table** (*pymeasure.instruments.oxfordinstruments.ITC503* property), 397  
**sweep\_time** (*pymeasure.instruments.agilent.Agilent8722ES* property), 156  
**sweep\_time** (*pymeasure.instruments.agilent.AgilentE4408B* property), 157  
**sweep\_time** (*pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel* property), 222  
**sweep\_time** (*pymeasure.instruments.hp.HP8116A* property), 250  
**sweep\_time** (*pymeasure.instruments.hp.hp856Xx.HP856Xx* attribute), 270  
**sweep\_time** (*pymeasure.instruments.rohdeschwarz.fsl.FSL* property), 421  
**sweep\_timing()** (*pymeasure.instruments.agilent.agilentB1500.AgilentB1500* method), 186  
**sweep\_twoway** (*pymeasure.instruments.keysight.KeysightN7776C* property), 343  
**sweep\_type** (*pymeasure.instruments.anritsu.anritsuMS464xB.MeasurementChannel* property), 222  
**sweep\_wl\_start** (*pymeasure.instruments.keysight.KeysightN7776C* property), 343  
**sweep\_wl\_stop** (*pymeasure.instruments.keysight.KeysightN7776C* property), 343  
**SweepCoupleMode** (class in *pymeasure.instruments.hp.hp856Xx*), 280  
**SweepMode** (class in *pymeasure.instruments.advantest.advantestR624X*), 143  
**SweepMode** (class in *pymeasure.instruments.agilent.agilentB1500*), 194  
**SweepOut** (class in *pymeasure.instruments.hp.hp856Xx*), 280  
**SwissArmyFake** (class in *pymeasure.instruments.fakes*), 113  
**switch\_heater\_enabled** (*pymeasure.instruments.oxfordinstruments.IPS120\_10* property), 401  
**switch\_heater\_status** (*pymeasure.instruments.oxfordinstruments.IPS120\_10* property), 401  
**switch\_off** (*pymeasure.instruments.velleman.VellemanK8090* property), 498  
**switch\_on** (*pymeasure.instruments.velleman.VellemanK8090* property), 498  
**SwitchHeaterError** (class in *pymeasure.instruments.oxfordinstruments.ips120\_10*), 402  
**sync\_sequence()** (*pymeasure.instruments.danfysik.Danfysik8500* method), 229  
**synchronous\_sweep\_source()** (*pymeasure.instruments.agilent.agilentB1500.SMU* method), 190  
**SystemCurrent** (*pymeasure.instruments.temptronic.ATS525* property), 482  
**system\_number** (*pymeasure.instruments.rohdeschwarz.sfm.SFM* property), 416  
**system\_setup** (*pymeasure.instruments.keysight.KeysightDSOX1102G* property), 340  
**system\_status\_code** (*pymeasure.instruments.siglentechnologies.siglent\_spdbase.SPDBase* property), 424  
**system\_temp** (*pymeasure.instruments.toptica.ibeamsmart.IBeamSmart* property), 497  
**SystemStatusCode** (class in *pymeasure.instruments.siglentechnologies.siglent\_spdbase*), 424

## T

**Table** (class in *pymeasure.display.widgets.table\_widget*), 97  
**TableWidget** (class in *pymeasure.display.widgets.table\_widget*), 97  
**TabWidget** (class in *pymeasure.display.widgets.tab\_widget*), 94  
**talk\_ascii** (*pymeasure.instruments.hp.HP3437A* property), 244  
**target\_current** (*pymeasure.instruments.ami.AMI430* property), 201  
**target\_field** (*pymeasure.instruments.ami.AMI430* property), 201  
**target\_voltage** (*pymeasure.instruments.oxfordinstruments.ITC503* property), 397  
**target\_voltage\_table** (*pymeasure.instruments.oxfordinstruments.ITC503*

property), 397

TC038 (class in `pymeasure.instruments.hcp`), 238

TC038D (class in `pymeasure.instruments.hcp`), 239

TDK\_Gen40\_38 (class in `pymeasure.instruments.tdk.tdk_gen40_38`), 456

TDK\_Gen80\_65 (class in `pymeasure.instruments.tdk.tdk_gen80_65`), 460

TDS2000 (class in `pymeasure.instruments.tektronix`), 465

TEDSetTemperature (pymeasure.instruments.thorlabs.ThorlabsPro8000 property), 489

TEDStatus (pymeasure.instruments.thorlabs.ThorlabsPro8000 property), 489

TeledyneOscilloscope (class in `pymeasure.instruments.teledyne`), 468

TeledyneOscilloscopeChannel (class in `pymeasure.instruments.teledyne.teledyne_oscilloscope`), 473

TeledyneT3AFG (class in `pymeasure.instruments.teledyne`), 465

TelnetAdapter (class in `pymeasure.adapters`), 63

temp (pymeasure.instruments.toptica.ibeamsmart.IBeamSmart property), 497

temperature (pymeasure.instruments.agilent.Agilent34450A property), 162

temperature (pymeasure.instruments.fluke.Fluke7341 property), 234

temperature (pymeasure.instruments.hcp.TC038 property), 238

temperature (pymeasure.instruments.hcp.TC038D property), 239

temperature (pymeasure.instruments.ipgphotonics.yar.YAR property), 287

temperature (pymeasure.instruments.keithley.Keithley2000 property), 293

temperature (pymeasure.instruments.tcpowerconversion.CVT property), 455

temperature (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature (pymeasure.instruments.thermotron.Thermotron3800 property), 488

temperature\_1 (pymeasure.instruments.oxfordinstruments.ITC503 property), 398

temperature\_2 (pymeasure.instruments.oxfordinstruments.ITC503 property), 398

temperature\_3 (pymeasure.instruments.oxfordinstruments.ITC503 property), 398

temperature\_celsius (pymeasure.instruments.lakeshore.LakeShore211 property), 353

temperature\_condition\_status\_code (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_digits (pymeasure.instruments.keithley.Keithley2000 property), 293

temperature\_error (pymeasure.instruments.oxfordinstruments.ITC503 property), 398

temperature\_event\_status (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_fahrenheit (pymeasure.instruments.lakeshore.LakeShore211 property), 353

temperature\_kelvin (pymeasure.instruments.lakeshore.LakeShore211 property), 353

temperature\_limit\_air\_dut (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_limit\_air\_high (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_limit\_air\_low (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_nplc (pymeasure.instruments.keithley.Keithley2000 property), 293

temperature\_reference (pymeasure.instruments.keithley.Keithley2000 property), 293

temperature\_seed (pymeasure.instruments.ipgphotonics.yar.YAR property), 287

temperature\_sensor (pymeasure.instruments.lakeshore.LakeShore211 property), 353

temperature\_setpoint (pymeasure.instruments.oxfordinstruments.ITC503 property), 398

temperature\_setpoint (pymeasure.instruments.temptronic.ATSBBase property), 481

temperature\_setpoint\_window (pymeasure.instruments.temptronic.ATSBBase property), 482

temperature\_soak\_time (pymeasure.instruments.temptronic.ATSBBase property), 482

TemperatureStatusCode (class in `pymeasure.instruments.temptronic.temptronic_base`), 482

terminals\_used (pymeasure.instruments.temptronic.ATSBBase property), 481

- `sure.instruments.hp.HP34401A` (property), 242
- `test_method()` (`pymeasure.generator.Generator` method), 70
- `test_property_getter()` (`pymeasure.generator.Generator` method), 70
- `test_property_setter()` (`pymeasure.generator.Generator` method), 70
- `test_property_setter_batch()` (`pymeasure.generator.Generator` method), 70
- `TexioPSW360L30` (class in `pymeasure.instruments.texio`), 484
- `text_enabled` (`pymeasure.instruments.keithley.Keithley2700` property), 318
- `textFromValue()` (`pymeasure.display.inputs.ScientificInput` method), 86
- `Thermotron3800` (class in `pymeasure.instruments.thermotron`), 487
- `Thermotron3800.Thermotron3800Mode` (class in `pymeasure.instruments.thermotron`), 487
- `theta` (`pymeasure.instruments.ametek.Ametek7270` property), 199
- `theta` (`pymeasure.instruments.srs.SR830` property), 445
- `theta` (`pymeasure.instruments.srs.SR860` property), 451
- `ThorlabsPM100USB` (class in `pymeasure.instruments.thorlabs`), 489
- `ThorlabsPro8000` (class in `pymeasure.instruments.thorlabs`), 489
- `threshold` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 253
- `TIME` (`pymeasure.instruments.agilent.agilentB1500.ADCMode` attribute), 193
- `time` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113
- `time` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` property), 416
- `time_constant` (`pymeasure.instruments.ametek.Ametek7270` property), 199
- `time_constant` (`pymeasure.instruments.signalrecovery.DSP7225` property), 432
- `time_constant` (`pymeasure.instruments.signalrecovery.DSP7265` property), 439
- `time_constant` (`pymeasure.instruments.srs.SR510` property), 441
- `time_constant` (`pymeasure.instruments.srs.SR830` property), 446
- `time_constant` (`pymeasure.instruments.srs.SR860` property), 451
- `time_stamp` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` (class in `pymeasure.instruments.agilent.agilentB1500` property), 186
- `timebase` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 340
- `timebase` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 370
- `timebase` (`pymeasure.instruments.srs.SR860` property), 451
- `timebase` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 470
- `timebase_hor_magnify` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 370
- `timebase_hor_position` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 370
- `timebase_mode` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 340
- `timebase_offset` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 340
- `timebase_offset` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 370
- `timebase_offset` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 470
- `timebase_range` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 340
- `timebase_scale` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 340
- `timebase_scale` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 370
- `timebase_scale` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 470
- `timebase_setup()` (`pymeasure.instruments.keysight.KeysightDSOX1102G` method), 340
- `timebase_setup()` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` method), 370
- `timebase_setup()` (`pymeasure.instruments.teledyne.TeledyneOscilloscope` method), 470
- `to_dict()` (`pymeasure.instruments.agilent.agilentB1500.QueryLearn` static method), 191
- `total_cycle_count` (`pymeasure.instruments.temptronic.ATSB` property), 482

`sure.instruments.anritsu.AnritsuMS464xB`),  
 223  
**Trace** (class in `pymeasure.instruments.hp.hp856Xx`), 277  
**trace()** (`pymeasure.instruments.agilent.AgilentE4408B`  
 method), 157  
**trace\_1** (`pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG`  
 property), 122  
**trace\_a\_minus\_b\_enabled** (`pymeasure.instruments.hp.hp856Xx.HP856Xx` at-  
 tribute), 266  
**trace\_a\_minus\_b\_plus\_dl\_enabled** (`pymeasure.instruments.hp.hp856Xx.HP856Xx` at-  
 tribute), 265  
**trace\_data\_format** (`pymeasure.instruments.hp.hp856Xx.HP856Xx` at-  
 tribute), 264  
**trace\_df()** (`pymeasure.instruments.agilent.AgilentE4408B` method), 157  
**trace\_marker** (`pymeasure.instruments.anritsu.AnritsuMS9710C`  
 property), 210  
**trace\_marker\_center** (`pymeasure.instruments.anritsu.AnritsuMS9710C`  
 property), 210  
**trace\_mode** (`pymeasure.instruments.rohdeschwarz.fsl.FSL` property), 421  
**tracking** (`pymeasure.instruments.ni.virtualbench.VirtualBench` property), 390  
**tracking\_adjust\_coarse** (`pymeasure.instruments.hp.HP8560A` property),  
 274  
**tracking\_adjust\_fine** (`pymeasure.instruments.hp.HP8560A` property),  
 274  
**train\_magnet()** (`pymeasure.instruments.oxfordinstruments.IPS120_10`  
 method), 401  
**transfer\_sequence()** (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040`  
 method), 423  
**translate\_to\_global()** (`pymeasure.display.widgets.table_widget.PandasModelBase`  
 method), 96  
**translate\_to\_global()** (`pymeasure.display.widgets.table_widget.PandasModelByColumn`  
 method), 96  
**translate\_to\_global()** (`pymeasure.display.widgets.table_widget.PandasModelByRow`  
 method), 97  
**translate\_to\_local()** (`pymeasure.display.widgets.table_widget.PandasModelBase`  
 method), 96  
**translate\_to\_local()** (`pymeasure.display.widgets.table_widget.PandasModelByColumn`  
 method), 96  
**translate\_to\_local()** (`pymeasure.display.widgets.table_widget.PandasModelByRow`  
 method), 97  
**triad()** (`pymeasure.instruments.keithley.Keithley2400` method), 306  
**triad()** (`pymeasure.instruments.keithley.Keithley2450` method), 313  
**triad()** (`pymeasure.instruments.keithley.Keithley2700` method), 318  
**triad()** (`pymeasure.instruments.keithley.Keithley6221` method), 323  
**trigger** (`pymeasure.instruments.hp.HP3437A` property), 244  
**trigger** (`pymeasure.instruments.hp.HP3478A` property), 247  
**TRIGGER** (`pymeasure.instruments.hp.hp856Xx.StatusRegister` attribute), 280  
**trigger** (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 371  
**trigger** (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 470  
**trigger()** (`pymeasure.instruments.activetechnologies.AWG401x_AWG`  
 method), 119  
**trigger()** (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X`  
 method), 124  
**trigger()** (`pymeasure.instruments.advantest.advantestR624X.SMUChannel`  
 method), 134  
**trigger()** (`pymeasure.instruments.agilent.Agilent33220A` method), 171  
**trigger()** (`pymeasure.instruments.agilent.Agilent33500` method), 175  
**trigger()** (`pymeasure.instruments.andeenhagerling.AH2500A` method), 205  
**trigger()** (`pymeasure.instruments.andeenhagerling.AH2700A` method), 207  
**trigger()** (`pymeasure.instruments.anritsu.AnritsuMS464xB` method), 219  
**trigger()** (`pymeasure.instruments.keithley.Keithley2400` method), 306  
**trigger()** (`pymeasure.instruments.keithley.Keithley2450` method), 314  
**trigger()** (`pymeasure.instruments.keithley.Keithley6221` method), 323  
**trigger()** (`pymeasure.instruments.keithley.Keithley6517B` method), 329  
**trigger\_auto\_delay\_enabled** (`pymeasure.instruments.hp.HP34401A` property),  
 242  
**trigger\_continuous()** (`pymeasure.instruments.anritsu.AnritsuMS464xB`  
 method), 219  
**trigger\_count** (`pymeasure.instruments.hp.HP34401A` property), 243

[trigger\\_count](#) (`pymeasure.instruments.keithley.Keithley2000` property), 293  
[trigger\\_count](#) (`pymeasure.instruments.keithley.Keithley2400` property), 306  
[trigger\\_coupling](#) (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope` property), 475  
[trigger\\_delay](#) (`pymeasure.instruments.hp.HP34401A` property), 243  
[trigger\\_delay](#) (`pymeasure.instruments.keithley.Keithley2000` property), 293  
[trigger\\_delay](#) (`pymeasure.instruments.keithley.Keithley2400` property), 306  
[trigger\\_immediately\(\)](#) (`pymeasure.instruments.keithley.Keithley2400` method), 306  
[trigger\\_immediately\(\)](#) (`pymeasure.instruments.keithley.Keithley6221` method), 323  
[trigger\\_immediately\(\)](#) (`pymeasure.instruments.keithley.Keithley6517B` method), 329  
[trigger\\_in](#) (`pymeasure.instruments.keysight.KeysightN7776C` property), 343  
[trigger\\_input](#) (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 133  
[trigger\\_level](#) (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope` property), 475  
[trigger\\_level2](#) (`pymeasure.instruments.lecroy.lecroyT3DSO1204.LeCroyT3DSO1204` property), 374  
[trigger\\_link\\_function\\_enabled](#) (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` property), 131  
[trigger\\_mode](#) (`pymeasure.instruments.hp.hp856Xx.HP856Xx` attribute), 251  
[trigger\\_mode](#) (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 371  
[trigger\\_mode](#) (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 470  
[trigger\\_on\\_bus\(\)](#) (`pymeasure.instruments.keithley.Keithley2400` method), 306  
[trigger\\_on\\_bus\(\)](#) (`pymeasure.instruments.keithley.Keithley6221` method), 323  
[trigger\\_on\\_bus\(\)](#) (`pymeasure.instruments.keithley.Keithley6517B` method), 329  
[trigger\\_on\\_external\(\)](#) (`pymeasure.instruments.keithley.Keithley2400` method), 306  
[trigger\\_on\\_external\(\)](#) (`pymeasure.instruments.keithley.Keithley6221` method), 323  
[trigger\\_out](#) (`pymeasure.instruments.keysight.KeysightN7776C` property), 343  
[trigger\\_output\\_signal\(\)](#) (`pymeasure.instruments.advantest.advantestR624X.AdvantestR624X` method), 126  
[trigger\\_output\\_timing](#) (`pymeasure.instruments.advantest.advantestR624X.SMUChannel` property), 132  
[trigger\\_ramp\\_to\\_level\(\)](#) (`pymeasure.instruments.yokogawa.YokogawaGS200` method), 501  
[trigger\\_select](#) (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` property), 371  
[trigger\\_select](#) (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 471  
[trigger\\_setup\(\)](#) (`pymeasure.instruments.lecroy.LeCroyT3DSO1204` method), 372  
[trigger\\_setup\(\)](#) (`pymeasure.instruments.teledyne.TeledyneOscilloscope` property), 475  
[trigger\\_single\(\)](#) (`pymeasure.instruments.anritsu.AnritsuMS464xB` method), 219  
[trigger\\_single\\_autozero\(\)](#) (`pymeasure.instruments.hp.HP34401A` method), 243  
[trigger\\_slope](#) (`pymeasure.instruments.hp.HP8116A` property), 250  
[trigger\\_slope](#) (`pymeasure.instruments.teledyne.teledyne_oscilloscope.TeledyneOscilloscope` property), 475  
[trigger\\_source](#) (`pymeasure.instruments.activetechnologies.AWG401x_AWG` property), 119  
[trigger\\_source](#) (`pymeasure.instruments.agilent.Agilent33220A` property), 171  
[trigger\\_source](#) (`pymeasure.instruments.agilent.Agilent33500` property), 175  
[trigger\\_source](#) (`pymeasure.instruments.agilent.Agilent33500` property), 175

- sure.instruments.agilent.AgilentE4980* (property), 158
- trigger\_source* (*pymea-  
sure.instruments.anritsu.AnritsuMS464xB*  
property), 219
- trigger\_source* (*pymea-  
sure.instruments.hp.HP34401A* property), 243
- trigger\_state* (*pymea-  
sure.instruments.agilent.Agilent33220A*  
property), 171
- trigger\_sweep()* (*pymea-  
sure.instruments.hp.hp856Xx.HP856Xx*  
method), 270
- triggered\_caplossvolt()* (*pymea-  
sure.instruments.andenhagerling.AH2500A*  
method), 205
- triggered\_caplossvolt()* (*pymea-  
sure.instruments.andenhagerling.AH2700A*  
method), 207
- TriggerInputType* (class in *pymea-  
sure.instruments.advantest.advantestR624X*), 143
- TriggerMode* (class in *pymea-  
sure.instruments.hp.hp856Xx*), 280
- TriggerOutputSignalTiming* (class in *pymea-  
sure.instruments.advantest.advantestR624X*), 146
- tristate\_lines()* (*pymea-  
sure.instruments.ni.virtualbench.VirtualBench.DigitalData*  
method), 380
- tune\_capacity* (*pymea-  
sure.instruments.tcpowerconversion.CXN*  
property), 455
- tune\_capacity* (*pymea-  
sure.instruments.tcpowerconversion.tccxn.PresetChannel*  
property), 455
- tuner* (*pymea-  
sure.instruments.tcpowerconversion.CXN*  
property), 455
- TV\_country* (*pymea-  
sure.instruments.rohdeschwarz.sfm.SFM*  
property), 407
- TV\_standard* (*pymea-  
sure.instruments.rohdeschwarz.sfm.SFM*  
property), 408
- U**
- unblank\_front()* (*pymea-  
sure.instruments.srs.SR570*  
method), 442
- under\_voltage* (*pymea-  
sure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38*  
property), 459
- under\_voltage* (*pymea-  
sure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65*  
property), 464
- Uniform* (*pymea-  
sure.instruments.hp.hp856Xx.WindowType*  
attribute), 281
- unique\_filename()* (in module *pymea-  
sure.experiment.results*), 81
- unit* (*pymea-  
sure.instruments.fluke.Fluke7341* property), 234
- unit* (*pymea-  
sure.instruments.lakeshore.LakeShore421*  
property), 358
- unit* (*pymea-  
sure.instruments.lakeshore.LakeShore425*  
property), 359
- unit* (*pymea-  
sure.instruments.lecroy.lecroyT3DSO1204.LeCroyT3DSO1204*  
property), 375
- unit* (*pymea-  
sure.instruments.mksinst.mks937b.MKS937B*  
property), 376
- units* (*pymea-  
sure.instruments.fwbell.FWBell5080* prop-  
erty), 236
- units* (*pymea-  
sure.instruments.heidenhain.ND287* prop-  
erty), 237
- units* (*pymea-  
sure.instruments.newport.esp300.Axis*  
property), 378
- UnknownProcedure* (class in *pymea-  
sure.experiment.procedure*), 74
- unlock\_harmonic\_number()* (*pymea-  
sure.instruments.hp.HP8561B* method), 276
- update()* (*pymea-  
sure.display.curves.Crosshairs*  
method), 84
- update\_channels()* (*pymea-  
sure.instruments.anritsu.AnritsuMS464xB*  
method), 220
- update\_data()* (*pymea-  
sure.display.curves.ResultsCurve* method), 84
- update\_estimates()* (*pymea-  
sure.display.widgets.estimator\_widget.EstimatorWidget*  
method), 90
- update\_frequency\_range()* (*pymea-  
sure.instruments.anritsu.anritsuMS464xB.MeasurementChannel*  
method), 222
- update\_line()* (*pymea-  
sure.experiment.experiment.Experiment*  
method), 72
- update\_parameter()* (*pymea-  
sure.display.inputs.Input*  
method), 85
- update\_plot()* (*pymea-  
sure.experiment.experiment.Experiment*  
method), 72
- update\_status()* (*pymea-  
sure.experiment.workers.Worker* method), 79
- update\_traces()* (*pymea-  
sure.instruments.anritsu.anritsuMS464xB.MeasurementChannel*  
method), 222
- updateEditorGeometry()* (*pymea-  
sure.display.widgets.sequencer\_widget.ComboBoxDelegate*

- method), 92
- updateEditorGeometry() (pymea-  
sure.display.widgets.sequencer\_widget.LineEditDelegate  
method), 93
- use\_absolute\_position() (pymea-  
sure.instruments.parker.ParkerGV6  
method), 404
- use\_external\_source (pymea-  
sure.instruments.rohdeschwarz.sfm.Sound\_Channel  
property), 418
- use\_front\_terminals() (pymea-  
sure.instruments.keithley.Keithley2400  
method), 307
- use\_front\_terminals() (pymea-  
sure.instruments.keithley.Keithley2450  
method), 314
- use\_rear\_terminals() (pymea-  
sure.instruments.keithley.Keithley2400  
method), 307
- use\_rear\_terminals() (pymea-  
sure.instruments.keithley.Keithley2450  
method), 314
- use\_relative\_position() (pymea-  
sure.instruments.parker.ParkerGV6  
method), 404
- V**
- V (pymea.sure.instruments.hp.hp856Xx.AmplitudeUnits  
attribute), 277
- validate() (pymea.sure.display.inputs.ScientificInput  
method), 86
- validate() (pymea.sure.display.widgets.sequencer\_widget.ExpressionWidget  
method), 92
- validate\_auto\_range\_terminal() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter  
method), 382
- validate\_channel() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope  
method), 389
- validate\_channel() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.PowerSupply  
method), 390
- validate\_dmm\_function() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter  
method), 382
- validate\_lines() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter  
method), 380
- validate\_range() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter  
static method), 382
- validate\_trigger\_instance() (pymea-  
sure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope  
static method), 389
- valueFromText() (pymea-  
sure.display.inputs.ScientificInput  
method), 86
- values() (pymea.sure.adapters.Adapter method), 48
- values() (pymea.sure.adapters.FakeAdapter method), 68
- values() (pymea.sure.adapters.PrologixAdapter  
method), 59
- values() (pymea.sure.adapters.SerialAdapter method), 55
- values() (pymea.sure.adapters.TelnetAdapter method), 65
- values() (pymea.sure.adapters.VISAAdapter method), 52
- values() (pymea.sure.adapters.VXI11Adapter method), 62
- values() (pymea.sure.instruments.common\_base.CommonBase  
method), 108
- values() (pymea.sure.instruments.hp.HPLegacyInstrument  
method), 283
- values() (pymea.sure.instruments.keysight.KeysightE36312A  
method), 350
- values() (pymea.sure.instruments.lecroy.LeCroyT3DSO1204  
method), 373
- values() (pymea.sure.instruments.rohdeschwarz.sfm.Sound\_Channel  
method), 418
- values() (pymea.sure.instruments.tcpowerconversion.CXN  
method), 455
- values() (pymea.sure.instruments.tcpowerconversion.tccxn.PresetChannel  
method), 455
- valve\_scaling (pymea-  
sure.instruments.oxfordinstruments.ITC503  
property), 398
- VAR1 (class in pymea-  
sure.instruments.agilent.agilent4156), 167
- VAR2 (class in pymea-  
sure.instruments.agilent.agilent4156), 168
- VARD (class in pymea-  
sure.instruments.agilent.agilent4156), 168
- VARX (class in pymea-  
sure.instruments.agilent.agilent4156), 168
- VectorParameter (class in pymea-  
sure.experiment.parameters), 78
- VellemanK8090 (class in pymea-  
sure.instruments.velleman), 497
- VellemanK8090Switches (class in pymea-  
sure.instruments.velleman), 499
- verify\_calibration\_data() (pymea-  
sure.instruments.hp.HP3478A method), 247
- verify\_calibration\_entry() (pymea-  
sure.instruments.hp.HP3478A method), 248
- version (pymea.sure.adapters.PrologixAdapter prop-  
erty), 59
- version (pymea.sure.instruments.attocube.anc300.ANC300Controller  
property), 224

`version` (`pymeasure.instruments.oxfordinstruments.IPS120_10` `sure.instruments.ni.virtualbench`), 392  
    `property`), 401  
    `VISAAdapter` (`class in pymeasure.adapters`), 49  
`version` (`pymeasure.instruments.oxfordinstruments.ITC503` `vision_average_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 398  
`version` (`pymeasure.instruments.rohdeschwarz.hmp.HMP4040` `property`), 416  
`version` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` `vision_balance` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 416  
`version` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` `vision_carrier_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 416  
`version` (`pymeasure.instruments.toptica.ibeamsmart.IBeamSmart` `vision_carrier_frequency` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 416  
`version` (`pymeasure.instruments.velleman.VellemanK8090` `vision_clamping_average` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
`vhighest` (`pymeasure.instruments.andeenhagerling.AH2500A` `vision_clamping_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
`vhighest` (`pymeasure.instruments.andeenhagerling.AH2700A` `vision_clamping_mode` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
`Video` (`pymeasure.instruments.hp.hp856Xx.TriggerMode` `vision_pre_correction_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
    `attribute`), 280  
`video_average` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` `vision_residual_carrier_level` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
    `attribute`), 261  
`video_bandwidth` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` `vision_sideband_filter_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
    `attribute`), 261  
`video_bandwidth` (`pymeasure.instruments.rohdeschwarz.fsl.FSL` `vision_videosignal_enabled` (`pymeasure.instruments.rohdeschwarz.sfm.SFM` `property`), 417  
    `property`), 421  
`video_bandwidth_to_resolution_bandwidth` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` `visit_tree()` (`pymeasure.display.widgets.sequencer_widget.SequencerTreeModel` `method`), 94  
    `attribute`), 261  
`video_trigger_level` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` `attribute`), 261  
`view_sense_modes` (`pymeasure.instruments.anritsu.AnritsuMS2090A` `property`), 214  
`view_trace()` (`pymeasure.instruments.hp.hp856Xx.HP856Xx` `method`), 263  
`VirtualBench` (`class in pymeasure.instruments.ni.virtualbench`), 379  
`VirtualBench.DigitalInputOutput` (`class in pymeasure.instruments.ni.virtualbench`), 379  
`VirtualBench.DigitalMultimeter` (`class in pymeasure.instruments.ni.virtualbench`), 381  
`VirtualBench.FunctionGenerator` (`class in pymeasure.instruments.ni.virtualbench`), 383  
`VirtualBench.MixedSignalOscilloscope` (`class in pymeasure.instruments.ni.virtualbench`), 384  
`VirtualBench.PowerSupply` (`class in pymeasure.instruments.ni.virtualbench`), 389  
`VirtualBench_Direct` (`class in pymeasure.instruments.ni.virtualbench`), 392  
`VMU` (`class in pymeasure.instruments.agilent.agilent4156`), 169  
`voltage` (`pymeasure.instruments.agilent.Agilent34450A` `property`), 162  
`VOLTAGE` (`pymeasure.instruments.agilent.agilentB1500.MeasOpMode` `attribute`), 194  
`voltage` (`pymeasure.instruments.aja.DCXS` `property`), 198  
`voltage` (`pymeasure.instruments.ametek.Ametek7270` `property`), 200  
`voltage` (`pymeasure.instruments.attocube.anc300.Axis` `property`), 226  
`voltage` (`pymeasure.instruments.bkprecision.BKPrecision9130B` `property`), 226  
`voltage` (`pymeasure.instruments.deltaelektronika.SM7045D` `property`), 231

[voltage \(pymeasure.instruments.eurotest.EurotestHPP120256 property\), 233](#)  
[voltage \(pymeasure.instruments.fakes.SwissArmyFake property\), 113](#)  
[voltage \(pymeasure.instruments.hp.HP6632A property\), 286](#)  
[voltage \(pymeasure.instruments.keithley.Keithley2000 property\), 293](#)  
[voltage \(pymeasure.instruments.keithley.keithley2200.PSChannel property\), 338](#)  
[voltage \(pymeasure.instruments.keithley.Keithley2260B property\), 297](#)  
[voltage \(pymeasure.instruments.keithley.Keithley2400 property\), 307](#)  
[voltage \(pymeasure.instruments.keithley.Keithley2450 property\), 314](#)  
[voltage \(pymeasure.instruments.keithley.Keithley6517B property\), 329](#)  
[voltage \(pymeasure.instruments.keysight.keysightE36312A.VoltageChannel property\), 351](#)  
[voltage \(pymeasure.instruments.keysight.KeysightN5767A property\), 342](#)  
[voltage \(pymeasure.instruments.rohdeschwarz.hmp.HMP4040 property\), 423](#)  
[voltage \(pymeasure.instruments.siglenttechnologies.siglent\\_spdbase.SPDChannel property\), 426](#)  
[voltage \(pymeasure.instruments.signalrecovery.DSP7225 property\), 432](#)  
[voltage \(pymeasure.instruments.signalrecovery.DSP7265 property\), 439](#)  
[voltage \(pymeasure.instruments.tdk.tdk\\_gen40\\_38.TDK\\_Gen40\\_38 property\), 459](#)  
[voltage \(pymeasure.instruments.tdk.tdk\\_gen80\\_65.TDK\\_Gen80\\_65 property\), 464](#)  
[voltage \(pymeasure.instruments.texio.TexioPSW360L30 property\), 486](#)  
[voltage\\_1 \(pymeasure.instruments.razorbill.razorbillRP1000 property\), 406](#)  
[voltage\\_2 \(pymeasure.instruments.razorbill.razorbillRP100 property\), 406](#)  
[voltage\\_ac \(pymeasure.instruments.agilent.Agilent34410A property\), 159](#)  
[voltage\\_ac \(pymeasure.instruments.agilent.Agilent34450A property\), 162](#)  
[voltage\\_ac \(pymeasure.instruments.hp.HP34401A property\), 243](#)  
[voltage\\_ac\\_auto\\_range \(pymeasure.instruments.agilent.Agilent34450A property\), 162](#)  
[voltage\\_ac\\_bandwidth \(pymeasure.instruments.keithley.Keithley2000 property\), 293](#)  
[voltage\\_ac\\_digits \(pymeasure.instruments.keithley.Keithley2000 property\), 293](#)  
[voltage\\_ac\\_nplc \(pymeasure.instruments.keithley.Keithley2000 property\), 294](#)  
[voltage\\_ac\\_range \(pymeasure.instruments.agilent.Agilent34450A property\), 162](#)  
[voltage\\_ac\\_range \(pymeasure.instruments.keithley.Keithley2000 property\), 294](#)  
[voltage\\_ac\\_reference \(pymeasure.instruments.keithley.Keithley2000 property\), 294](#)  
[voltage\\_ac\\_resolution \(pymeasure.instruments.agilent.Agilent34450A property\), 162](#)  
[voltage\\_amplitude \(pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG property\), 120](#)  
[voltage\\_amplitude\\_max \(pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG property\), 121](#)  
[voltage\\_amplitude\\_min \(pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG property\), 121](#)  
[voltage\\_and\\_current \(pymeasure.instruments.rohdeschwarz.hmp.HMP4040 property\), 423](#)  
[voltage\\_auto\\_range \(pymeasure.instruments.agilent.Agilent34450A property\), 163](#)  
[voltage\\_dc \(pymeasure.instruments.agilent.Agilent34410A property\), 159](#)  
[voltage\\_digits \(pymeasure.instruments.keithley.Keithley2000 property\), 294](#)  
[voltage\\_filter\\_count \(pymeasure.instruments.keithley.Keithley2450 property\), 314](#)  
[voltage\\_filter\\_type \(pymeasure.instruments.keithley.Keithley2450 property\), 314](#)  
[voltage\\_fixed\\_level\\_sweep\(\) \(pymeasure.instruments.advantest.advantestR624X.SMUChannel method\), 135](#)  
[voltage\\_fixed\\_pulsed\\_sweep\(\) \(pymeasure.instruments.advantest.advantestR624X.SMUChannel method\), 136](#)  
[voltage\\_high \(pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG property\), 121](#)  
[voltage\\_high \(pymeasure.instruments.agilent.Agilent33220A property\), 172](#)

`voltage_high` (`pymea-` `sure.instruments.agilent.Agilent33500` `prop-` `erty`), 314  
`voltage_high` (`pymea-` `sure.instruments.agilent.agilent33500` `property`), 175  
`voltage_high_max` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_high_min` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_limit` (`pymea-` `sure.instruments.ami.AMI430` `property`), 202  
`voltage_limit` (`pymea-` `sure.instruments.keithley.keithley2200.PSChannel` `property`), 338  
`voltage_limit` (`pymea-` `sure.instruments.yokogawa.YokogawaGS200` `property`), 501  
`voltage_limit_enabled` (`pymea-` `sure.instruments.keithley.keithley2200.PSChannel` `property`), 338  
`voltage_low` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_low` (`pymea-` `sure.instruments.agilent.Agilent33220A` `property`), 172  
`voltage_low` (`pymea-` `sure.instruments.agilent.Agilent33500` `property`), 175  
`voltage_low` (`pymea-` `sure.instruments.agilent.agilent33500` `property`), 178  
`voltage_low_max` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_low_min` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_name` (`pymea-` `sure.instruments.agilent.agilent4156.SMU` `property`), 167  
`voltage_name` (`pymea-` `sure.instruments.agilent.agilent4156.VMU` `property`), 169  
`voltage_name` (`pymea-` `sure.instruments.agilent.agilent4156.VSU` `property`), 169  
`voltage_nplc` (`pymea-` `sure.instruments.keithley.Keithley2000` `prop-` `erty`), 294  
`voltage_nplc` (`pymea-` `sure.instruments.keithley.Keithley2400` `prop-` `erty`), 307  
`voltage_nplc` (`pymea-` `sure.instruments.keithley.Keithley2450` `prop-` `erty`), 314  
`voltage_nplc` (`pymea-` `sure.instruments.keithley.Keithley6517B` `property`), 329  
`voltage_offset` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_offset_max` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_offset_min` (`pymea-` `sure.instruments.activetechnologies.AWG401x.ChannelAFG` `property`), 121  
`voltage_output_off_state` (`pymea-` `sure.instruments.keithley.Keithley2450` `prop-` `erty`), 314  
`voltage_pulsed_source()` (`pymea-` `sure.instruments.advantest.advantestR624X.SMUChannel` `method`), 135  
`voltage_pulsed_sweep()` (`pymea-` `sure.instruments.advantest.advantestR624X.SMUChannel` `method`), 136  
`voltage_ramp` (`pymea-` `sure.instruments.agilent.Agilent34450A` `property`), 163  
`voltage_range` (`pymea-` `sure.instruments.agilent.Agilent34450A` `property`), 163  
`voltage_range` (`pymea-` `sure.instruments.eurotest.EurotestHPP120256` `property`), 233  
`voltage_range` (`pymea-` `sure.instruments.keithley.Keithley2000` `prop-` `erty`), 294  
`voltage_range` (`pymea-` `sure.instruments.keithley.Keithley2400` `prop-` `erty`), 307  
`voltage_range` (`pymea-` `sure.instruments.keithley.Keithley2450` `prop-` `erty`), 314  
`voltage_range` (`pymea-` `sure.instruments.keithley.Keithley6517B` `property`), 329  
`voltage_range` (`pymea-` `sure.instruments.keysight.KeysightN5767A` `property`), 342  
`voltage_reference` (`pymea-` `sure.instruments.keithley.Keithley2000` `prop-` `erty`), 294

- erty), 294
- voltage\_resolution (pymeasure.instruments.agilent.Agilent34450A property), 163
- voltage\_set\_random\_memory() (pymeasure.instruments.advantest.advantestR624X.SMUChannel method), 137
- voltage\_setpoint (pymeasure.instruments.eurotest.EurotestHPP120256 property), 233
- voltage\_setpoint (pymeasure.instruments.keithley.keithley2200.PSChannel property), 338
- voltage\_setpoint (pymeasure.instruments.keithley.Keithley2260B property), 297
- voltage\_setpoint (pymeasure.instruments.keysight.keysightE36312A.VoltageChannel property), 351
- voltage\_setpoint (pymeasure.instruments.siglentechnologies.siglent\_spdbase.SPDChannel property), 426
- voltage\_setpoint (pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38 property), 460
- voltage\_setpoint (pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65 property), 464
- voltage\_setpoint (pymeasure.instruments.texio.TexioPSW360L30 property), 486
- voltage\_source() (pymeasure.instruments.advantest.advantestR624X.SMUChannel method), 135
- voltage\_step (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 property), 423
- voltage\_sweep() (pymeasure.instruments.advantest.advantestR624X.SMUChannel method), 136
- voltage\_to\_max() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423
- voltage\_to\_min() (pymeasure.instruments.rohdeschwarz.hmp.HMP4040 method), 423
- voltage\_unit (pymeasure.instruments.activetechnologies.AWG401x.ChannelAFG property), 121
- VoltageChannel (class in pymeasure.instruments.keysight.keysightE36312A), 350
- VoltageRange (class in pymeasure.instruments.advantest.advantestR624X), 143
- VSH (class in pymeasure.instruments.thyracont.smartline\_v2), 494
- VSR (class in pymeasure.instruments.thyracont.smartline\_v2), 494
- VSU (class in pymeasure.instruments.agilent.agilent4156), 169
- VXI11Adapter (class in pymeasure.adapters), 61
- W (pymeasure.instruments.hp.hp856Xx.AmplitudeUnits attribute), 277
- wait() (pymeasure.instruments.anritsu.AnritsuMS9710C method), 210
- wait\_for() (pymeasure.instruments.andeenhagerling.AH2700A method), 207
- wait\_for() (pymeasure.instruments.attocube.anc300.ANC300Controller method), 224
- wait\_for() (pymeasure.instruments.Channel method), 112
- wait\_for() (pymeasure.instruments.common\_base.CommonBase method), 109
- wait\_for() (pymeasure.instruments.fwbell.FWBell5080 method), 236
- wait\_for() (pymeasure.instruments.Instrument method), 111
- wait\_for() (pymeasure.instruments.keithley.Keithley2000 method), 294
- wait\_for() (pymeasure.instruments.keithley.Keithley2200 method), 338
- wait\_for() (pymeasure.instruments.keithley.Keithley2260B method), 297
- wait\_for() (pymeasure.instruments.keithley.Keithley2306 method), 299
- wait\_for() (pymeasure.instruments.keithley.Keithley2400 method), 307
- wait\_for() (pymeasure.instruments.keithley.Keithley2450 method), 314
- wait\_for() (pymeasure.instruments.keithley.Keithley2600 method), 333
- wait\_for() (pymeasure.instruments.keithley.Keithley2700 method), 318
- wait\_for() (pymeasure.instruments.keithley.Keithley2750 method), 332
- wait\_for() (pymeasure.instruments.keithley.Keithley6221 method), 324
- wait\_for() (pymeasure.instruments.keithley.Keithley6517B method), 329
- wait\_for() (pymeasure.instruments.keysight.KeysightE36312A method), 350
- wait\_for() (pymeasure.instruments.lecroy.LeCroyT3DSO1204 method), 373

`wait_for()` (`pymeasure.instruments.signalrecovery.DSP7225` method), 233  
method), 432 `wait_for_ready()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 229  
`wait_for()` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38` method), 460  
method), 460 `wait_for_settling()` (`pymeasure.instruments.temptronic.ATSBBase` method), 462  
`wait_for()` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65` method), 464  
method), 464 `wait_for_srq()` (`pymeasure.adapters.PrologixAdapter` method), 60  
`wait_for()` (`pymeasure.instruments.teledyne.TeledyneT3AFG` method), 467  
method), 467 `wait_for_srq()` (`pymeasure.adapters.VISAAdapter` method), 52  
`wait_for()` (`pymeasure.instruments.texio.TexioPSW360L30` method), 486  
method), 486 `wait_for_stop()` (`pymeasure.instruments.newport.esp300.Axis` method), 378  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley2000` method), 294  
method), 294 `wait_for_sweep()` (`pymeasure.instruments.anritsu.AnritsuMS9710C` method), 210  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley2400` method), 307  
method), 307 `wait_for_temperature()` (`pymeasure.instruments.lakeshore.lakeshore_base.LakeShoreTemperature` method), 359  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley2450` method), 314  
method), 314 `wait_for_temperature()` (`pymeasure.instruments.oxfordinstruments.ITC503` method), 398  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley2700` method), 318  
method), 318 `wait_for_trigger()` (`pymeasure.instruments.agilent.Agilent33220A` method), 172  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley6221` method), 324  
method), 324 `wait_for_trigger()` (`pymeasure.instruments.agilent.Agilent33500` method), 175  
`wait_for_buffer()` (`pymeasure.instruments.keithley.Keithley6517B` method), 329  
method), 329 `wait_time()` (`pymeasure.instruments.agilent.agilentB1500.AgilentB1500` method), 186  
`wait_for_buffer()` (`pymeasure.instruments.signalrecovery.DSP7225` method), 433  
method), 433 `WaitTimeType` (class in `pymeasure.instruments.agilent.agilentB1500`), 196  
`wait_for_buffer()` (`pymeasure.instruments.signalrecovery.DSP7265` method), 439  
method), 439 `wave` (`pymeasure.instruments.fakes.SwissArmyFake` property), 113  
`wait_for_buffer()` (`pymeasure.instruments.srs.SR830` method), 446  
method), 446 `waveform_abort()` (`pymeasure.instruments.keithley.Keithley6221` method), 324  
`wait_for_completion()` (`pymeasure.instruments.anaheimautomation.DPSeriesMotorController` method), 204  
method), 204 `waveform_amplitude` (`pymeasure.instruments.keithley.Keithley6221` property), 324  
`wait_for_current()` (`pymeasure.instruments.danfysik.Danfysik8500` method), 229  
method), 229 `waveform_arm()` (`pymeasure.instruments.keithley.Keithley6221` method), 324  
`wait_for_data()` (`pymeasure.experiment.experiment.Experiment` method), 72  
method), 72 `waveform_data` (`pymeasure.instruments.keysight.KeysightDSOX1102G` property), 341  
`wait_for_holding()` (`pymeasure.instruments.ami.AMI430` method), 202  
method), 202 `waveform_duration_cycles` (`pymeasure.instruments.keithley.Keithley6221` property), 324  
`wait_for_idle()` (`pymeasure.instruments.oxfordinstruments.IPS120_10` method), 401  
method), 401 `waveform_duration_set_infinity()` (`pymeasure.instruments.keithley.Keithley6221` method), 324  
`wait_for_output_voltage_reached()` (`pymeasure.instruments.eurotest.EurotestHPPI20256` method), 324  
method), 324 `waveform_duration_time` (`pymeasure.instruments.keithley.Keithley6221` property), 324

<code>sure.instruments.keithley.Keithley6221</code> (property), 324	<code>sure.instruments.keysight.KeysightDSOX1102G</code> (property), 341
<code>waveform_dutycycle</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 324	<code>waveform_sparsing</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> property), 374
<code>waveform_first_point</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> property), 373	<code>waveform_sparsing</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> property), 473
<code>waveform_first_point</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> property), 472	<code>waveform_start()</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> method), 325
<code>waveform_format</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> property), 341	<code>waveform_use_phasemarker</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 325
<code>waveform_frequency</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 324	<code>waveforms</code> ( <code>pymeasure.instruments.activetechnologies.AWG401x_AWG</code> property), 119
<code>waveform_function</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 324	<code>wavelength</code> ( <code>pymeasure.instruments.keysight.KeysightN7776C</code> property), 343
<code>waveform_offset</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 324	<code>wavelength</code> ( <code>pymeasure.instruments.thorlabs.ThorlabsPM100USB</code> property), 489
<code>waveform_phasemarker_line</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 324	<code>wavelength_center</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_phasemarker_phase</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 325	<code>wavelength_marker_value</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_points</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> property), 341	<code>wavelength_max</code> ( <code>pymeasure.instruments.thorlabs.ThorlabsPM100USB</code> property), 489
<code>waveform_points</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> property), 373	<code>wavelength_min</code> ( <code>pymeasure.instruments.thorlabs.ThorlabsPM100USB</code> property), 489
<code>waveform_points</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> property), 472	<code>wavelength_span</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_points_mode</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> property), 341	<code>wavelength_start</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_preamble</code> ( <code>pymeasure.instruments.keysight.KeysightDSOX1102G</code> property), 341	<code>wavelength_stop</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_preamble</code> ( <code>pymeasure.instruments.lecroy.LeCroyT3DSO1204</code> property), 373	<code>wavelength_temperature</code> ( <code>pymeasure.instruments.ipgphotonics.yar.YAR</code> property), 287
<code>waveform_preamble</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> property), 472	<code>wavelength_value_in</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_ranging</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 325	<code>wavelengths</code> ( <code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> property), 210
<code>waveform_source</code> ( <code>pymeasure.instruments.keithley.Keithley6221</code> property), 325	<code>wavetype</code> ( <code>pymeasure.instruments.teledyne.teledyneT3AFG.SignalChannel</code> property), 468
	<code>WindowType</code> (class in <code>pymeasure.instruments.hp.hp856Xx</code> ), 280
	<code>wipe_sweep_table()</code> ( <code>pymeasure.instruments.teledyne.TeledyneOscilloscope</code> method), 472

`sure.instruments.oxfordinstruments.ITC503`  
`method`), 398

`wires` (`pymeasure.instruments.keithley.Keithley2400`  
`property`), 307

`wires` (`pymeasure.instruments.keithley.Keithley2450`  
`property`), 314

`wl_logging` (`pymeasure.instruments.keysight.KeysightN7776C`  
`property`), 343

`Worker` (class in `pymeasure.experiment.workers`), 79

`write()` (`pymeasure.adapters.Adapter` method), 49

`write()` (`pymeasure.adapters.FakeAdapter` method), 69

`write()` (`pymeasure.adapters.PrologixAdapter` method),  
60

`write()` (`pymeasure.adapters.SerialAdapter` method), 55

`write()` (`pymeasure.adapters.TelnetAdapter` method), 65

`write()` (`pymeasure.adapters.VISAAdapter` method), 52

`write()` (`pymeasure.adapters.VXI11Adapter` method),  
62

`write()` (`pymeasure.instruments.advantest.advantestR624X`  
`method`), 123

`write()` (`pymeasure.instruments.agilent.agilentB1500.SMU`  
`method`), 188

`write()` (`pymeasure.instruments.anaheimautomation.DPSeriesMotionControl`  
`method`), 204

`write()` (`pymeasure.instruments.andeenhagerling.AH2700A`  
`method`), 207

`write()` (`pymeasure.instruments.Channel` method), 112

`write()` (`pymeasure.instruments.eurotest.EurotestHPP120366`  
`method`), 233

`write()` (`pymeasure.instruments.fwbell.FWBell5080`  
`method`), 236

`write()` (`pymeasure.instruments.hcp.TC038` method),  
238

`write()` (`pymeasure.instruments.hcp.TC038D` method),  
239

`write()` (`pymeasure.instruments.hp.HP34401A`  
`method`), 243

`write()` (`pymeasure.instruments.hp.HP8116A` method),  
250

`write()` (`pymeasure.instruments.hp.HPLegacyInstrument`  
`method`), 284

`write()` (`pymeasure.instruments.Instrument` method),  
111

`write()` (`pymeasure.instruments.keithley.Keithley2000`  
`method`), 294

`write()` (`pymeasure.instruments.keithley.Keithley2260B`  
`method`), 297

`write()` (`pymeasure.instruments.keithley.Keithley2306`  
`method`), 299

`write()` (`pymeasure.instruments.keithley.Keithley2400`  
`method`), 307

`write()` (`pymeasure.instruments.keithley.Keithley2450`  
`method`), 315

`write()` (`pymeasure.instruments.keithley.Keithley2600`  
`method`), 334

`write()` (`pymeasure.instruments.keithley.Keithley2700`  
`method`), 318

`write()` (`pymeasure.instruments.keithley.Keithley2750`  
`method`), 332

`write()` (`pymeasure.instruments.keithley.Keithley6221`  
`method`), 325

`write()` (`pymeasure.instruments.keithley.Keithley6517B`  
`method`), 330

`write()` (`pymeasure.instruments.keysight.KeysightE36312A`  
`method`), 350

`write()` (`pymeasure.instruments.lakeshore.LakeShore421`  
`method`), 358

`write()` (`pymeasure.instruments.lecroy.LeCroyT3DSO1204`  
`method`), 374

`write()` (`pymeasure.instruments.mksinst.mks937b.MKS937B`  
`method`), 376

`write()` (`pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInput`  
`method`), 381

`write()` (`pymeasure.instruments.oxfordinstruments.base.OxfordInstrument`  
`method`), 394

`write()` (`pymeasure.instruments.signalrecovery.DSP7225`  
`method`), 433

`write()` (`pymeasure.instruments.signalrecovery.DSP7265`  
`method`), 439

`write()` (`pymeasure.instruments.tcpowerconversion.CXN`  
`method`), 455

`write()` (`pymeasure.instruments.tdk.tdk_gen40_38.TDK_Gen40_38`  
`method`), 460

`write()` (`pymeasure.instruments.tdk.tdk_gen80_65.TDK_Gen80_65`  
`method`), 464

`write()` (`pymeasure.instruments.teledyne.TeledyneOscilloscope`  
`method`), 473

`write()` (`pymeasure.instruments.teledyne.TeledyneT3AFG`  
`method`), 467

`write()` (`pymeasure.instruments.texio.TexioPSW360L30`  
`method`), 486

`write()` (`pymeasure.instruments.thermotron.Thermotron3800`  
`method`), 488

`write()` (`pymeasure.instruments.thyracont.smartline_v1.SmartlineV1`  
`method`), 491

`write()` (`pymeasure.instruments.thyracont.smartline_v2.SmartlineV2`  
`method`), 494

`write()` (`pymeasure.instruments.velleman.VellemanK8090`  
`method`), 498

`write_binary_values()` (`pymeasure.adapters.Adapter`  
`method`), 49

`write_binary_values()` (`pymeasure.adapters.FakeAdapter` method), 69

`write_binary_values()` (`pymeasure.adapters.PrologixAdapter`  
`method`),  
60

`write_binary_values()` (`pymeasure.adapters.SerialAdapter` method), 55

`write_binary_values()` (*pymeasure.adapters.TelnetAdapter* method), 65  
`write_binary_values()` (*pymeasure.adapters.VISAAadapter* method), 52  
`write_binary_values()` (*pymeasure.adapters.VXIIAdapter* method), 63  
`write_binary_values()` (*pymeasure.instruments.andenhagerling.AH2700A* method), 207  
`write_binary_values()` (*pymeasure.instruments.Channel* method), 112  
`write_binary_values()` (*pymeasure.instruments.fwbell.FWBell5080* method), 236  
`write_binary_values()` (*pymeasure.instruments.Instrument* method), 111  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2000* method), 294  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2200* method), 338  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2260B* method), 297  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2306* method), 299  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2400* method), 307  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2450* method), 315  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2600* method), 334  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2700* method), 319  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley2750* method), 332  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley6221* method), 325  
`write_binary_values()` (*pymeasure.instruments.keithley.Keithley6517B* method), 330  
`write_binary_values()` (*pymeasure.instruments.keysight.KeysightE36312A* method), 350  
`write_binary_values()` (*pymeasure.instruments.lecroy.LeCroyT3DSO1204* method), 374  
`write_binary_values()` (*pymeasure.instruments.signalrecovery.DSP7225* method), 433  
`write_binary_values()` (*pymeasure.instruments.signalrecovery.DSP7265* method), 440  
`write_binary_values()` (*pymeasure.instruments.tdk.tdk\_gen40\_38.TDK\_Gen40\_38* method), 460  
`write_binary_values()` (*pymeasure.instruments.tdk.tdk\_gen80\_65.TDK\_Gen80\_65* method), 464  
`write_binary_values()` (*pymeasure.instruments.teledyne.TeledyneT3AFG* method), 467  
`write_binary_values()` (*pymeasure.instruments.texio.TexioPSW360L30* method), 487  
`write_bytes()` (*pymeasure.adapters.Adapter* method), 49  
`write_bytes()` (*pymeasure.adapters.FakeAdapter* method), 69  
`write_bytes()` (*pymeasure.adapters.PrologixAdapter* method), 60  
`write_bytes()` (*pymeasure.adapters.SerialAdapter* method), 55  
`write_bytes()` (*pymeasure.adapters.TelnetAdapter* method), 66  
`write_bytes()` (*pymeasure.adapters.VISAAadapter* method), 52  
`write_bytes()` (*pymeasure.adapters.VXIIAdapter* method), 63  
`write_bytes()` (*pymeasure.instruments.andenhagerling.AH2700A* method), 208  
`write_bytes()` (*pymeasure.instruments.Channel* method), 112  
`write_bytes()` (*pymeasure.instruments.fwbell.FWBell5080* method), 237  
`write_bytes()` (*pymeasure.instruments.Instrument* method), 111  
`write_bytes()` (*pymeasure.instruments.keithley.Keithley2000* method), 295  
`write_bytes()` (*pymeasure.instruments.keithley.Keithley2200* method), 338  
`write_bytes()` (*pymeasure.instruments.keithley.Keithley2260B* method), 297  
`write_bytes()` (*pymeasure.instruments.keithley.Keithley2306* method), 297

- [method](#)), 299
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley2400](#) [method](#)), 308
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley2450](#) [method](#)), 315
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley2600](#) [method](#)), 334
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley2700](#) [method](#)), 319
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley2750](#) [method](#)), 332
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley6221](#) [method](#)), 325
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keithley.Keithley6517B](#) [method](#)), 330
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.keysight.KeysightE36312A](#) [method](#)), 350
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.lecroy.LeCroyT3DSO1204](#) [method](#)), 374
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [method](#)), 433
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [method](#)), 440
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.tdk.tdk\\_gen40\\_38.TDK\\_Gen40\\_38](#) [method](#)), 460
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.tdk.tdk\\_gen80\\_65.TDK\\_Gen80\\_65](#) [method](#)), 464
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.teledyne.TeledyneT3AFG](#) [method](#)), 467
  - [write\\_bytes\(\)](#) ([pymeasure.instruments.texio.TexioPSW360L30](#) [method](#)), 487
  - [write\\_calibration\\_data\(\)](#) ([pymeasure.instruments.hp.HP3478A](#) [method](#)), 248
  - [write\\_composition\(\)](#) ([pymeasure.instruments.thyracont.smartline\\_v2.SmartlineV2](#) [method](#)), 494
  - [write\\_file\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_getter\\_test\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_init\\_test\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_method\\_test\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_method\\_tests\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_property\\_tests\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [write\\_raw\(\)](#) ([pymeasure.adapters.VXI11Adapter](#) [method](#)), 63
  - [write\\_setter\\_test\(\)](#) ([pymeasure.generator.Generator](#) [method](#)), 70
  - [writeA0\(\)](#) (in module [pymeasure.instruments.comedi](#)), 116
- ## X
- [x](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200
  - [x](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [property](#)), 433
  - [x](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [property](#)), 440
  - [x](#) ([pymeasure.instruments.srs.SR830](#) [property](#)), 446
  - [x](#) ([pymeasure.instruments.srs.SR860](#) [property](#)), 451
  - [x1](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200
  - [x2](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200
  - [x\\_pointer](#) ([pymeasure.instruments.oxfordinstruments.ITC503](#) [property](#)), 398
  - [xroll\\_frequency](#) ([pymeasure.instruments.hp.hp856Xx.HP856Xx](#) [attribute](#)), 269
  - [xy](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200
  - [xy](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [property](#)), 433
  - [xy](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [property](#)), 440
  - [xy](#) ([pymeasure.instruments.srs.SR830](#) [property](#)), 446
- ## Y
- [y](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200
  - [y](#) ([pymeasure.instruments.signalrecovery.DSP7225](#) [property](#)), 433
  - [y](#) ([pymeasure.instruments.signalrecovery.DSP7265](#) [property](#)), 440
  - [y](#) ([pymeasure.instruments.srs.SR830](#) [property](#)), 446
  - [y](#) ([pymeasure.instruments.srs.SR860](#) [property](#)), 451
  - [y1](#) ([pymeasure.instruments.ametek.Ametek7270](#) [property](#)), 200

`y2` (`pymeasure.instruments.ametek.Ametek7270` property), [200](#)  
`y_pointer` (`pymeasure.instruments.oxfordinstruments.ITC503` property), [398](#)  
`YAR` (class in `pymeasure.instruments.ipgphotonics.yar`), [286](#)  
`YAR.Status` (class in `pymeasure.instruments.ipgphotonics.yar`), [286](#)  
`Yokogawa7651` (class in `pymeasure.instruments.yokogawa`), [499](#)  
`YokogawaGS200` (class in `pymeasure.instruments.yokogawa`), [501](#)

## Z

`zero()` (`pymeasure.instruments.ami.AMI430` method), [202](#)  
`zero()` (`pymeasure.instruments.newport.esp300.Axis` method), [378](#)  
`zero_probe()` (`pymeasure.instruments.lakeshore.LakeShore421` method), [358](#)  
`zero_probe()` (`pymeasure.instruments.lakeshore.LakeShore425` method), [359](#)