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 References
- About PyMeasure

Information about development is also available:

- Getting involved
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.

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

### 2.1 Setting up Python

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

Download and install the appropriate Python 3.5 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):

```bash
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.
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 Checking the version

Now that you have Python and PyMeasure installed, open up a “Jupyter Notebook” to test which version you have installed. Execute the following code into a notebook cell.

```python
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?
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.

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

Then construct an object by passing the GPIB address. For this example we connect to the instrument over GPIB (using VISA) with an address of 4. See the *adapters* section below for more details.

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

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

```python
sourcemeter.id
```

This is equivalent to manually calling the SCPI command.

```python
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.

```python
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.

### 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 class to connect to the instrument using VISA. Instead of passing a string, we could equally pass an adapter object.

```python
from pymeasure.adapters import VISAAdapter
adapter = VISAAdapter("GPIB::4")
sourcemeter = Keithely2400(adapter)
```

To instead use a Prologix GPIB device connected on `/dev/ttyUSB0` (proper permissions are needed in Linux, see `PrologixAdapter`), the adapter is constructed in a similar way. Unlike the VISA adapter which is specific to each instrument, the Prologix adapter can be shared by many instruments. Therefore, they are addressed separately based on the GPIB address number when passing the adapter into the instrument construction.

```python
from pymeasure.adapters import PrologixAdapter
adapter = PrologixAdapter('/dev/ttyUSB0')
sourcemeter = Keithley2400(adapter.gpib(4))
```

For instruments using serial communication that have particular settings that need to be matched, a custom `Adapter` sub-class can be made. For example, the LakeShore 425 Gaussmeter connects via USB, but uses particular serial communication settings. Therefore, a `LakeShoreUSBAdapter` class enables these requirements in the background.

```python
from pymeasure.instruments.lakeshore import LakeShore425
gaussmeter = LakeShore425('/dev/lakeshore425')
```

Behind the scenes the `/dev/lakeshore425` port is passed to the `LakeShoreUSBAdapter`.

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.

```python
from pymeasure.adapters import VXI11Adapter
from pymeasure.instruments import Instrument
adapter = VXI11Adapter("TCPIP::192.168.0.100::inst0::INSTR")
instr = Instrument(adapter, "my_instrument")
```

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. Connect to the Keithley 2400
4. Set up the instrument for the IV characteristic
5. Allocate arrays to store the resulting measurements
6. Loop through the current points, measure the voltage, and record
7. Save the final data to a CSV file
8. Shutdown the instrument

These steps are expressed in code as follows.

```python
# 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 = 50
max_current = 0.01
min_current = -max_current

# Connect and configure the instrument
sourcemeter = Keithley2400("GPIB::4")
sourcemeter.reset()
sourcemeter.use_front_terminals()
sourcemeter.measure_voltage()
sourcemeter.config_current_source()
sleep(0.1)  # wait here to give the instrument time to react
sourcemeter.set_buffer(averages)

# 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)

# Loop through each current point, measure and record the voltage
for i in range(data_points):
    sourcemeter.current = currents[i]
    sourcemeter.reset_buffer()
    sleep(0.1)
    sourcemeter.start_buffer()
    sourcemeter.wait_for_buffer()
```

(continues on next page)
# Record the average and standard deviation
voltages[i] = sourcemeter.means
voltage_stds[i] = sourcemeter.standard_devs

# 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')
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`.

```python
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):
    """ Loops over each iteration and emits the current iteration, before waiting for 0.01 sec, and then checking if the procedure should stop """
```
At the top of the SimpleProcedure class we define the required Parameters. In this case, `iterations` is an 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.

```python
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.

```python
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.

3.2. Making a measurement
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):
        """ 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.

```python
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.

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.

```python
# Import necessary packages
from pymeasure.instruments.keithley import Keithley2400
from pymeasure.experiment import Procedure
from pymeasure.experiment import IntegerParameter, FloatParameter
from time import sleep

class IVProcedure(Procedure):
    data_points = IntegerParameter('Data points', default=50)
    averages = IntegerParameter('Averages', default=50)
    max_current = FloatParameter('Maximum Current', units='A', default=0.01)
    min_current = FloatParameter('Minimum Current', units='A', default=-0.01)

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

    def startup(self):
        log.info("Connecting and configuring the instrument")
        self.sourcemeter = Keithley2400("GPIB::4")
        self.sourcemeter.reset()
        self.sourcemeter.use_front_terminals()
        self.sourcemeter.measure_voltage()
        self.sourcemeter.config_current_source()
        sleep(0.1)  # wait here to give the instrument time to react
        self.sourcemeter.set_buffer(averages)

    def execute(self):
        currents = np.linspace(self.min_current, self.max_current, num=self.data_points)

        # Loop through each current point, measure and record the voltage
        for current in currents:
            log.info("Setting the current to \%g A" % current)
            self.sourcemeter.current = current
            self.sourcemeter.reset_buffer()
            sleep(0.1)
            self.sourcemeter.start_buffer()
            log.info("Waiting for the buffer to fill with measurements")
            self.sourcemeter.wait_for_buffer()

            self.emit('results', {
                'Current (A)': current,
                'Voltage (V)': self.sourcemeter.means,
                'Voltage Std (V)': self.sourcemeter.standard_devs
            })
            sleep(0.01)
```

(continues on next page)
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 = 100
    procedure.averages = 50
    procedure.max_current = -0.01
    procedure.min_current = 0.01

    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")

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-featured 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.

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.

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

(continues on next page)
import random
from time import sleep
from pymeasure.log import console_log
from pymeasure.display import Plotter
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

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

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

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

    def execute(self):
        log.info("Starting the loop of \$d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
            }
            self.emit('results', data)
            log.debug("Emitting results: \%s" % data)
            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")
The important addition is the construction of the Plotter from the Results object.

```python
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.

```python
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 QtGui
from pymeasure.display.windows import ManagedWindow
from pymeasure.experiment import Procedure, Results
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

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

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

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

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

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

(continues on next page)
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__":
    app = QtGui.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.

### 3.3.3 Customising the plot options

For both the PlotterWindow and ManagedWindow, plotting is provided by the [pyqtgraph](https://www.pyqtgraph.org/) 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:

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

For `ManagedWindow`, Similarly to the Plotter, the `setup_plot()` method can be overridden by your sub-class in order to do the set-up

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

It is also possible to access the `plot` attribute while outside of your sub-class, for example we could modify the previous section’s example

```python
if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)
    window = MainWindow()
    window.plot.setLogMode(x=True)  # use logarithmic x-axis (e.g. for frequency sweeps)
    window.show()
    sys.exit(app.exec_())
```

See pyqtgraph’s API documentation on `PlotItem` for further details.

## 3.3.4 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 `MainWindow` has to be modified slightly (where modified lines are marked):

```python
class MainWindow(ManagedWindow):
    def __init__(self):
        super(MainWindow, self).__init__(
            procedure_class=TestProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
```
This adds the sequencer underneath the input panel.
The widget contains a tree-view where you can build the sequence. It has three columns: level (indicated how deep an item is nested), parameter (a drop-down menu to select which parameter is being sequenced by that item), and sequence (the text-box where you can define the sequence). While the two former columns are rather straightforward, filling in the later requires some explanation.

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

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

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

Finally, it is possible to write a simple text file to quickly load a pre-defined sequence with the Load sequence button, such that the user does not need to write the sequence again each time. 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(MainWindow, self).__init__` call, as was done in the example.
The adapter classes allow the instruments to be independent of the communication method used. Adapters for specific instruments should be grouped in an `adapters.py` file in the corresponding manufacturer’s folder of `pymeasure.instruments`. For example, the adapter for communicating with LakeShore instruments over USB, `LakeShoreUSBAdapter`, is found in `pymeasure.instruments.lakeshore.adapters`.

## 4.1 Adapter base class

```python
class pymeasure.adapters.Adapter
    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.

    ask(command)
        Writes the command to the instrument and returns the resulting ASCII response

        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

        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

    read()
        Reads until the buffer is empty and returns the resulting ASCII response
```
Returns String ASCII response of the instrument.

values(command, separator=’,’, cast=<class ‘float’>)

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

Parameters

- command – SCPI command to be sent to the instrument
- separator – A separator character to split the string into a list
- cast – A type to cast the result

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

write(command)

Writes a command to the instrument

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

4.2 Fake adapter

class pymeasure.adapters.FakeAdapter
    Bases: pymeasure.adapters.adapter.Adapter

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

```python
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)

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

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

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

read()

Returns the last commands given after the last read call.

values(command, separator=’,’, cast=<class ‘float’>)

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

Parameters
• **command** – SCPI command to be sent to the instrument
• **separator** – A separator character to split the string into a list
• **cast** – A type to cast the result

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

### 4.3 Serial adapter

```python
class pymeasure.adapters.SerialAdapter(port, **kwargs)
```

Bases: pymeasure.adapters.adapter.Adapter

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

**Parameters**

- **port** – Serial port
- **kwargs** – Any valid key-word argument for serial.Serial

```python
ask(command)
```

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

**Parameters**

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

**Returns**
String ASCII response of the instrument

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

Returns a numpy array from a query for binary data

**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

```python
read()
```

Reads until the buffer is empty and returns the resulting ASCII response

**Returns**
String ASCII response of the instrument.

```python
values(command, separator=' ', cast=<class 'float'>)
```

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

**Parameters**

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result

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

```python
write(command)
```

Writes a command to the instrument

**Parameters**

- **command** – SCPI command string to be sent to the instrument
4.4 Prologix adapter

```python
class pymeasure.adapters.PrologixAdapter(port, address=None, rw_delay=None, serial_timeout=0.5, **kwargs)

Bases: pymeasure.adapters.serial.SerialAdapter

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

Each PrologixAdapter is constructed based on a serial port or connection and the GPIB address to be communicated to. Serial connection sharing is achieved by using the `gpib()` method to spawn new PrologixAdapters for different GPIB addresses.

Parameters

- **port** – The Serial port name or a serial.Serial object
- **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.
- **kwargs** – Key-word arguments if constructing a new serial object

Variables

- **address** – Integer GPIB address of the desired instrument

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
```

**ask(command)**

Ask the Prologix controller, include a forced delay for some instruments.

**Parameters**

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

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

Returns a numpy array from a query for binary data

**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

**gpib(address, rw_delay=None)**

Returns and 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
- **rw_delay** – Set a custom Read/Write delay for the instrument

**Returns**

PrologixAdapter for specific GPIB address
read()
Reads the response of the instrument until timeout

Returns String ASCII response of the instrument

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

values(command, separator='.', cast=<class 'float'>)
Writes a command to the instrument and returns a list of formatted values from the result

Parameters
• command – SCPI command to be sent to the instrument
• separator – A separator character to split the string into a list
• cast – A type to cast the result

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

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

Parameters
• timeout – Timeout duration in seconds
• delay – Time delay between checking SRQ in seconds

write(command)
Writes the command to the GPIB address stored in the address

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

4.5 VISA adapter

class pymeasure.adapters.VISAAdapter(resourceName, visa_library='', **kwargs)
Bases: pymeasure.adapters.adapter.Adapter

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

Parameters
• resource – VISA resource name that identifies the address
• visa_library – VisaLibrary Instance, path of the VISA library or VisaLibrary spec string (@py or @ni). if not given, the default for the platform will be used.
• kwargs – Any valid key-word arguments for constructing a PyVISA instrument

ask(command)
Writes the command to the instrument and returns the resulting ASCII response

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

Returns String ASCII response of the instrument

ask_values(command)
Writes a command to the instrument and returns a list of formatted values from the result. The format of the return is configurated by self.config().

Parameters command – SCPI command to be sent to the instrument

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

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

cfg(is_binary=False, datatype='str', container=<built-in function array>, converter='s', separator=', ', is_big_endian=False)  
Configurate the format of data transfer to and from the instrument.

Parameters

- **is_binary** – If True, data is in binary format, otherwise ASCII.
- **datatype** – Data type.
- **container** – Return format. Any callable/type that takes an iterable.
- **converter** – String converter, used in dealing with ASCII data.
- **separator** – Delimiter of a series of data in ASCII.
- **is_big_endian** – Endianness.

static has_supported_version()  
Returns True if the PyVISA version is greater than 1.8

read()  
Reads until the buffer is empty and returns the resulting ASCII response

Returns String ASCII response of the instrument.

read_bytes(size)  
Reads specified number of bytes from the buffer and returns the resulting ASCII response

Parameters **size** – Number of bytes to read from the buffer

Returns String ASCII response of the instrument.

gvalues(command, separator=', ', cast=<class 'float'>)  
Writes a command to the instrument and returns a list of formatted values from the result

Parameters

- **command** – SCPI command to be sent to the instrument
- **separator** – A separator character to split the string into a list
- **cast** – A type to cast the result

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

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

Parameters

- **timeout** – Timeout duration in seconds
- **delay** – Time delay between checking SRQ in seconds
write(command)

    Writes a command to the instrument

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

4.6 VXI-11 adapter

class pymeasure.adapters.VXI11Adapter (host, **kwargs)

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

   Parameters host – string containing the visa connection information.

ask(command)

    Wrapper function for the ask command using the vx11 interface.

   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.

   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

   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

read()

    Wrapper function for the read command using the vx11 interface.

    :return string containing a response from the device.

read_raw()

    Wrapper function for the read_raw command using the vx11 interface.

    :returns binary string containing the response from the device.

values (command, separator=',', cast=<class 'float'>)

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

   Parameters

      • command – SCPI command to be sent to the instrument
      • separator – A separator character to split the string into a list
      • cast – A type to cast the result
Returns A list of the desired type, or strings where the casting fails

write(command)
Wrapper function for the write command using the vxi11 interface.

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

write_raw(command)
Wrapper function for the write_raw command using the vxi11 interface.

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

```python
class pymeasure.experiment.Experiment:
    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.
data
  Data property which returns analysed data, if an analyse function is defined, otherwise returns the raw data.
pcolor(xname, yname, zname, *args, **kwargs)
  Plot the results from the experiment.data pandas dataframe in a pcolor graph. Store the plots in a plots list attribute.
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_pcolor(ax, xname, yname, zname)
  Update a pcolor graph with new data.
update_plot()
  Update the plots in the plots list with new data from the experiment.data pandas dataframe.
wait_for_data()
  Wait for the data attribute to fill with datapoints.
pymeasure.experiment.experiment.create_filename(title)
  Create a new filename according to the style defined in the config file. If no config is specified, create a temporary file.
pymeasure.experiment.experiment.get_array(start, stop, step)
  Returns a numpy array from start to stop
pymeasure.experiment.experiment.get_array_steps(start, stop, numsteps)
  Returns a numpy array from start to stop in numsteps
pymeasure.experiment.experiment.get_array_zero(maxval, step)
  Returns a numpy array from 0 to maxval to -maxval to 0

5.2 Listener class

class pymeasure.experiment.listeners.Listener(port, topic=",", timeout=0.01)
  Bases: pymeasure.thread.StoppableThread
  Base class for Threads that need to listen for messages on a ZMQ TCP port and can be stopped by a thread-safe method call
message_waiting()
receive(flags=0)

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

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

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

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.

```python
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
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.

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

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

parameters_are_set()
    Returns True if all parameters are set

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

set_parameters(parameters, except_missing=True)
    Sets a dictionary of parameters and raises an exception if additional parameters are present if except_missing is True
shutdown()
Executes the commands necessary to shut down the instruments and leave them in a safe state. This method is always run at the end.

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

class pymeasure.experiment.procedure.UnknownProcedure(parameters)
Handles the case when a Procedure object can not be imported during loading in the Results class
startup()
Executes the commands needed at the start-up of the measurement

## 5.4 Parameter classes

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

```python
class pymeasure.experiment.parameters.BooleanParameter(name, default=None, ui_class=None)
```

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

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

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

```python
class pymeasure.experiment.parameters.FloatParameter(name, units=None, minimum=-1000000000.0, maximum=1000000000.0, **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)
- **default** – The default floating point value
- **ui_class** – A Qt class to use for the UI of this parameter

```python
class pymeasure.experiment.parameters.IntegerParameter(name, units=None, minimum=-1000000000.0, maximum=1000000000.0, **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

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

*Parameter* sub-class that stores the value as a list.

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

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.Parameter(name, default=None, ui_class=None)

Encapsulates the information for an experiment parameter with information about the name, and units if supplied.

Variables

• **value** – The value of the parameter

Parameters

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

is_set()
Returns True if the Parameter value is set

5.4. Parameter classes
class pymeasure.experiment.parameters.PhysicalParameter(name, uncertaintyType='absolute', **kwargs)

VectorParameter sub-class of 2 dimensions to store a value and its uncertainty.

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

Parameters

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

class pymeasure.experiment.parameters.VectorParameter(name, length=3, units=None, **kwargs)

Parameter sub-class that stores the value in a vector format.

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

Parameters

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

5.5 Worker class

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

Bases: pymeasure.thread.StoppableThread

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

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

handle_abort ()

handle_error ()

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

Parameters timeout – Timeout duration in seconds

run ()
Method representing the thread’s activity.

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

5.6 Results class

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

format(record)
Formats a record as csv.

Parameters record (dict) – record to format.

Returns a string

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

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

Parameters
• procedure – Procedure object
• data_filename – The data filename where the data is or should be stored

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

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

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

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

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

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

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

pymeasure.experiment.results.unique_filename(directory, prefix='DATA', suffix='', ext='csv', dated_folder=False, index=True, datetimeformat='%Y-%m-%d')

Returns a unique filename based on the directory and prefix
This section contains specific documentation on the classes and methods of the package.

### 6.1 Browser classes

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

Bases: sphinx.ext.autodoc.importer._MockObject
```
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.

```python
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.
```

```python
class pymeasure.display.browser.BrowserItem(results, curve, parent=None)
Bases: sphinx.ext.autodoc.importer._MockObject
```

### 6.2 Curves classes

```python
class pymeasure.display.curves.BufferCurve(errors=False, **kwargs)
Bases: sphinx.ext.autodoc.importer._MockObject
```
Creates a curve based on a predefined buffer size and allows data to be added dynamically, in addition to supporting error bars.
append \((x, y, xError=None, yError=None)\)
Appends data to the curve with optional errors

prepare \((size, dtype=<\text{class} \text{ numpy.float32}>)\)
Prepares the buffer based on its size, data type

class pymeasure.display.curves.Crosshairs \((\text{plot}, \text{pen}=None)\)
Bases: sphinx.ext.autodoc.importer._MockObject
Attaches crosshairs to the a plot and provides a signal with the x and y graph coordinates

mouseMoved \((\text{event}=None)\)
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 \((\text{results, x, y, xerr=None, yerr=None, force_reload=False, **kwargs})\)
Bases: sphinx.ext.autodoc.importer._MockObject
Creates a curve loaded dynamically from a file through the Results object and supports error bars. 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()
Updates the data by polling the results

class pymeasure.display.curves.ResultsImage \((\text{results, x, y, z, force_reload=False})\)
Bases: sphinx.ext.autodoc.importer._MockObject
Creates an image loaded dynamically from a file through the Results object.

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

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

6.3 Inputs classes

class pymeasure.display.inputs.BooleanInput \((\text{parameter, parent}=None, **kwargs)\)
Bases: sphinx.ext.autodoc.importer._MockObject, pymeasure.display.inputs.Input
Checkbox for boolean values, connected to a BooleanParameter.

set_parameter \((\text{parameter})\)
Connects a new parameter to the input box, and initializes the box value.

Parameters parameter – parameter to connect.

class pymeasure.display.inputs.FloatInput \((\text{parameter, parent}=None, **kwargs)\)
Bases: sphinx.ext.autodoc.importer._MockObject, pymeasure.display.inputs.Input
Spin input box for floating-point values, connected to a FloatParameter.

See also:
Class **ScientificInput** For inputs in scientific notation.

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

**Parameters**

- **parameter** – parameter to connect.

```python
class pymeasure.display.inputs.Input(parameter, **kwargs)
```
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.

**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.

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

**Parameters**

- **parameter** – parameter to connect.

```python
update_parameter()
```
Update the parameter value with the Input GUI element’s current value.

```python
class pymeasure.display.inputs.IntegerInput(parameter, parent=None, **kwargs)
```
Spin input box for integer values, connected to a **IntegerParameter**.

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

**Parameters**

- **parameter** – parameter to connect.

```python
class pymeasure.display.inputs.ListInput(parameter, parent=None, **kwargs)
```
Dropdown for list values, connected to a **ListParameter**.

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

**Parameters**

- **parameter** – parameter to connect.

```python
class pymeasure.display.inputs.ScientificInput(parameter, parent=None, **kwargs)
```
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.

```python
set_parameter(parameter)
```
Connects a new parameter to the input box, and initializes the box value.
Parameters `parameter` – parameter to connect.

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

Bases: sphinx.ext.autodoc.importer._MockObject, pymeasure.display.inputs.Input

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

### 6.4 Listeners classes

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

Bases: sphinx.ext.autodoc.importer._MockObject

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

class pymeasure.display.listeners.QListener(`port`, `topic="", timeout=0.01`)

Bases: pymeasure.display.thread.StoppableQThread

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

### 6.5 Log classes

class pymeasure.display.log.LogHandler(`parent=None`)

Bases: sphinx.ext.autodoc.importer._MockObject, logging.Handler

emit (`record`)

Do whatever it takes to actually log the specified logging record.

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

### 6.6 Manager classes

class pymeasure.display.manager.Experiment(`results`, `curve`, `browser_item`, `parent=None`)

Bases: sphinx.ext.autodoc.importer._MockObject

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

Parameters

- `results` – `Results` object
- `curve` – `ResultsCurve` object
- `browser_item` – `BrowserItem` object

class pymeasure.display.manager.ExperimentQueue

Bases: sphinx.ext.autodoc.importer._MockObject

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.ImageExperiment(results, curve, image, browser_item, parent=None)
Bases: pymeasure.display.manager.Experiment
    Adds saving of the experiments image to Experiment. Needed to make image features work

class pymeasure.display.manager.ImageExperimentQueue
Bases: pymeasure.display.manager.ExperimentQueue
    Overwrites needed features from ExperimentQueue to make image features work

class pymeasure.display.manager.ImageManager(plot, im_plot, browser, port=5888, log_level=20, parent=None)
Bases: pymeasure.display.manager.Manager
    Overwrites needed features from Manager to make image features work

load(experiment)
    Load a previously executed Experiment

remove(experiment)
    Removes an Experiment

class pymeasure.display.manager.Manager(plot, browser, port=5888, log_level=20, parent=None)
Bases: sphinx.ext.autodoc.importer._MockObject
    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 PyQt-Graph PlotItem within the user interface, which are updated in accordance with the execution status of the Experiments.

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

clear()
    Remove all Experiments

is_running()
    Returns True if a procedure is currently running

load(experiment)
    Load a previously executed Experiment

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

queue(experiment)
    Adds an experiment to the queue.

remove(experiment)
    Removes an Experiment

resume()
    Resume processing of the queue.

6.6. Manager classes
6.7 Plotter class

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

Plotter dynamically plots data from a file through the Results object and supports error bars.

See also:

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

run()
    Method representing the thread’s activity.

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

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

        Parameters plot – This window’s PlotItem instance.

6.8 Qt classes

All Qt imports should reference pymeasure.display.Qt, for 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 accessability.

6.9 Thread classes

class pymeasure.display.thread.StoppableQThread(parent=None)
    Bases: sphinx.ext.autodoc.importer._MockObject

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

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

        Parameters timeout – Timeout duration in seconds

6.10 Widget classes

class pymeasure.display.widgets.BrowserWidget(*args, parent=None)
    Bases: sphinx.ext.autodoc.importer._MockObject

class pymeasure.display.widgets.ImageFrame(x_axis, y_axis, z_axis=None, refresh_time=0.2, check_status=True, parent=None)
    Bases: sphinx.ext.autodoc.importer._MockObject
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

**parse_axis**(axis)

Returns the units of an axis by searching the string

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

Bases: sphinx.ext.autodoc.importer._MockObject

Extends the PlotFrame to allow different columns of the data to be dynamically choosen

```python
new_image(results)
```

Creates a new image

```python
class pymeasure.display.widgets.InputsWidget (procedure_class, inputs=(), parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

```python
get_procedure()
```

Returns the current procedure

```python
class pymeasure.display.widgets.LogWidget (parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

```python
class pymeasure.display.widgets.PlotFrame (x_axis=None, y_axis=None, refresh_time=0.2, check_status=True, parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

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

```python
parse_axis(axis)
```

Returns the units of an axis by searching the string

```python
class pymeasure.display.widgets.PlotWidget (columns, x_axis=None, y_axis=None, refresh_time=0.2, check_status=True, parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

Extends the PlotFrame to allow different columns of the data to be dynamically choosen

```python
class pymeasure.display.widgets.ResultsDialog (columns, x_axis=None, y_axis=None, parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

```python
exception pymeasure.display.widgets.SequenceEvaluationException
```

Bases: Exception

Raised when the evaluation of a sequence string goes wrong.

```python
class pymeasure.display.widgets.SequencerWidget (inputs=None, sequence_file=None, parent=None)
```

Bases: sphinx.ext.autodoc.importer._MockObject

Widget that allows to generate a sequence of measurements with varying parameters. Moreover, one can write a simple text file to easily load a sequence.

Currently requires a queue function of the ManagedWindow to have a “procedure” argument.

```python
static eval_string(string, name=None, depth=None)
```

Evaluate the given string. The string is evaluated using a list of pre-defined functions that are deemed safe to use, to prevent the execution of malicious code. For this purpose, also any built-in functions or global variables are not available.
Parameters

- **string** – String to be interpreted.
- **name** – Name of the to-be-interpreted string, only used for error messages.
- **depth** – Depth of the to-be-interpreted string, only used for error messages.

```python
load_sequence(*, fileName=None)
```

Load a sequence from a .txt file.

**Parameters**

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

```python
queue_sequence()
```

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

## 6.11 Windows classes

```python
class pymeasure.display.windows.ManagedImageWindow(procedure_class, x_axis, y_axis, z_axis=None, inputs=(), displays=(), log_channel=", log_level=20, parent=None)
```

**Bases:** sphinx.ext.autodoc.importer._MockObject

Abstract base class.

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

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

**See also:**

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

```python
plot
```

The `pyqtgraph.PlotItem` object for this window. Can be accessed to further customise the plot view programmatically, e.g., display log-log or semi-log axes by default, change axis range, etc.

```python
queue()
```

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.

For example:

```python
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.

setup_im_plot(im_plot)
This method does nothing by default, but can be overridden by the child class in order to set up custom options for the image plot
This method is called during the constructor, after all other set up has been completed, and is provided as a convenience method to parallel Plotter.

Parameters im_plot – This window’s ImageItem instance.

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
This method is called during the constructor, after all other set up has been completed, and is provided as a convenience method to parallel Plotter.

Parameters plot – This window’s PlotItem instance.

class pymeasure.display.windows.ManagedWindow(procedure_class, inputs=(), displays=(), x_axis=None, y_axis=None, log_channel='', log_level=20, parent=None, sequencer=False, sequencer_inputs=None, sequence_file=None, inputs_in_scrollarea=False)

Bases: sphinx.ext.autodoc.importer._MockObject
Abstract base class.

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

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

See also:

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

plot
The pyqtgraph.PlotItem object for this window. Can be accessed to further customise the plot view programmatically, e.g., display log-log or semi-log axes by default, change axis range, etc.

queue()
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.

For example:

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

(continues on next page)
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.

**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

This method is called during the constructor, after all other set up has been completed, and is provided as a convenience method to parallel Plotter.

**Parameters**

`plot` – This window’s PlotItem instance.

```python
class pymeasure.display.windows.PlotterWindow(plotter, refresh_time=0.1, parent=None)
Bases: sphinx.ext.autodoc.importer._MockObject

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
This section contains documentation on the instrument classes.

### 7.1 Instrument classes

```python
class pymeasure.instruments.Instrument(adapter, name, includeSCPI=True, **kwargs)
```

This provides the base class for all Instruments, which is independent of the particular Adapter used to connect for communication to the instrument. It provides basic SCPI commands by default, but can be toggled with includeSCPI.

**Parameters**

- **adapter** – An `Adapter` object
- **name** – A string name
- **includeSCPI** – A boolean, which toggles the inclusion of standard SCPI commands

```python
def ask(command)
    Writes the command to the instrument through the adapter and returns the read response.

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

```python
def check_errors()
    Return any accumulated errors. Must be reimplemented by subclasses.
```

```python
def clear()
    Clears the instrument status byte
```

```python
static control(get_command, set_command, docs, validator=<function Instrument.<lambda>>,
               values=(), map_values=False, get_process=<function Instrument.<lambda>>,
               set_process=<function Instrument.<lambda>>, check_set_errors=False,
               check_get_errors=False, **kwargs)
```

Returns a property for the class based on the supplied commands. This property may be set and read from the instrument.

**Parameters**
• **get_command** – A string command that asks for the value
• **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
• **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
• **check_set_errors** – Toggles checking errors after setting
• **check_get_errors** – Toggles checking errors after getting

**id**
Requests and returns the identification of the instrument.

**static measurement** (**get_command**, **docs**, **values**=(), **map_values**=None, **get_process**=<function Instrument.<lambda>>, **command_process**=<function Instrument.<lambda>>, **check_get_errors**=False, **kwargs**)
Returns 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 both getting and setting
• **check_set_errors** – Toggles checking errors after setting

**read**()
Reads from the instrument through the adapter and returns the response.

**reset**()
Resets the instrument.

**static setting** (**set_command**, **docs**, **validator**=<function Instrument.<lambda>>, **values**=(), **map_values**=False, **set_process**=<function Instrument.<lambda>>, **check_set_errors**=False, **kwargs**)
Returns 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

**shutdown()**
Brings the instrument to a safe and stable state

**values**(command, **kwargs)
Reads a set of values from the instrument through the adapter, passing on any key-word arguments.

**write**(command)
Writes the command to the instrument through the adapter.

Parameters command – command string to be sent to the instrument

**class** pymeasure.instruments.Mock(wait=0.1, **kwargs)
**Bases:** pymeasure.instruments.instrument.Instrument

Mock instrument for testing.

**get_time()**
Get elapsed time

**get_voltage()**
Get the voltage.

**get_wave()**
Get wave.

**reset_time()**
Reset the timer to 0 s.

**set_output_voltage**(value)
Set the voltage.

**set_time**(value)
Wait for the timer to reach the specified time. If value = 0, reset.

**time**
Get elapsed time

**voltage**
Get the voltage.

**wave**
Get wave.
7.2 Validator functions

Validators are used in conjunction with the :func:`Instrument.control` function to allow properties with complex restrictions for valid values. They are described in more detail in the Advanced properties 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)

Join a list of validators together as a single. Expects a list of validator functions and values.

**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 the maximum and greater than the minimum of the range. 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

```python
def 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

    ```

```python
def 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.

```python
getAI = pyMeasure.instruments.comedi.getAI
getAO = pyMeasure.instruments.comedi.getAO
readAI = pyMeasure.instruments.comedi.readAI
writeAO = pyMeasure.instruments.comedi.writeAO
```

## 7.4 Resource Manager

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

```python
list_resources = pyMeasure.instruments.list_resources()
```

```
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 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.

7.5.1 Advantest R3767CG Vector Network Analyzer

class pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG(resourceName, **kwargs)

Bases: pymeasure.instruments.instrument.Instrument

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

center_frequency  
Center Frequency in Hz

id  
Reads the instrument identification

span_frequency  
Span Frequency in Hz

start_frequency  
Starting frequency in Hz

stop_frequency  
Stoping frequency in Hz

trace_1  
Reads the Data array from trace 1 after formatting

7.6 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.

7.6.1 Agilent 8257D Signal Generator

class pymeasure.instruments.agilent.Agilent8257D(adapter, **kwargs)

Bases: pymeasure.instruments.instrument.Instrument

Represents the Agilent 8257D Signal Generator and provides a high-level interface for interacting with the instrument.

```
generator = Agilent8257D("GPIB::1")
generator.power = 0  # Sets the output power to 0 dBm
generator.frequency = 5  # Sets the output frequency to 5 GHz
generator.enable()  # Enables the output
```

amplitude_depth  
A floating point property that controls the amplitude modulation in percent, which can take values from 0 to 100 %.
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’.

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.

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.

frequency

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

has_amplitude_modulation

   Reads a boolean value that is True if the amplitude modulation is enabled.

has_modulation

   Reads a boolean value that is True if the modulation is enabled.

has_pulse_modulation

   Reads a boolean value that is True if the pulse modulation is enabled.

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.

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’.

is_enabled

   Reads a boolean value that is True if the output is on.

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

low_freq_out_source

   A string property which controls the source of the low frequency output, which can take the values ‘internal’ [2]’ for the inernal source, or ‘function [2]’ for an internal function generator which can be configured.

power

   A floating point property that represents the output power in dBm. This property can be set.

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.

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’.

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.

start_frequency

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

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.

step_points

   An integer number of points in a step sweep. This property can be set.
**stop_frequency**
A floating point property that represents the stop frequency in Hz. This property can be set.

**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.

### 7.6.2 Agilent 8722ES Vector Network Analyzer

```python
class pymeasure.instruments.agilent.Agilent8722ES(resourceName, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument
```

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

**data**
Returns the real and imaginary data from the last scan

**disable_averaging()**
Disables averaging

**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=1, blocking=True, timeout=25, delay=0.1)**
Initiates a scan with the number of averages specified and blocks until the operation is complete if blocking is True

**scan_continuous()**
Initiates a continuous scan

**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)**
Turns on averaging of a specific number between 0 and 999

**set_fixed_frequency(frequency)**
Sets the scan to be of only one frequency in Hz

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

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

7.6.3 Agilent E4408B Spectrum Analyzer

class pymeasure.instruments.agilent.AgilentE4408B(resourceName, **kwargs)

Bases: pymeasure.instruments.instrument.Instrument

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

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

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

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

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

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

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

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

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

def 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).

7.6.4 Agilent E4980 LCR Meter

class pymeasure.instruments.agilent.AgilentE4980(adapter, **kwargs)

Bases: pymeasure.instruments.instrument.Instrument

Represents LCR meter E4980A/AL

ac_current
AC current level, in Amps

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**

**frequency**

AC frequency (range depending on model), in Hertz

**impedance**

Measured data A and B, according to **mode**

**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: Resitance [Ohm] and reactance [Ohm]

• ZTD: Impedance, magnitude [Ohm] and phase [deg]

• ZTR: Impedance, magnitude [Ohm] and phase [rad]

• GB: Conductance [S] and susceptance [S]

• YTD: Admittance, magnitude [Ohm] and phase [deg]

• YTR: Admittance magnitude [Ohm] and phase [rad]

**trigger_source**

Select trigger source; accept the values:
• HOLD: manual
• INT: internal
• BUS: external bus (GPIB/LAN/USB)
• EXT: external connector

7.6.5 Agilent 34410A Multimeter

class pymeasure.instruments.agilent.Agilent34410A(adapter, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument
    Represent the HP/Agilent/Keysight 34410A and related multimeters.
    Implemented measurements: voltage_dc, voltage_ac, current_dc, current_ac, resistance, resistance_4w

    current_ac
        AC current, in Amps

    current_dc
        DC current, in Amps

    resistance
        Resistance, in Ohms

    resistance_4w
        Four-wires (remote sensing) resistance, in Ohms

    voltage_ac
        AC voltage, in Volts

    voltage_dc
        DC voltage, in Volts

7.6.6 Agilent 4155/4156 Semiconductor Parameter Analyzer

class pymeasure.instruments.agilent.agilent4156.Agilent4156(resourceName, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument
    Represents the Agilent 4155/4156 Semiconductor Parameter Analyzer and provides a high-level interface for
taking current-voltage (I-V) measurements.

    from pymeasure.instruments.agilent import Agilent4156
    # explicitly define r/w terminations; set sufficiently large timeout or None.
    smu = Agilent4156("GPIB0::25", read_termination = '\n', write_termination = '\n',
                      timeout=None)

    # reset the instrument
    smu.reset()

    # define configuration file for instrument and load config
    smu.configure("configuration_file.json")

    # save data variables, some or all of which are defined in the json config file.
    smu.save(['VC', 'IC', 'VB', 'IB'])
# take measurements
status = smu.measure()

# measured data is a pandas dataframe and can be exported to csv.
data = smu.get_data(path='./t1.csv')

The JSON file is an ascii text configuration file that defines the settings of each channel on the instrument. The JSON file is used to configure the instrument using the convenience function `configure()` as shown in the example above. For example, the instrument setup for a bipolar transistor measurement is shown below.

```json
{
    "SMU1": {
        "voltage_name": "VC",
        "current_name": "IC",
        "channel_function": "VAR1",
        "channel_mode": "V",
        "series_resistance": "0OHM"
    },
    "SMU2": {
        "voltage_name": "VB",
        "current_name": "IB",
        "channel_function": "VAR2",
        "channel_mode": "I",
        "series_resistance": "0OHM"
    },
    "SMU3": {
        "voltage_name": "VE",
        "current_name": "IE",
        "channel_function": "CONS",
        "channel_mode": "V",
        "constant_value": 0,
        "compliance": 0.1
    },
    "SMU4": {
        "voltage_name": "VS",
        "current_name": "IS",
        "channel_function": "CONS",
        "channel_mode": "V",
        "constant_value": 0,
        "compliance": 0.1
    },
    "VAR1": {
        "start": 1,
        "stop": 2,
        "step": 0.1,
        "spacing": "LINEAR",
        "compliance": 0.1
    },
    "VAR2": {
        "start": 0,
        "step": 10e-6,
```
```json
{
  "points" : 3,
  "compliance" : 2
}
```

**analyzer_mode**
A string property that controls the instrument operating mode.

- Values: SWEEP, SAMPLING

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

**configure (config_file)**
Convenience function to configure the channel setup and sweep using a JSON (JavaScript Object Notation) configuration file.

**Parameters**
- **config_file** – JSON file to configure instrument channels.

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

**data_variables**
Gets 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**

```python
header = instr.data_variables
```

**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.

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

**disable_all ()**
Disables all channels in the instrument.

```python
instr.disable_all()
```

**get_data (path=None)**
Gets 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**

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

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

- Values: SHORT, MEDIUM, LONG

```python
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.

```python
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.

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

**stop()**
Stops the ongoing measurement

```python
instr.stop()
```

**class**
pymeasure.instruments.agilent.agilent4156.SMU(resourceName, channel, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

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

- Values: VAR1, VAR2, VARD or CONS.

```python
instr.smu1.channel_function = "VAR1"
```

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

- Values: V, I or COMM
VPULSE AND IPULSE are not yet supported.

```python
instr.smu1.channel_mode = "V"
```

**compliance**

This command 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().

```python
instr.smu1.compliance = 0.1
```

**constant_value**

This command sets 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.

```python
instr.smu1.constant_value = 1
```

**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.

```python
instr.smu1.current_name = "Ibase"
```

**disable**

This command deletes the settings of SMU<n>.

```python
instr.smu1.disable()
```

**series_resistance**

This command controls the series resistance of SMU<n>.

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

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

**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.

```python
instr.smu1.voltage_name = "Vbase"
```

### class `pymeasure.instruments.agilent.agilent4156.VAR1`(*resourceName*, **kwargs)

**Bases:** `pymeasure.instruments.agilent.agilent4156.VARX`

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

**spacing**

This command selects the sweep type of VAR1.
• Values: LINEAR, LOG10, LOG25, LOG50.

class pymeasure.instruments.agilent.agilent4156.VAR2(resourceName, **kwargs)
    Bases: pymeasure.instruments.agilent.agilent4156.VARX

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

    points
    This command 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(resourceName, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument

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

    compliance
    This command sets the sweep COMPLIANCE value of VARD.

        instr.vard.compliance = 0.1

    offset
    This command sets the OFFSET value of VARD. For each step of sweep, the output values of VAR1’ are determined by the following equation: VARD = VAR1 X RATio + OFFSet You use this command only if there is an SMU or VSU whose function is VARD.

        instr.vard.offset = 1

    ratio
    This command sets the RATIO of VAR1’. For each step of sweep, the output values of VAR1’ are determined by the following equation: VAR1’ = VAR1 X 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(resourceName, var_name, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument

    Base class to define sweep variable settings

    compliance
    Sets the sweep COMPLIANCE value.

        instr.var1.compliance = 0.1

    start
    Sets the sweep START value.

        instr.var1.start = 0

    step
    Sets the sweep STEP value.

        instr.var1.step = 0.1

    stop
    Sets the sweep STOP value.

7.6. Agilent
class pymeasure.instruments.agilent.agilent4156.VMU(resourceName, channel, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

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

• Values: V, D, VOL

disable
This command disables the settings of VMU<n>.

    instr.vmul.disable()

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.vmul.voltage_name = "Vanode"

class pymeasure.instruments.agilent.agilent4156.VSU(resourceName, channel, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

channel_function
A string property that controls the VSU channel function.

• Value: VAR1, VAR2, VARD or CONS.

channel_mode
Get channel mode of VSU<n>.

constant_value
This command sets the constant source value of VSU<n>.

    instr.vsul.constant_value = 0

disable
This command deletes the settings of VSU<n>.

    instr.vsul.disable()

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.vsul.voltage_name = "Ve"

7.6.7 Agilent 33220A Arbitrary Waveform Generator

class pymeasure.instruments.agilent.Agilent33220A(adapter, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument
Represents the Agilent 33220A Arbitrary Waveform Generator.

```python
# 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 = True        # Enable burst mode
wfg.burst_ncycles = 10e3  # 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"

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

**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.

**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.

**beeper_state**

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

**burst_mode**

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

**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.

**burst_state**

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

**check_errors()**

Read all errors from the instrument.

**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.

**offset**

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

**output**

A boolean property that turns on (True) or off (False) the output of the function generator. Can be set.
pulse_dutycycle
A floating point property that controls the duty cycle of a pulse waveform function in percent. Can be set.

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>.

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.

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. Can be set.

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.

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

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.

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

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.

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 only be set.

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

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.

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.

### 7.6.8 Agilent 33500 Function/Arbitrary Waveform Generator Family

**class** `pymeasure.instruments.agilent.Agilent33500(adapter, **kwargs)`

**Bases:** `pymeasure.instruments.instrument.Instrument`

Represents the Agilent 33500 Function/Arbitrary Waveform Generator family. Individual devices are represented by subclasses.

```python
generator = Agilent33500("GPIB::1")

generator.shape = 'SIN'  # Sets the output signal shape to sine
generator.frequency = 1e3  # Sets the output frequency to 1 kHz
generator.amplitude = 5  # Sets the output amplitude to 5 Vpp
generator.output = 'on'  # Enables the output

generator.shape = 'ARB'  # Set shape to arbitrary

generator.arb_srate = 1e6  # Set sample rate to 1MSa/s

generator.data_volatile_clear()  # Clear volatile internal memory

generator.data_arb('test',
range(-10000, 10000, +20),
data_format='DAC')  # In this case a simple ramp

generator.arb_file = 'test'  # Select the transmitted waveform 'test'
```

**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. Can be set.

**amplitude_unit**

A string property that controls the units of the amplitude. Valid values are VPP (default), VRMS, and DBM. Can be set.

**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).

**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 memore (set by data_arb()). Can be set.

**arb_filter**

A string property that selects the filter setting for arbitrary signals. Can be set to ‘NORM<AL>’, ‘STEP’ and ‘OFF’.

**arb_srate**

An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1 µSa/s to 250 MSa/s (maximum range, can be lower depending on your device). This can be set.

**beep()**

Causes a system beep.

**burst_mode**

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

burst_period
A floating point property that controls the period of subsequent bursts. Has to follow the equation burst_period > (burst_ncycles / frequency) + 1 µs. Valid values are 1 µs to 8000 s. Can be set.

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

check_errors()
Read all errors from the instrument.

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). TODO: Binary is not yet implemented

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 the 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.

display
A string property which is displayed on the front panel of the device. Can be set.

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. Can be set.

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. Can be set.

id
Reads the instrument identification
**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_{max} - \text{voltage}) / 2). Can be set.

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

**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. Can be set.

**pulse_dutyCycle**
A floating point property that controls the duty cycle of a pulse waveform function in percent, from 0% to 100%. Can be set.

**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>.

**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.

**pulse_transition**
A floating point property that controls the edge time in seconds for both the rising and falling edges. It is defined as the time between the 10% and 90% thresholds of the edge. Valid values are between 8.4 ns to 1 µs. Can be set.

**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. Can be set.

**ramp_symmetry**
A floating point property that controls the symmetry percentage for the ramp waveform, from 0.0% to 100.0%. Can be set.

**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.

**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. Can be set.

**trigger()**
Send a trigger signal to the function generator.

**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.

**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 by at least 1 mV). Can be set.
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 by at least 1 mV). Can be set.

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.

7.6.9 Agilent 33521A Function/Arbitrary Waveform Generator

class pymeasure.instruments.agilent.Agilent33521A (adapter, **kwargs)
Bases: pymeasure.instruments.agilent.Agilent33500.Agilent33500

Represents the Agilent 33521A Function/Arbitrary Waveform Generator. This documentation page shows only methods different from the parent class Agilent33500.

arb_srate
An floating point property that sets the sample rate of the currently selected arbitrary signal. Valid values are 1 µSa/s to 250 MSa/s. This can be set.

frequency
A floating point property that controls the frequency of the output waveform in Hz, from 1 uHz to 30 MHz, depending on the specified function. Can be set.

7.7 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.

7.7.1 Ametek 7270 DSP Lockin Amplifier

class pymeasure.instruments.ametek.Ametek7270(resourceName, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

This is the class for the Ametek DSP 7270 lockin amplifier

adc1
Reads the input value of ADC1 in Volts

adc2
Reads the input value of ADC2 in Volts

adc3
Reads the input value of ADC3 in Volts

adc4
Reads the input value of ADC4 in Volts

dac1
A floating point property that represents the output value on DAC1 in Volts. This property can be set.
dac2
A floating point property that represents the output value on DAC2 in Volts. This property can be set.

dac3
A floating point property that represents the output value on DAC3 in Volts. This property can be set.

dac4
A floating point property that represents the output value on DAC4 in Volts. This property can be set.

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

harmonic
An integer property that represents the reference harmonic mode control, taking values from 1 to 127. This property can be set.

id
Reads the instrument identification

mag
Reads the magnitude in Volts

phase
A floating point property that represents the reference harmonic phase in degrees. This property can be set.

sensitivity
A floating point property that controls the sensitivity range in Volts, which can take discrete values from 2 nV to 1 V. This property can be set.

set_channel_A_mode()
Sets instrument to channel A mode – assuming it is in voltage mode

set_differential_mode(lineFiltering=True)
Sets instrument to differential mode – assuming it is in voltage mode

set_voltage_mode()
Sets instrument to voltage control mode

shutdown()
Ensures the instrument in a safe state

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.

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.

voltage
A floating point property that represents the voltage in Volts. This property can be set.

x
Reads the X value in Volts

x1
Reads the first harmonic X value in Volts

x2
Reads the second harmonic X value in Volts
xy
    Reads both the X and Y values in Volts

y
    Reads the Y value in Volts

y1
    Reads the first harmonic Y value in Volts

y2
    Reads the second harmonic Y value in Volts

7.8 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.

7.8.1 AMI 430 Power Supply

class pymeasure.instruments.ami.AMI430(resourceName, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument

Represents the AMI 430 Power supply and provides a high-level for interacting with the instrument.

```python
magnet = AMI430("TCP/IP::web.address.com::7180::SOCKET")

magnet.coilconst = 1.182  # kGauss/A
magnet.voltage_limit = 2.2  # Sets the voltage limit in V
magnet.target_current = 10  # Sets the target current to 10 A
magnet.target_field = 1  # Sets target field to 1 kGauss
magnet.ramp_rate_current = 0.0357  # Sets the ramp rate in A/s
magnet.ramp_rate_field = 0.0422  # Sets the ramp rate in kGauss/s
magnet.ramp  # Initiates the ramping
magnet.pause  # Pauses the ramping
magnet.status  # Returns the status of the magnet
magnet.ramp_to_current(5)  # Ramps the current to 5 A
magnet.shutdown()  # Ramps the current to zero and disables output
```

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.

field
    Reads the field in kGauss of the magnet.
has_persistent_switch_enabled()
Returns a boolean if the persistent switch is enabled.

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.

ramp_rate_current
A floating point property that sets the current ramping rate in A/s.

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.

state
Reads the field in kGauss of the magnet.

supply_current
Reads the current in Amps of the power supply.

target_current
A floating point property that sets the target current in A for the magnet.

target_field
A floating point property that sets the target field in kGauss for the magnet.

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.

7.9 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.

7.9.1 Anritsu MG3692C Signal Generator

class pymeasure.instruments.anritsu.AnritsuMG3692C (resourceName, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

Represents the Anritsu MG3692C Signal Generator
disable()  
Disables the signal output.

enable()  
Enables the signal output.

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

output  
A boolean property that represents the signal output state. This property can be set to control the output.

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.

7.9.2 Anritsu MS9710C Optical Spectrum Analyzer

class pymeasure.instruments.anritsu.AnritsuMS9710C(adapter, **kwargs)  
Bases: pymeasure.instruments.instrument.Instrument

Anritsu MS9710C Optical Spectrum Analyzer.

analysis  
Analysis Control

analysis_result  
Read back analysis result from current scan.

average_point  
Number of averages to take on each point (2-1000), or OFF

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.

data_memory_a_condition  
Returns the data condition of data memory register A. Starting wavelength, and a sampling point (11, 12, n).

data_memory_a_size  
Returns the number of points sampled in data memory register A.

data_memory_a_values  
Reads the binary data from memory register A.

data_memory_b_condition  
Returns the data condition of data memory register B. Starting wavelength, and a sampling point (11, 12, n).

data_memory_b_size  
Returns the number of points sampled in data memory register B.

data_memory_b_values  
Reads the binary data from memory register B.

data_memory_select  
Memory Data Select.
**dip_search**
Dip Search Mode

**ese2**
Extended Event Status Enable Register 2

**esr2**
Extended Event Status Register 2

**level_lin**
Level Linear Scale (/div)

**level_log**
Level Log Scale (/div)

**level_opt_attn**
Optical Attenuation Status (ON/OFF)

**level_scale**
Current Level Scale

**measure_mode**
Returns the current Measure Mode the OSA is in.

**measure_peak()**
Measure the peak and return the trace marker.

**peak_search**
Peak Search Mode

**read_memory(slot='A')**
Read the scan saved in a memory slot.

**resolution**
Resolution (nm)

**resolution_actual**
Resolution Actual (ON/OFF)

**resolution_vbw**
Video Bandwidth Resolution

**sampling_points**
Number of sampling points

**single_sweep(**kwargs**)**
Perform a single sweep and wait for completion.

**trace_marker**
Sets the trace marker with a wavelength. Returns the trace wavelength and power.

**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.

**wavelength_center**
Center Wavelength of Spectrum Scan in nm.
wavelength_marker_value
  Wavelength Marker Value (wavelength or freq.?)

wavelength_span
  Wavelength Span of Spectrum Scan in nm.

wavelength_start
  Wavelength Start of Spectrum Scan in nm.

wavelength_stop
  Wavelength Stop of Spectrum Scan in nm.

wavelength_value_in
  Wavelength value in Vacuum or Air

wavelengths
  Return a numpy array of the current wavelengths of scans.

7.10 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.

7.10.1 Danfysik Serial Adapter

class pymeasure.instruments.danfysik.DanfysikAdapter(port)
  Bases: pymeasure.adapters.serial.SerialAdapter

  Provides a SerialAdapter with the specific baudrate and timeout for Danfysik serial communication.

  Initiates the adapter to open serial communication over the supplied port.

  Parameters

  port – A string representing the serial port

  read()
    Overwrites the SerialAdapter.read method to automatically 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

  write(command)
    Overwrites the SerialAdapter.write method to automatically append a Unix-style linebreak at the end of the command.

    Parameters

    command – SCPI command string to be sent to the instrument

7.10.2 Danfysik 8500 Power Supply

class pymeasure.instruments.danfysik.Danfysik8500(port)
  Bases: pymeasure.instruments.instrument.Instrument

  Represents the Danfysik 8500 Electromanget Current Supply and provides a high-level interface for interacting with the instrument.

  To allow user access to the Prolific Technology PL2303 Serial port adapter in Linux, create the file: /etc/udev/rules.d/50-danfysik.rules, with contents:
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

**current**

The actual current in Amps. This property can be set through **current_ppm**.

**current_ppm**

The current in parts per million. This property can be set.

**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.

**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.

**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.

remote()
    Sets the instrument in remote mode, where 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)

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

status
    A list of human-readable strings that contain the instrument status information, based on status_hex.

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

## 7.11 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.

### 7.11.1 Delta Elektronika SM7045D Power source

**class** `pymeasure.instruments.deltaelektronika.SM7045D(resourceName, **kwargs)`

**Bases:** `pymeasure.instruments.instrument.Instrument`

This is the class for the SM 70-45 D power supply.

```python
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
```

**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.

**max_current**

A floating point property that represents the maximum output current of the power supply in Amps. This property can be set.

**max_voltage**

A floating point property that represents the maximum output voltage of the power supply in Volts. This property can be set.

**measure_current**

Measures the actual output current of the power supply in Amps.
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)

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.

voltage
A floating point property that represents the output voltage setting of the power supply in Volts. This property can be set.

7.12 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.

7.12.1 F.W. Bell 5080 Handheld Gaussmeter

class pymeasure.instruments.fwbell.FWBell5080 (port)
Bases: pymeasure.instruments.instrument.Instrument

Represents the F.W. Bell 5080 Handheld Gaussmeter and provides a high-level interface for interacting with the instrument

Parameters 

- port – The serial port of the instrument

```
meter = FWBell5080('/dev/ttyUSB0') # Connects over serial port /dev/ttyUSB0

meter.units = 'gauss' # Sets the measurement units to Gauss
meter.range = 3e3 # 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

ask(command) # Overwrites the Instrument.ask method to remove the last 2 characters from the output.
```
auto_range()

Enables the auto range functionality.

field

Reads a floating point value of the field in the appropriate units.

fields (samples=1)

Returns a numpy array of field samples for a given sample number.

Parameters samples – The number of samples to perform

id

Reads the identification information.

range

A floating point property that controls the maximum field range in the active units. This can take the values of 300 G, 3 kG, and 30 kG for Gauss, 30 mT, 300 mT, and 3 T for Tesla, and 23.88 kAm, 238.8 kAm, and 2388 kAm for Amp-meter.

read()

Overwrites the Instrument.read method to remove the last 2 characters from the output.

reset()

Resets the instrument.

units

A string property that controls the field units, which can take the values: ‘gauss’, ‘gauss ac’, ‘tesla’, ‘tesla ac’, ‘amp-meter’, and ‘amp-meter ac’. The AC versions configure the instrument to measure AC.

values (command)

Overwrites the Instrument.values method to remove the lastv2 characters from the output.

7.13 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.

7.13.1 HP 33120A Arbitrary Waveform Generator

class pymeasure.instruments.hp.HP33120A (resourceName, **kwargs)

Bases: pymeasure.instruments.instrument.Instrument

Represents the Hewlett Packard 33120A Arbitrary Waveform Generator and provides a high-level interface for interacting with the instrument.

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.

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.
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.

max_amplitude
   Reads the maximum amplitude in Volts for the given shape

max_frequency
   Reads the maximum frequency in Hz for the given shape

max_offset
   Reads the maximum offset in Volts for the given shape

min_amplitude
   Reads the minimum amplitude in Volts for the given shape

min_frequency
   Reads the minimum frequency in Hz for the given shape

min_offset
   Reads the minimum offset in Volts for the given shape

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.

shape
   A string property that controls the shape of the wave, which can take the values: sinusoid, square, triangle, ramp, noise, dc, and user.

7.13.2 HP 34401A Multimeter

class pymeasure.instruments.hp.HP34401A(resourceName, **kwargs)
   Bases: pymeasure.instruments.instrument.Instrument

   Represents the HP 34401A instrument.

   current_ac
      AC current, in Amps

   current_dc
      DC current, in Amps

   resistance
      Resistance, in Ohms

   resistance_4w
      Four-wires (remote sensing) resistance, in Ohms

   voltage_ac
      AC voltage, in Volts

   voltage_dc
      DC voltage, in Volts

7.14 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.
7.14.1 Keithley 2000 Multimeter

```python
class Keithley2000(
    adapter, **kwargs
)
```


Represents the Keithley 2000 Multimeter and provides a high-level interface for interacting with the instrument.

```python
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

### beep_state
A string property that enables or disables the system status beeper, which can take the values: `enabled` and `disabled`.

### buffer_data
Returns a numpy array of values from the buffer.

### 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.

### 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.

### current
Reads a DC or AC current measurement in Amps, based on the active `mode`.

### 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.

### 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.
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.

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.

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.

current_digits
An integer property that controls the number of digits in the DC current readings, which can take values from 4 to 7.

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.

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.

current_reference
A floating point property that controls the DC current reference value in Amps, which can take values from -3.1 to 3.1 A.

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process.

disable_filter(mode=None)
Disables the averaging filter for the active mode, or can set another mode by its name.

Parameters
mode – A valid mode name, or None for the active mode

disable_reference(mode=None)
Disables the reference for the active mode, or can set another mode by its name.

Parameters
mode – A valid mode name, or None for the active mode

enable_filter(mode=None, type='repeat', count=1)
Enables the averaging filter for the active mode, or can set another mode by its name.

Parameters
• mode – A valid mode name, or None for the active mode
• type – The type of averaging filter, either ‘repeat’ or ‘moving’.
• count – A number of averages, which can take take values from 1 to 100

enable_reference(mode=None)
Enables the reference for the active mode, or can set another mode by its name.

Parameters
mode – A valid mode name, or None for the active mode

frequency
Reads a frequency measurement in Hz, based on the active mode.
**frequency_aperture**
A floating point property that controls the frequency aperture in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

**frequency_digits**
An integer property that controls the number of digits in the frequency readings, which can take values from 4 to 7.

**frequency_reference**
A floating point property that controls the frequency reference value in Hz, which can take values from 0 to 15 MHz.

**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.

**id**
Requests and returns the identification of the instrument.

**is_buffer_full()**
Returns True if the buffer is full of measurements.

**local()**
Returns control to the instrument panel, and enables the panel if disabled.

**measure_continuity()**
Configures the instrument to perform continuity testing.

**measure_current (max_current=0.01, ac=False)**
Configures the instrument to measure current, based on a maximum current to set the range, and a boolean flag to determine if DC or AC is required.

**Parameters**
- **max_current** – A current in Volts to set the current range
- **ac** – False for DC current, and True for AC current

**measure_diode()**
Configures the instrument to perform diode testing.

**measure_frequency()**
Configures the instrument to measure the frequency.

**measure_period()**
Configures the instrument to measure the period.

**measure_resistance (max_resistance=1000000.0, 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

**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`, `frequency`, `temperature`, `diode`, and `frequency`.

**period**
Reads a period measurement in seconds, based on the active `mode`.

**period_aperature**
A floating point property that controls the period aperture in seconds, which sets the integration period and measurement speed. Takes values from 0.01 to 1.0 s.

**period_digits**
An integer property that controls the number of digits in the period readings, which can take values from 4 to 7.

**period_reference**
A floating point property that controls the period reference value in seconds, which can take values from 0 to 1 s.

**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.

**remote()**
Places the instrument in the remote state, which is does not need to be explicity called in general.

**remote_lock()**
Disables and locks the front panel controls to prevent changes during remote operations. This is disabled by calling `local()`.

**reset()**
Resets the instrument state.

**reset_buffer()**
Resets the buffer.

**resistance**
Reads a resistance measurement in Ohms for both 2-wire and 4-wire configurations, based on the active `mode`.

**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.

**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.

**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.
**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.

**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.

**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.

**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.

**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.

**stop_buffer()**
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

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

**temperature_digits**
An integer property that controls the number of digits in the temperature readings, which can take values from 4 to 7.

**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.

**temperature_reference**
A floating point property that controls the temperature reference value in Celsius, which can take values from -200 to 1372 C.

**trigger_count**
An integer property that controls the trigger count, which can take values from 1 to 9,999.

**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.

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

**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.
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.

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.

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.

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.

voltage_digits
An integer property that controls the number of digits in the DC voltage readings, which can take values from 4 to 7.

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.

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.

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_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

7.14.2 Keithley 2400 SourceMeter

class pymeasure.instruments.keithley.Keithley2400 (adapter, **kwargs)

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
(continues on next page)
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_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.

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

buffer_data
Returns a numpy array of values from the buffer.

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.
**compliance_current**
A floating point property that controls the compliance current in Amps.

**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.

**current**
Reads the current in Amps, if configured for this reading.

**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.

**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.

**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.

**error**
Returns a tuple of an error code and message from a single error.

**id**
Requests and returns the identification of the instrument.

**is_buffer_full()**
Returns True if the buffer is full of measurements.

**max_current**
Returns the maximum current from the buffer

**max_resistance**
Returns the maximum resistance from the buffer

**max_voltage**
Returns the maximum voltage from the buffer

**maximums**
Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list.
mean_current
Returns the mean current from the buffer

mean_resistance
Returns the mean resistance from the buffer

mean_voltage
Returns the mean voltage from the buffer

means
Reads the calculated means (averages) for voltage, current, and resistance from the buffer data as a list.

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

min_current
Returns the minimum current from the buffer

min_resistance
Returns the minimum resistance from the buffer

min_voltage
Returns the minimum voltage from the buffer

minimums
Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

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

**reset** ()
Resets the instrument and clears the queue.

**reset_buffer** ()
Resets the buffer.

**resistance**
Reads the resistance in Ohms, if configured for this reading.

**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.

**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.

**source_current**
A floating point property that controls the source current in Amps.
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.

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].

source_delay_auto
A boolean property that enables or disables auto delay. Valid values are True and False.

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.

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.

source_voltage
A floating point property that controls the source voltage in Volts.

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.

standard_devs
Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

start_buffer()
Starts the buffer.

std_current
Returns the current standard deviation from the buffer

std_resistance
Returns the resistance standard deviation from the buffer

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.

trigger_count
An integer property that controls the trigger count, which can take values from 1 to 9,999.
**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 GET or *TRG.

**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.

**voltage**
Reads the voltage in Volts, if configured for this reading.

**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.

**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_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

**wires**
An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

### 7.14.3 Keithley 2450 SourceMeter

**class pymeasure.instruments.keithley.Keithley2450 (adapter, **kwargs)**

Represents the Keithely 2450 SourceMeter and provides a high-level interface for interacting with the instrument.

```python
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.measureVoltage()          # Sets up to measure voltage
keithley.rampToCurrent(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

buffer_data
Returns a numpy array of values from the buffer.

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.

compliance_current
A floating point property that controls the compliance current in Amps.

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.

**current**
Reads the current in Amps, if configured for this reading.

**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.

**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.

**error**
Returns a tuple of an error code and message from a single error.

**id**
Requests and returns the identification of the instrument.

**is_buffer_full()**
Returns True if the buffer is full of measurements.

**max_current**
Returns the maximum current from the buffer

**max_resistance**
Returns the maximum resistance from the buffer

**max_voltage**
Returns the maximum voltage from the buffer

**maximums**
Reads the calculated maximums for voltage, current, and resistance from the buffer data as a list.

**mean_current**
Returns the mean current from the buffer

**mean_resistance**
Returns the mean resistance from the buffer

**mean_voltage**
Returns the mean voltage from the buffer

**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

- \texttt{nplc} – Number of power line cycles (NPLC) from 0.01 to 10
- \texttt{current} – Upper limit of current in Amps, from -1.05 A to 1.05 A
- \texttt{auto\_range} – Enables auto\_range if True, else uses the set current

\texttt{measure\_resistance} (nplc=1, resistance=210000.0, auto\_range=True)

Configures the measurement of resistance.

Parameters

- \texttt{nplc} – Number of power line cycles (NPLC) from 0.01 to 10
- \texttt{resistance} – Upper limit of resistance in Ohms, from -210 MOhms to 210 MOhms
- \texttt{auto\_range} – Enables auto\_range if True, else uses the set resistance

\texttt{measure\_voltage} (nplc=1, voltage=21.0, auto\_range=True)

Configures the measurement of voltage.

Parameters

- \texttt{nplc} – Number of power line cycles (NPLC) from 0.01 to 10
- \texttt{voltage} – Upper limit of voltage in Volts, from -210 V to 210 V
- \texttt{auto\_range} – Enables auto\_range if True, else uses the set voltage

\texttt{min\_current}

Returns the minimum current from the buffer

\texttt{min\_resistance}

Returns the minimum resistance from the buffer

\texttt{min\_voltage}

Returns the minimum voltage from the buffer

\texttt{minimums}

Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list.

\texttt{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

- \texttt{target\_current} – A current in Amps
- \texttt{steps} – An integer number of steps
- \texttt{pause} – A pause duration in seconds to wait between steps

\texttt{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

- \texttt{target\_voltage} – A voltage in Amps
- \texttt{steps} – An integer number of steps
- \texttt{pause} – A pause duration in seconds to wait between steps

\texttt{reset} ()

Resets the instrument and clears the queue.
reset_buffer()  
Resets the buffer.

resistance  
Reads the resistance in Ohms, if configured for this reading.

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.

resistance_range  
A floating point property that controls the resistance range in Ohms, which can take values from 0 to 210 MΩ. Auto-range is disabled when this property is set.

shutdown()  
Ensures that the current or voltage is turned to zero and disables the output.

source_current  
A floating point property that controls the source current in Amps.

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.

source_enabled  
Reads a boolean value that is True if the source is enabled.

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.

source_voltage  
A floating point property that controls the source voltage in Volts.

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.

standard_devs  
Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list.

start_buffer()  
Starts the buffer.

std_current  
Returns the current standard deviation from the buffer

std_resistance  
Returns the resistance standard deviation from the buffer

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.

**voltage**

Reads the voltage in Volts, if configured for this reading.

**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.

**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_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

**wires**

An integer property that controls the number of wires in use for resistance measurements, which can take the value of 2 or 4.

### 7.14.4 Keithley 2700 MultiMeter/Switch System

class pymeasure.instruments.keithley.Keithley2700(adapter, **kwargs)

Represent the Keithley 2700 Multimeter/Switch System and provides a high-level interface for interacting with the instrument.

```python
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
buffer_data
Returns a numpy array of values from the buffer.

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

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

closed_channels
Parameter that controls the opened and closed channels. All mentioned channels are closed, other channels
will be opened.

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 pro-
cess.

display_closed_channels()
Show the presently closed channels on the display of the Keithley 2700.

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.

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

**id**
Requests and returns the identification of the instrument.

**is_buffer_full**()
Returns True if the buffer is full of measurements.

**open_all_channels**()
Open all channels of the Keithley 2700.

**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

**options**
Property that lists the installed cards in the Keithley 2700. Returns a dict with the integer card numbers on the position.

**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.

**stop_buffer**()
Aborts the buffering measurement, by stopping the measurement arming and triggering sequence. If possible, a Selected Device Clear (SDC) is used.

**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_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

### 7.14.5 Keithley 6221 AC and DC Current Source

class pymeasure.instruments.keithley.Keithley6221(adapter, **kwargs)

**Bases:** pymeasure.instruments.instrument.Instrument, pymeasure.instruments.keithley.buffer.KeithleyBuffer

Represents the Keithley 6221 AC and DC current source and provides a high-level interface for interacting with the instrument.

```python
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

**buffer_data**

Returns a numpy array of values from the buffer.

**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.

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.

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.

error
Returns a tuple of an error code and message from a single error.

id
Requests and returns the identification of the instrument.

is_buffer_full()
Returns True if the buffer is full of measurements.

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.

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.

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.

**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.

**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

**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.

**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.

**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.

**source_auto_range**
A boolean property that controls the auto range of the current source. Valid values are True or False.

**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].

**source_current**
A floating point property that controls the source current in Amps.

**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].

**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.
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.

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.

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.

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.

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 GET or *TRG.

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_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.
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.

waveform_duration_cycles
A floating point property that controls the duration of the waveform in cycles. Valid values are in range 1e-3 to 99999999900.

waveform_duration_set_infinity()
Set the waveform duration to infinity.

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.

waveform_duty_cycle
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.

waveform_frequency
A floating point property that controls the frequency of the waveform in Hertz. Valid values are in range 1e-3 to 1e5.

waveform_function
A string property that controls the selected wave function. Valid values are “sine”, “ramp”, “square”, “arbitrary1”, “arbitrary2”, “arbitrary3” and “arbitrary4”.

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.

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

waveform_use_phasemarker
A boolean property that controls whether the phase marker option is turned on or off. Valid values True (on) or False (off). Other settings for the phase marker have not yet been implemented.

7.14.6 Keithley 2750 Multimeter/Switch System

class pymeasure.instruments.keithley.Keithley2750 (adapter, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

Represents the Keithley2750 multimeter/switch system and provides a high-level interface for interacting with the instrument.

check_errors()
Return any accumulated errors. Must be reimplemented by subclasses.

close (channel)
Closes (connects) the specified channel.

Parameters channel (int) – 3-digit number for the channel

Returns None
closed_channels
    Reads the list of closed channels

id
    Requests and returns the identification of the instrument.

open(channel)
    Opens (disconnects) the specified channel.

    Parameters channel (int) – 3-digit number for the channel

    Returns None

open_all()
    Opens (disconnects) all the channels on the switch matrix.

    Returns None

reset()
    Resets the instrument.

shutdown()
    Brings the instrument to a safe and stable state

7.15 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.

7.15.1 Keysight N5767A Power Supply

class pymeasure.instruments.keysight.KeysightN5767A(adapter, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument

    Represents the Keysight N5767A Power supply interface for interacting with the instrument.

    check_errors()
        Read all errors from the instrument.

    current
        Reads a setting current in Amps.

    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.

    id
        Requests and returns the identification of the instrument.

    is_enabled()
        Returns True if the current supply is enabled.
reset()
    Resets the instrument.

shutdown()
    Brings the instrument to a safe and stable state

voltage
    Reads a DC voltage measurement in Volts.

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.

7.16 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.

7.16.1 Lake Shore Adapters

class pymeasure.instruments.lakeshore.LakeShoreUSBAdapter (port)
Bases: pymeasure.adapters.serial.SerialAdapter

Provides a SerialAdapter with the specific baudrate, timeout, parity, and byte size for LakeShore USB communication.

Initiates the adapter to open serial communication over the supplied port.

Parameters port – A string representing the serial port

ask (command)
    Writes the command to the instrument and returns the resulting ASCII response

    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

    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

read()
    Reads until the buffer is empty and returns the resulting ASCII response

    Returns String ASCII response of the instrument.

values (command, separator=’,’, cast=<class ‘float’>)
    Writes a command to the instrument and returns a list of formatted values from the result

    Parameters
        • command – SCPI command to be sent to the instrument
• **separator** – A separator character to split the string into a list
• **cast** – A type to cast the result

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

```python
def write(command):
    """Overwrites the SerialAdapter.write method to automatically append a Unix-style linebreak at the end of the command."

    Parameters
        command -- SCPI command string to be sent to the instrument
```

### 7.16.2 Lake Shore 331 Temperature Controller

```
class pymeasure.instruments.lakeshore.LakeShore331(adapter, **kwargs):
    """Bases: pymeasure.instruments.instrument.Instrument"

    Represents the Lake Shore 331 Temperature Controller and provides a high-level interface for interacting with the instrument.

    controller = LakeShore331("GPIB::1")
    print(controller.setpoint_1)  # Print the current setpoint for loop 1
    controller.setpoint_1 = 50  # Change the setpoint to 50 K
    controller.heater_range = 'low'  # Change the heater range to Low
    controller.wait_for_temperature()  # Wait for the temperature to stabilize
    print(controller.temperature_A)  # Print the temperature at sensor A
```

```python
def disable_heater():
    """Turns the heater_range to off to disable the heater."

def heater_range:
    """A string property that controls the heater range, which can take the values: off, low, medium, and high. These values correlate to 0, 0.5, 5 and 50 W respectively."

def setpoint_1:
    """A floating point property that controls the setpoint temperature in Kelvin for Loop 1."

def setpoint_2:
    """A floating point property that controls the setpoint temperature in Kelvin for Loop 2."

def temperature_A:
    """Reads the temperature of the sensor A in Kelvin."

def temperature_B:
    """Reads the temperature of the sensor B in Kelvin."

def wait_for_temperature(accuracy=0.1, interval=0.1, sensor='A', setpoint=1, timeout=360, should_stop=<function LakeShore331.<lambda>>):
    """Blocks the program, waiting for the temperature to reach the setpoint within the accuracy (%), checking this each interval time in seconds."

    Parameters
        accuracy -- An acceptable percentage deviation between the setpoint and temperature
        interval -- A time in seconds that controls the refresh rate
        sensor -- The desired sensor to read, either A or B
        setpoint -- The desired setpoint loop to read, either 1 or 2
        timeout -- A timeout in seconds after which an exception is raised"
```
• **should_stop** – A function that returns True if waiting should stop, by default this always returns False

### 7.16.3 Lake Shore 425 Gaussmeter

**class** `pymeasure.instruments.lakeshore.LakeShore425(port)`

**Bases:** `pymeasure.instruments.instrument.Instrument`

Represents the LakeShore 425 Gaussmeter and provides a high-level interface for interacting with the instrument.

To allow user access to the LakeShore 425 Gaussmeter in Linux, create the file: `/etc/udev/rules.d/52-lakeshore425.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="1fb9",ATTRS{idProduct}=="0401",MODE="0666",
→SYMLINK+="lakeshore425"
```

Then reload the udev rules with:

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

The device will be accessible through `/dev/lakeshore425`.

**ac_mode** *(wideband=True)*

Sets up a measurement of an oscillating (AC) field

**auto_range** ()

Sets the field range to automatically adjust

**dc_mode** *(wideband=True)*

Sets up a steady-state (DC) measurement of the field

**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

**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.

**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

### 7.17 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.*
7.17.1 ESP 300 Motion Controller

class pymeasure.instruments.newport.ESP300 (resourceName, **kwargs)
   Bases: pymeasure.instruments.instrument.Instrument

Represents the Newport ESP 300 Motion Controller and provides a high-level for interacting with the instru-
ment.

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.

axes
   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.

error
   Reads an error code from the motion controller.

errors
   Returns a list of error Exceptions that can be later raised, or used to diagnose the situation.

shutdown ()
   Shuts down the controller by disabling all of the axes.

class pymeasure.instruments.newport.esp300.Axis (axis, controller)
   Bases: object

Represents an axis of the Newport ESP300 Motor Controller, which can have independent parameters from the
other axes.

define_position (position)
   Overwrites the value of the current position with the given value.

disable ()
   Disables motion for the axis.

enable ()
   Enables motion for the axis.

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.

left_limit
   A floating point property that controls the left software limit of the axis.

motion_done
   Returns a boolean that is True if the motion is finished.
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.

right_limit
A floating point property that controls the right software limit of the axis.

units
A string property that controls the displacement units of the axis, which can take values of: enconder count, motor step, millimeter, micrometer, inches, milli-inches, micro-inches, degree, gradient, radian, milliradian, and microradian.

wait_for_stop (*delay=0, interval=0.05*)
Blocks the program until the motion is completed. A further delay can be specified in seconds.

zero()
Resets the axis position to be zero at the current 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.

7.18 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.

7.18.1 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.

```python
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)
```

(continues on next page)
Class `pymeasure.instruments.ni.virtualbench.VirtualBench`:

```python
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='')`:

```python
class DigitalInputOutput(virtualbench, lines, reset, vb_name='')
```

**Bases:** `pymeasure.instruments.ni.virtualbench.VirtualBenchInstrument`

Represents Digital Input Output (DIO) Module of Virtual Bench device. Allows to read/write digital channels and/or set channels to export the start signal of FGEN module or trigger of MSO module.

**Methods**

- `export_signal (line, digitalSignalSource)`
  - Exports a signal to the specified line.
  - **Parameters**
    - `line (str)`: Line string
    - `digitalSignalSource (int)`: 0 for FGEN start or 1 for MSO trigger

- `query_export_signal (line)`
  - Indicates the signal being exported on the specified line.
  - **Parameters**
    - `line (str)`: Line string
  - **Returns**
    - Exported signal (FGEN start or MSO trigger)
  - **Return type**
    - `enum`

- `query_line_configuration ()`
  - Indicates the current line configurations. Tristate Lines, Static Lines, and Export Lines contain comma-separated range_data and/or colon-delimited lists of all acquired lines
**read** *(lines)*
Reads the current state of the specified lines.

**Parameters**
- **lines** *(str)* – Line string, requires full name specification e.g. 'VB8012-xxxxxxx/dig/0:7' since instrument_handle is not required (only library_handle)

**Returns**
List of line states (HIGH/LOW)

**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

**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

**write** *(lines, data)*
Writes data to the specified lines.

**Parameters**
- **lines** *(str)* – Line string
- **data** *(list or tuple)* – List of data, (True = High, False = Low)

**class DigitalMultimeter** *(virtualbench, reset, vb_name=“)*

**Bases:** pymeasure.instruments.ni.virtualbench.VirtualBenchInstrument

Represents Digital Multimeter (DMM) Module of Virtual Bench device. Allows to measure either DC/AC voltage or current, Resistance or Diodes.

**configure_ac_current** *(auto_range_terminal)*
Configure auto rage terminal for AC current measurement

**Parameters**
- **auto_range_terminal** – Terminal to perform auto ranging ('LOW' or 'HIGH')

**configure_dc_current** *(auto_range_terminal)*
Configure auto rage terminal for DC current measurement

**Parameters**
- **auto_range_terminal** – Terminal to perform auto ranging ('LOW' or 'HIGH')
configure_dc_voltage\(dmm\_input\_resistance\)
Configure DC voltage input resistance

**Parameters**

- \texttt{dmm\_input\_resistance} \((\text{int or str})\) – Input resistance
  - \{'TEN\_MEGA\_OHM', 'TEN\_GIGA\_OHM'\}

configure_measurement\(dmm\_function, auto\_range=True, manual\_range=1.0\)
Configure Instrument to take a DMM measurement

:doc:`configure_measurement`

**Parameters**

- \texttt{dmm\_function} : DMM function index or name:
  - 'DC\_VOLTS', 'AC\_VOLTS'
  - 'DC\_CURRENT', 'AC\_CURRENT'
  - 'RESISTANCE'
  - 'DIODE'

- \texttt{auto\_range} \((\text{bool})\) – Enable/Disable auto ranging
- \texttt{manual\_range} \((\text{float})\) – 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**

- \text{Auto range, range data}

**Return type**

(\text{bool, float})

read()
Read measurement value from the instrument

**Returns**

- \text{Measurement value}

**Return type**

\text{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**

- \texttt{auto\_range\_terminal} \((\text{int or str})\) – Terminal to perform auto ranging
  - \{'LOW', 'HIGH'\}

**Returns**

- \text{Auto range terminal to pass to the instrument}

**Return type**

\text{int}

validate_dmm_function\(dmm\_function\)
Check if DMM function \texttt{dmm\_function} exists

**Parameters**

- \texttt{dmm\_function} \((\text{int or str})\) – DMM function index or name:
  - 'DC\_VOLTS', 'AC\_VOLTS'
  - 'DC\_CURRENT', 'AC\_CURRENT'
  - 'RESISTANCE'
  - 'DIODE'

**Returns**

- \text{DMM function index to pass to the instrument}

**Return type**

\text{int}
static validate_range(dmm_function, range)
Checks if range is valid for the chosen dmm_function
Parameters
• dmm_function(int) – DMM Function
• range(int or float) – Range value, e.g. maximum value to measure
Returns  Range value to pass to instrument
Return type  int

class FunctionGenerator(virtualbench, reset, vb_name=“)
Bases: pymeasure.instruments.ni.virtualbench.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 %

filter
Enables or disables the filter on the instrument.
Parameters enable_filter(bool) – Enable/Disable filter

query_arbitrary_waveform()
Returns the samples per second for arbitrary waveform generation.
Returns  Samples per second
Return type  int

query_arbitrary_waveform_gain_and_offset()
Returns the settings for arbitrary waveform generation that includes gain and offset settings.
Returns  Gain, DC offset
Return type  (float, float)

query_generation_status()
Returns the status of waveform generation on the instrument.
Returns  Status
Return type  enum
**query_standard_waveform** ()
Returns the settings for a standard waveform generation.

- **Returns**: Waveform function, amplitude, dc_offset, frequency, duty_cycle
- **Return type**: (enum, float, float, float, int)

**query_waveform_mode** ()
Indicates whether the waveform output by the instrument is a standard or arbitrary waveform.

- **Returns**: Waveform mode
- **Return type**: enum

**reset_instrument** ()
Resets the session configuration to default values, and resets the device and driver software to a known state.

**run** ()
Transitions the session from the Stopped state to the Running state.

**self_calibrate** ()
Performs offset nulling calibration on the device. You must run FGEN Initialize prior to running this method.

**shutdown** ()
Removes the session and deallocates any resources acquired during the session. If output is enabled on any channels, they remain in their current state.

**stop** ()
Transitions the acquisition from either the Triggered or Running state to the Stopped state.

**class MixedSignalOscilloscope** (*virtualbench, reset, vb_name="*
Bases: pymeasure.instruments.ni.virtualbench.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**: 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**: 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**: Trigger instance ("A" or "B")
- **Returns**: Trigger instance
- **Return type**: int

**class PowerSupply(virtualbench, reset, vb_name=”)**
Bases: pymeasure.instruments.ni.virtualbench.VirtualBenchInstrument

 Represents Power Supply (PS) Module of Virtual Bench device
configure_current_output (channel, current_level, voltage_limit)
Configures a current output on the specified channel. This method should be called once for every channel you want to configure to output current.

configure_voltage_output (channel, voltage_level, current_limit)
Configures a voltage output on the specified channel. This method should be called once for every channel you want to configure to output voltage.

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 output 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.

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 \texttt{reset} (bool, optional) – Reset the FGEN module, defaults to False

\textbf{acquire\_mixed\_signal\_oscilloscope} \texttt{(reset=False)}

Establishes communication with the MSO module. This method should be called once per session.

Parameters \texttt{reset} (bool, optional) – Reset the MSO module, defaults to False

\textbf{acquire\_power\_supply} \texttt{(reset=False)}

Establishes communication with the PS module. This method should be called once per session.

Parameters \texttt{reset} (bool, optional) – Reset the PS module, defaults to False

\textbf{collapse\_channel\_string} \texttt{(names\_in)}

Collapses a channel string into a comma and colon-delimited equivalent. Last element is the number of channels.

Parameters \texttt{names\_in} (str) – Channel string

Returns Channel string with colon notation where possible, number of channels

Return type (str, int)

\textbf{convert\_timestamp\_to\_values} \texttt{(timestamp)}

Converts a timestamp to seconds and fractional seconds

Parameters \texttt{timestamp} (pyvb.Timestamp) – VirtualBench timestamp

Returns (seconds\_since\_1970, fractional\_seconds)

Return type (int, float)

\textbf{convert\_values\_to\_datetime} \texttt{(timestamp)}

Converts timestamp to datetime object

Parameters \texttt{timestamp} (pyvb.Timestamp) – VirtualBench timestamp

Returns Timestamp as DateTime object

Return type DateTime

\textbf{convert\_values\_to\_timestamp} \texttt{(seconds\_since\_1970, fractional\_seconds)}

Converts seconds and fractional seconds to a timestamp

Parameters

\begin{itemize}
  \item \texttt{seconds\_since\_1970} (int) – Date/Time in seconds since 1970
  \item \texttt{fractional\_seconds} (float) – Fractional seconds
\end{itemize}

Returns VirtualBench timestamp

Return type pyvb.Timestamp

\textbf{expand\_channel\_string} \texttt{(names\_in)}

Expands a channel string into a comma-delimited (no colon) equivalent. Last element is the number of channels. \texttt{'dig/0:2'} -> \texttt{('dig/0, dig/1, dig/2', 3)}

Parameters \texttt{names\_in} (str) – Channel string

Returns Channel string with all channels separated by comma, number of channels

Return type (str, int)

\textbf{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(device_name=", name='VirtualBench')
    Bases: sphinx.ext.autodoc.importer._MockObject

    Represents National Instruments Virtual Bench main frame. This class provides direct access to the arm-strap/pyvirtualbench Python wrapper.

7.19 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.

7.19.1 Oxford Instrument Intelligent Temperature Controller 503

class pymeasure.instruments.oxfordinstruments.ITC503(resourceName,
clear_buffer=True, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument

    Represents the Oxford Intelligent Temperature Controller 503.

    itc = ITC503("GPIB::24")      # Default channel for the ITC503
    itc.control_mode = "RU"        # Set the control mode to remote
    itc.heater_gas_mode = "AUTO"  # Turn on auto heater and flow
    itc.auto_pid = True            # Turn on auto-pid
    print(itc.temperature_setpoint)   # Print the current set-point
    itc.temperature_setpoint = 300  # Change the set-point to 300 K
    itc.wait_for_temperature()      # Wait for the temperature to stabilize
    print(itc.temperature_1)        # Print the temperature at sensor 1

    auto_pid
        A boolean property that sets the Auto-PID mode on (True) or off (False).

    control_mode
        A string property that sets the ITC in LOCAL or REMOTE and LOCKES, or UNLOCKES, the LOC/REM button. Allowed values are: LL: LOCAL & LOCKED RL: REMOTE & LOCKED LU: LOCAL & UN-LOCKED RU: REMOTE & UNLOCKED.

    heater_gas_mode
        A string property that sets the heater and gas flow control to AUTO or MANUAL. Allowed values are: MANUAL: HEATER MANUAL, GAS MANUAL AM: HEATER AUTO, GAS MANUAL MA: HEATER MANUAL, GAS AUTO AUTO: HEATER AUTO, GAS AUTO.

    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

**sweep_status**
An integer property that sets the sweep status. Values are: 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.

**sweep_table**
A property that sets values in the sweep table. Relies on the xpointer and ypointer to point at the location in the table that is to be set.

**temperature_1**
Reads the temperature of the sensor 1 in Kelvin.

**temperature_2**
Reads the temperature of the sensor 2 in Kelvin.

**temperature_3**
Reads the temperature of the sensor 3 in Kelvin.

**temperature_error**
Reads the difference between the set-point and the measured temperature in Kelvin. Positive when set-point is larger than measured.

**temperature_setpoint**
A floating point property that controls the temperature set-point of the ITC in kelvin.

**wait_for_temperature**

```python
(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 set to zero, 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.

**xpointer**
An integer property to set pointers into tables for loading and examining values in the table. For programming the sweep table values from 1 to 16 are allowed, corresponding to the maximum number of steps.
**ypointer**

An integer property to set pointers into tables for loading and examining values in the table. For programming the sweep table the allowed values are: 1: Setpoint temperature, 2: Sweep-time to set-point, 3: Hold-time at set-point.

### 7.20 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.

#### 7.20.1 Parker GV6 Servo Motor Controller

```python
class pymeasure.instruments.parker.ParkerGV6(port)
    Bases: pymeasure.instruments.instrument.Instrument
```

Represents the Parker Gemini GV6 Servo Motor Controller and provides a high-level interface for interacting with the instrument.

- **angle**
  - Returns the angle in degrees based on the position and whether relative or absolute positioning is enabled, returning None on error

- **angle_error**
  - Returns the angle error in degrees based on the position error, or returns None on error

- **disable()**
  - Disables the motor from moving

- **echo (enable=False)**
  - Enables (True) or disables (False) the echoing of all commands that are sent to the instrument

- **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

- **position**
  - Returns an integer number of counts that correspond to the angular position where 1 revolution equals 4000 counts

- **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

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

write (command)
Overwrites the Insturment.write command to provide the correct line break syntax

7.21 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.

7.21.1 DSP 7265 Lock-in Amplifier

class pymeasure.instruments.signalrecovery.DSP7265 (resourceName, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

This is the class for the DSP 7265 lockin amplifier

adc1
Reads the input value of ADC1 in Volts

adc2
Reads the input value of ADC2 in Volts

dac1
A floating point property that represents the output value on DAC1 in Volts. This property can be set.

dac2
A floating point property that represents the output value on DAC2 in Volts. This property can be set.

dac3
A floating point property that represents the output value on DAC3 in Volts. This property can be set.

dac4
A floating point property that represents the output value on DAC4 in Volts. This property can be set.

frequency
A floating point property that represents the lock-in frequency in Hz. This property can be set.
**harmonic**
An integer property that represents the reference harmonic mode control, taking values from 1 to 65535. This property can be set.

**id**
Reads the instrument identification

**mag**
Reads the magnitude in Volts

**phase**
A floating point property that represents the reference harmonic phase in degrees. This property can be set.

**reference**
Controls the oscillator reference. Can be “internal”, “external rear” or “external front”

**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.

**setDifferentialMode**(lineFiltering=True)
Sets lockin to differential mode, measuring A-B

**shutdown**()
Brings the instrument to a safe and stable state

**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.

**time_constant**
A floating point property that controls the time constant in seconds, which takes values from 10 microseconds to 50,000 seconds. This property can be set.

**values**(command)
Rewrite the method because of extra character in return string.

**voltage**
A floating point property that represents the voltage in Volts. This property can be set.

**x**
Reads the X value in Volts

**xy**
Reads both the X and Y values in Volts

**y**
Reads the Y value in Volts

### 7.22 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.
7.22.1 SR830 Lock-in Amplifier

class pymeasure.instruments.srs.SR830(resourceName, **kwargs)
   Bases: pymeasure.instruments.instrument.Instrument

    adc1
       Reads the Aux input 1 value in Volts with 1/3 mV resolution.

    adc2
       Reads the Aux input 2 value in Volts with 1/3 mV resolution.

    adc3
       Reads the Aux input 3 value in Volts with 1/3 mV resolution.

    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

    aux_in_1
       Reads the Aux input 1 value in Volts with 1/3 mV resolution.

    aux_in_2
       Reads the Aux input 2 value in Volts with 1/3 mV resolution.

    aux_in_3
       Reads the Aux input 3 value in Volts with 1/3 mV resolution.

    aux_in_4
       Reads the Aux input 4 value in Volts with 1/3 mV resolution.

    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.

    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.

    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.

    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.

    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.

    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.

    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.
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.

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.

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.

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.

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.

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.

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 precent and the exapnsion term that are used to scale the channel in question.

harmonic
An integer property that controls the harmonic that is measured. Allowed values are 1 to 19999. Can be set.

input_config
An string property that controls the input configuration. Allowed values are: ['A', 'A - B', 'I (1 MOhm)', 'I (100 MOhm)']

input_coupling
An string property that controls the input coupling. Allowed values are: ['AC', 'DC']

input_grounding
An string property that controls the input shield grounding. Allowed values are: ['Float', 'Ground']

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.

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)

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.

reference_source
An string property that controls the reference source. Allowed values are: ['External', 'Internal']

sample_frequency
Gets the sample frequency in Hz.
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, percent, expand=0)
Sets the offset of a channel (X=1, Y=2, R=3) to a certain percent (-105% to 105%) of the signal, with an optional expansion term (0, 10=1, 100=2)

sine_voltage
A floating point property that represents the reference sine-wave voltage in Volts. This property can be set.

theta
Reads the theta value in degrees.

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

x
Reads the X value in Volts.

y
Reads the Y value in Volts.

7.23 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.

7.23.1 TDS2000 Oscilloscope

class pymeasure.instruments.tektronix.TDS2000(resourceName, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

Represents the Tektronix TDS 2000 Oscilloscope and provides a high-level for interacting with the instrument.

7.23.2 AFG3152C Arbitrary function generator

class pymeasure.instruments.tektronix.AFG3152C(adapter, **kwargs)
Bases: pymeasure.instruments.instrument.Instrument

Represents the Tektronix AFG 3000 series (one or two channels) arbitrary function generator and provides a high-level for interacting with the instrument.

afg=AFG3152C("GPIB::1") # AFG on GPIB 1 afg.reset() # Reset to default afg.ch1.shape='sinusoidal' # Sinusoidal shape afg.ch1.unit='VPP' # Sets CH1 unit to VPP afg.ch1.amp_vpp=1 # Sets the CH1 level to 1 VPP afg.ch1.frequency=1e3 # Sets the CH1 frequency to 1KHz afg.ch1.enable() # Enables the output from CH1
7.24 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.

7.24.1 Thorlabs PM100USB Powermeter

class pymeasure.instruments.thorlabs.ThorlabsPM100USB(adapter, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument
    Represents Thorlabs PM100USB powermeter

    measure_power (wavelength)
        Set wavelength in nm and get power in W If wavelength is out of range it will be set to range limit

    power
        Power, in Watts

    sensor()
        Get sensor info

    wavelength
        Wavelength in nm; not set outside of range

    wavelength_max
        Get maximum wavelength, in nm

    wavelength_min
        Get minimum wavelength, in nm

class pymeasure.instruments.thorlabs.ThorlabsPro8000(resourceName, **kwargs)
    Bases: pymeasure.instruments.instrument.Instrument
    Represents Thorlabs Pro 8000 modular laser driver

    LDCCurrent
        Laser current.

    LDCCurrentLimit
        Set Software current Limit (value must be lower than hardware current limit).

    LDCPolarity
        Set laser diode polarity. Allowed values are: ['AG', 'CG']

    LDCStatus
        Set laser diode status. Allowed values are: ['ON', 'OFF']

    TEDSetTemperature
        Set TEC temperature

    TEDStatus
        Set TEC status. Allowed values are: ['ON', 'OFF']

    slot
        Slot selection. Allowed values are: range(1, 9)
7.25 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.

7.25.1 Yokogawa 7651 Programmable Supply

class pymeasure.instruments.yokogawa.Yokogawa7651(adapter, **kwargs)

Represents the Yokogawa 7651 Programmable DC Source and provides a high-level for interacting with the instrument.

```python
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.

compliance_current

A floating point property that sets the compliance current in Amps, which can take values from 5 to 120 mA.

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.

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.

**source_current**

A floating point property that controls the source current in Amps, if that mode is active.

**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.

**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.

**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.

**source_voltage**

A floating point property that controls the source voltage in Volts, if that mode is active.

**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.
Contributions to the instrument repository and the main code base are highly encouraged. This section outlines the basic workflow for new contributors.

## 8.1 Using the development version

New features are added to the development version of PyMeasure, hosted on GitHub. We use Git version control to track and manage changes to the source code. On Windows, we recommend using GitHub Desktop. Make sure you have an appropriate version of Git (or GitHub Desktop) installed and that you have a GitHub account.

In order to add your feature, you need to first fork PyMeasure. This will create a copy of the repository under your GitHub account.

The instructions below assume that you have set up Anaconda, as described in the Quick Start guide and describe the terminal commands necessary. If you are using GitHub Desktop, take a look through their documentation to understand the corresponding steps.

Clone your fork of PyMeasure `your-github-username/pymeasure`. In the following terminal commands replace your desired path and GitHub username.

```
cd /path/for/code
git clone https://github.com/your-github-username/pymeasure.git
```

If you had already installed PyMeasure using `pip`, make sure to uninstall it before continuing.

```
pip uninstall pymeasure
```

Install PyMeasure in the editable mode.

```
cd /path/for/code/pymeasure
pip install -e .
```
This will allow you to edit the files of PyMeasure and see the changes reflected. Make sure to reset your notebook kernel or Python console when doing so. Now you have your own copy of the development version of PyMeasure installed!

## 8.2 Working on a new feature

We use branches in Git to allow multiple features to be worked on simultaneously, without causing conflicts. The master branch contains the stable development version. Instead of working on the master branch, you will create your own branch off the master and merge it back into the master when you are finished.

Create a new branch for your feature before editing the code. For example, if you want to add the new instrument “Extreme 5000” you will make a new branch “dev/extreme-5000”.

```bash
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.

```bash
git fetch
```

Once you have created the branch, change your current branch to match the new one.

```bash
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.

```bash
git add file-that-changed.py
git commit -m "A short description about what changed"
```

## 8.3 Making a pull-request

While you are working, it’s helpful to start a pull-request (PR) on the `master` branch of *ralph-group/pymeasure*. This will allow you to discuss your feature with other contributors. We encourage you to start this pull-request after your first commit.

[Start a pull-request](https://github.com/ralph-group/pymeasure) on the PyMeasure GitHub page.

Your pull-request will be merged by the PyMeasure maintainers once it meets the coding standards and passes unit tests. You will notice that your pull-request is automatically checked with the unit tests.

## 8.4 Unit testing

Unit tests are run each time a new commit is made to a branch. The purpose is to catch changes that break the current functionality, by testing each feature unit. PyMeasure relies on `pytest` to preform these tests, which are run on TravisCI and Appveyor for Linux/macOS and Windows respectively.

Running the unit tests while you develop is highly encouraged. This will ensure that you have a working contribution when you create a pull request.
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!

```
python setup.py test
```
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 issue before posting.
You can make a significant contribution to PyMeasure by adding a new instrument to the `pymeasure.instruments` package. Even adding an instrument with a few features can help get the ball rolling, since its likely that others are interested in the same instrument.

Before getting started, become familiar with the `contributing work-flow` for PyMeasure, which steps through the process of adding a new feature (like an instrument) to the development version of the source code. This section will describe how to lay out your instrument code.

## 10.1 File structure

Your new instrument should be placed in the directory corresponding to the manufacturer of the instrument. For example, if you are going to add an “Extreme 5000” instrument you should add the following files assuming “Extreme” is the manufacturer. Use lowercase for all filenames to distinguish packages from CamelCase Python classes.

```
|-> pymeasure/pymeasure/instruments/extreme/
  |-> __init__.py
  |-> extreme5000.py
```

### 10.1.1 Updating the init file

The `__init__.py` file in the manufacturer directory should import all of the instruments that correspond to the manufacturer, to allow the files to be easily imported. For a new manufacturer, the manufacturer should also be added to `pymeasure/pymeasure/instruments/__init__.py`.

### 10.1.2 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.
Copy an existing instrument documentation file, which will automatically generate the documentation for the instrument. The `index.rst` file should link to the `extreme5000.rst`. For a new manufacturer, the manufacturer should be also linked in `pymeasure/docs/api/instruments/index.rst`.

### 10.2 Instrument file

All standard instruments should be child class of `Instrument`. This provides the basic functionality for working with `Adapters`, which perform the actual communication.

The most basic instrument, for our “Extreme 5000” example starts like this:

```python
# This file is part of the PyMeasure package.
#
# Copyright (c) 2013-2020 PyMeasure Developers
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
# THE SOFTWARE.
#
# from pymeasure.instruments import Instrument

class Extreme5000(Instrument):
    """ Represents the imaginary Extreme 5000 instrument. """

    def __init__(self, resourceName, **kwargs):
        super(Extreme5000, self).__init__(resourceName,
                                           "Extreme 5000",
                                           **kwargs
        )
```

Make sure to include the PyMeasure license to each file, and add yourself as an author to the `AUTHORS.txt` file.
In principle you are free to write any functions 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 command instead of calling the adapter directly.

In practice, we have developed a number of convenience functions for making instruments easy to write and maintain. The following sections detail these conveniences and are highly encouraged.

### 10.3 Writing properties

In PyMeasure, **Python properties** are the preferred method for dealing with variables that are read or set. PyMeasure comes with two convenience functions for making properties for classes. The `Instrument.measurement` function returns a property that issues a GPIB/SCPI requests when the value is used. For example, if our “Extreme 5000” has the `*IDN?` command we can write the following property to be added above the `def __init__` line in our above example class, or added to the class after the fact as in the code here:

```python
Extreme5000.id = Instrument.measurement(
    "*IDN?", """ Reads the instrument identification """
)
```

You will notice that a documentation string is required, and should be descriptive and specific.

When we use this property we will get the identification information.

```python
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.id
'Extreme 5000 identification from instrument'
```

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.

```python
Extreme5000.voltage = Instrument.control(
    "::VOLT?", "::VOLT %g",
    """ A floating point property that controls the voltage
    in Volts. This property can be set.
    """
)
```

You will notice that we use the Python string format `%g` to pass through the floating point.

We can use this property to set the voltage to 100 mV, which will execute the command and then request the current voltage.

```python
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.1 # Executes "::VOLT 0.1"
>>> extreme.voltage
0.1
```

Using both of these functions, you can create a number of properties for basic measurements and controls. The next section details additional features of `Instrument.control` that allow you to write properties that cover specific ranges, or have to map between a real value to one used in the command.
10.4 Advanced properties

Many GPIB/SCIP 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.

10.4.1 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.

```python
Extreme5000.voltage = Instrument.control(
    ">:VOLT?", ":VOLT %g",
    """" A floating point property that controls the voltage
    in Volts, from -1 to 1 V. This property can be set. """
    validator=strict_range,
    values=[-1, 1]
)
```

Now our voltage will raise a `ValueError` if the value is out of the range.

```python
>>> 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.

```python
Extreme5000.voltage = Instrument.control(
    ">:VOLT?", ":VOLT %g",
    """" A floating point property that controls the voltage
    in Volts, from -1 to 1 V. Invalid voltages are truncated.
    This property can be set. """
    validator=truncated_range,
    values=[-1, 1]
)
```

Now our voltage will not raise an error, and will truncate the value to the range bounds.

```python
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 100 # Executes ":VOLT 1"
>>> extreme.voltage
1.0
```
10.4.2 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.

```python
Extreme5000.voltage = Instrument.control(
    "::RANG?", "::RANG %g",
    " A floating point property that controls the voltage range in Volts. This property can be set. ",
    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.

```python
>>> extreme = Extreme5000("GPIB::1")
>>> extreme.voltage = 0.08
>>> extreme.voltage
0.1
```

10.4.3 Using maps

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.

```python
Extreme5000.voltage = Instrument.control(
    "::RANG?", "::RANG %d",
    " A floating point property that controls the voltage range in Volts, which takes values of 10 mV, 100 mV and 1 V. This property can be set. ",
    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.

```python
>>> 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.

```python
Extreme5000.voltage = Instrument.control(
    "::RANG?", "::RANG %d",
    """A floating point property that controls the voltage
    range in Volts, which takes values of 10 mV, 100 mV and 1 V.
    This property can be set. """,
    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:

```python
Extreme5000.channel = Instrument.control(
    "::CHAN?", "::CHAN %d",
    """A string property that controls the measurement channel,
    which can take the values X, Y, or 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.
In order to maintain consistency across the different instruments in the PyMeasure repository, we enforce the following standards.

11.1 Python style guides

Only Python 3 is used in PyMeasure. This prevents the maintainance overhead of supporting Python 2.7, which will lose official support in the future.

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.

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.

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 Advanced properties section, these types of properties are preferred.
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
Vikram Sekar
Casper Schippers
Sumatran Tiger
Dennis Feng
Stefano Pirotta
Moritz Jung
Copyright (c) 2013-2020 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.
pymeadure.display.browser, 43
pymeadure.display.curves, 43
pymeadure.display.inputs, 44
pymeadure.display.listeners, 46
pymeadure.display.log, 46
pymeadure.display.manager, 46
pymeadure.display.plotter, 48
pymeadure.display.thread, 48
pymeadure.display.widgets, 48
pymeadure.display.windows, 50
pymeadure.experiment.experiment, 35
pymeadure.experiment.listeners, 36
pymeadure.experiment.parameters, 38
pymeadure.experiment.procedure, 37
pymeadure.experiment.results, 41
pymeadure.experiment.workers, 40
pymeadure.instruments, 52
pymeadure.instruments.advantest, 57
pymeadure.instruments.advantest.advantestR3767CG, 58
pymeadure.instruments.agilent, 58
pymeadure.instruments.agilent.agilent4156, 64
pymeadure.instruments.ametek, 76
pymeadure.instruments.ami, 78
pymeadure.instruments.anritsu, 79
pymeadure.instruments.comedi, 57
pymeadure.instruments.danfysik, 82
pymeadure.instruments.deltaelektronika, 85
pymeadure.instruments.fwbell, 86
pymeadure.instruments.hp, 87
pymeadure.instruments.keithley, 88
pymeadure.instruments.keysight, 113
pymeadure.instruments.lakeshore, 114
pymeadure.instruments.newport, 116
pymeadure.instruments.ni, 118
pymeadure.instruments.oxfordinstruments, 129
pymeadure.instruments.parker, 131
pymeadure.instruments.signalrecovery, 132
pymeadure.instruments.srs, 133
pymeadure.instruments.tektronix, 136
pymeadure.instruments.thorlabs, 136
pymeadure.instruments.validators, 55
pymeadure.instruments.yokogawa, 137
A

abort() (pymeasure.display.manager.Manager method), 47
ac_current (pymeasure.instruments.agilent.AgilentE4980 attribute), 62
ac_mode() (pymeasure.instruments.lakeshore.LakeShore425 method), 116
ac_voltage (pymeasure.instruments.agilent.AgilentE4980 attribute), 62
acquire_digital_input_output() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 127
acquire_digital_multimeter() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 127
acquire_function_generator() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 127
acquire_mixed_signal_oscilloscope() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
acquire_power_supply() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
acquire_reference() (pymeasure.instruments.keithley.Keithley2000 method), 89
Adapter (class in pymeasure.adapters), 27
adc1 (pymeasure.instruments.ametek.Ametek7270 attribute), 76
adc1 (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
adc1 (pymeasure.instruments.srs.SR830 attribute), 134
adc2 (pymeasure.instruments.ametek.Ametek7270 attribute), 76
adc2 (pymeasure.instruments.ametek.Ametek7270 attribute), 134
adc3 (pymeasure.instruments.srs.SR830 attribute), 134
adc4 (pymeasure.instruments.ametek.Ametek7270 attribute), 76
adc4 (pymeasure.instruments.srs.SR830 attribute), 134
add() (pymeasure.display.browser.Browser method), 43
add_ramp_step() (pymeasure.instruments.danfysik.Danfysik8500 method), 83
AdvantestR3767CG (class in pymeasure.instruments.advantest.advantestR3767CG), 58
AGFG3152C (class in pymeasure.instruments.tektronix), 136
Agilent33220A (class in pymeasure.instruments.agilent), 70
Agilent33500 (class in pymeasure.instruments.agilent), 73
Agilent33521A (class in pymeasure.instruments.agilent), 76
Agilent34410A (class in pymeasure.instruments.agilent), 64
Agilent4156 (class in pymeasure.instruments.agilent.agilent4156), 64
Agilent8257D (class in pymeasure.instruments.agilent), 58
Agilent8722ES (class in pymeasure.instruments.agilent), 61
AgilentE4408B (class in pymeasure.instruments.agilent), 62
AgilentE4980 (class in pymeasure.instruments.agilent), 62
Ametek7270 (class in pymeasure.instruments.ametek), 76
AMI430 (class in pymeasure.instruments.ami), 78
amplitude (pymeasure.instruments.agilent.Agilent33220A attribute), 71
amplitude (pymeasure.instruments.agilent.Agilent33500 attribute), 71
attribute, 73

amplitude (pymeasure.instruments.hp.HP33120A

attribute), 87

amplitude_depth (pymeasure.instruments.agilent.Agilent8257D

attribute), 58

amplitude_source (pymeasure.instruments.agilent.Agilent8257D

attribute), 58

amplitude_unit (pymeasure.instruments.agilent.Agilent33220A

attribute), 71

amplitude_unit (pymeasure.instruments.agilent.Agilent33500

attribute), 73

amplitude_units (pymeasure.instruments.hp.HP33120A

attribute), 87

amplitude_unit (pymeasure.instruments.agilent.Agilent33500

attribute), 73

analysis (pymeasure.instruments.anritsu.AnritsuMS9710C

attribute), 80

analysis_result (pymeasure.instruments.anritsu.AnritsuMS9710C

attribute), 80

analyzer_mode (pymeasure.instruments.agilent.Agilent4156.Agilent4156

attribute), 66

angle (pymeasure.instruments.parker.ParkerGV6

attribute), 131

angle_error (pymeasure.instruments.parker.ParkerGV6

attribute), 131

AnritsuMG3692C (class in pymeasure.instruments.anritsu), 79

AnritsuMS9710C (class in pymeasure.instruments.anritsu), 80

aperture() (pymeasure.instruments.agilent.AgilentE4980

method), 62

append() (pymeasure.display.curves.BufferCurve

method), 43

apply_current() (pymeasure.instruments.keithley.Keithley2400

method), 95

apply_current() (pymeasure.instruments.keithley.Keithley2450

method), 101

apply_current() (pymeasure.instruments.yokogawa.Yokogawa7651

method), 138

apply_voltage() (pymeasure.instruments.keithley.Keithley2400

method), 95

apply_voltage() (pymeasure.instruments.keithley.Keithley2450

method), 101

arb_advance (pymeasure.instruments.agilent.Agilent33500

attribute), 73

arb_file (pymeasure.instruments.agilent.Agilent33500

attribute), 73

arb_filter (pymeasure.instruments.agilent.Agilent33500

attribute), 73

arb_srate (pymeasure.instruments.agilent.Agilent33500

attribute), 73

arb_srate (pymeasure.instruments.agilent.Agilent33521A

attribute), 76

ask() (pymeasure.adapters.Adapter method), 27

ask() (pymeasure.adapters.FakeAdapter method), 28

ask() (pymeasure.adapters.PrologixAdapter method), 30

ask() (pymeasure.adapters.VISAAdapter method), 31

ask() (pymeasure.adapters.VXI11Adapter method), 33

ask() (pymeasure.instruments.fwbell.FWBell5080

method), 86

ask() (pymeasure.instruments.Instrument method), 53

ask() (pymeasure.instruments.lakeshore.LakeShoreUSBAdapter

method), 114

ask() (pymeasure.adapters.VXI11Adapter

method), 33

ask_values() (pymeasure.adapters.VISAAdapter

method), 31

auto_offset() (pymeasure.instruments.srs.SR830

method), 134

auto_output_off (pymeasure.instruments.keithley.Keithley2400

attribute), 95

auto_pid (pymeasure.instruments.oxfordinstruments.ITC503

attribute), 129

auto_range() (pymeasure.instruments.fwbell.FWBell5080

method), 86

auto_range() (pymeasure.instruments.keithley.Keithley2000

method), 89

auto_range() (pymeasure.instruments.lakeshore.LakeShore425

method), 116

auto_range_source() (pymeasure.instruments.keithley.Keithley2400

method), 95

auto_range_source() (pymeasure.instruments.keithley.Keithley2450

method), 101
auto_setup() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 124
auto_zero(pymeasure.instruments.keithley.Keithley2400 attribute), 95
aux_in_1 (pymeasure.instruments.srs.SR830 attribute), 134
aux_in_2 (pymeasure.instruments.srs.SR830 attribute), 134
aux_in_3 (pymeasure.instruments.srs.SR830 attribute), 134
aux_in_4 (pymeasure.instruments.srs.SR830 attribute), 134
aux_out_1 (pymeasure.instruments.srs.SR830 attribute), 134
aux_out_2 (pymeasure.instruments.srs.SR830 attribute), 134
aux_out_3 (pymeasure.instruments.srs.SR830 attribute), 134
aux_out_4 (pymeasure.instruments.srs.SR830 attribute), 134
average_point (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
average_sweep (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
axes (pymeasure.instruments.newport.ESP300 attribute), 117
Axis (class in pymeasure.instruments.newport.esp300), 117
AxisError (class in pymeasure.instruments.newport.esp300), 118

beep() (pymeasure.instruments.agilent.Agilent33220A method), 71
beep() (pymeasure.instruments.agilent.Agilent33500 method), 73
beep() (pymeasure.instruments.hp.HP33120A method), 87
beep() (pymeasure.instruments.keithley.Keithley2000 method), 89
beep() (pymeasure.instruments.keithley.Keithley2400 method), 95
beep() (pymeasure.instruments.keithley.Keithley2450 method), 101
beep() (pymeasure.instruments.keithley.Keithley2700 method), 105
beep() (pymeasure.instruments.keithley.Keithley6221 method), 108
beep_state (pymeasure.instruments.keithley.Keithley2000 attribute), 89
beep_state (pymeasure.instruments.keithley.Keithley2400 attribute), 89
beeper_state (pymeasure.instruments.agilent.Agilent33220A attribute), 71
binary_values() (pymeasure.adapters.Adapter method), 27
binary_values() (pymeasure.adapters.FakeAdapter method), 28
binary_values() (pymeasure.adapters.PrologixAdapter method), 30
binary_values() (pymeasure.adapters.SerialAdapter method), 29
binary_values() (pymeasure.adapters.VISAAdapter method), 31
binary_values() (pymeasure.adapters.VXI11Adapter method), 33
Buffering (class in pymeasure.display.inputs), 44
Browser (class in pymeasure.display.browser), 43
BrowserItem (class in pymeasure.display.browser), 43
BrowserWidget (class in pymeasure.display.widgets), 48
buffer_data (pymeasure.instruments.keithley.Keithley2000 attribute), 89
buffer_data (pymeasure.instruments.keithley.Keithley2400 attribute), 95
buffer_data (pymeasure.instruments.keithley.Keithley2450 attribute), 101
buffer_data (pymeasure.instruments.keithley.Keithley2700 attribute), 105
buffer_data (pymeasure.instruments.keithley.Keithley6221 attribute), 108
buffer_data (pymeasure.instruments.keithley.Keithley2000 attribute), 89
buffer_data (pymeasure.instruments.keithley.Keithley2400 attribute), 95
buffer_data (pymeasure.instruments.keithley.Keithley2450 attribute), 101
buffer_data (pymeasure.instruments.keithley.Keithley2700 attribute), 106
buffer_data (pymeasure.instruments.keithley.Keithley6221 attribute), 108

Index 163
BufferCurve (class in pymeasure.display.curves), 43

burst_mode (pymeasure.instruments.agilent.Agilent33220A attribute), 71

burst_mode (pymeasure.instruments.agilent.Agilent33500 attribute), 73

burst_ncycles (pymeasure.instruments.agilent.Agilent33220A attribute), 71

burst_ncycles (pymeasure.instruments.agilent.Agilent33500 attribute), 73

burst_period (pymeasure.instruments.agilent.Agilent33500 attribute), 74

burst_state (pymeasure.instruments.agilent.Agilent33220A attribute), 71

burst_state (pymeasure.instruments.agilent.Agilent33500 attribute), 74

center_at_peak() (pymeasure.instruments.anritsu.AnritsuMS9710C method), 80

center_frequency (pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG attribute), 58

center_frequency (pymeasure.instruments.agilent.Agilent8257D attribute), 59

center_frequency (pymeasure.instruments.agilent.AgilentE4408B attribute), 62

channel1 (pymeasure.instruments.srs.SR830 attribute), 134

channel12 (pymeasure.instruments.srs.SR830 attribute), 134

channel_function (pymeasure.instruments.agilent.Agilent4156.SMU attribute), 67

channel_function (pymeasure.instruments.agilent.Agilent4156.SMU attribute), 70

channel_mode (pymeasure.instruments.agilent.Agilent4156.VSU attribute), 70

channels_from_rows_columns() (pymeasure.instruments.keithley.Keithley2700 method), 106

check_errors() (pymeasure.instruments.agilent.Agilent33220A method), 71

check_errors() (pymeasure.instruments.agilent.Agilent33500 method), 74

check_errors() (pymeasure.instruments.Instrument method), 53

check_errors() (pymeasure.instruments.keithley.Keithley2000 method), 89

check_errors() (pymeasure.instruments.keithley.Keithley2400 method), 95

check_errors() (pymeasure.instruments.keithley.Keithley2450 method), 101

check_errors() (pymeasure.instruments.keithley.Keithley2700 method), 106

check_errors() (pymeasure.instruments.keithley.Keithley2750 method), 112

check_errors() (pymeasure.instruments.keithley.Keithley6221 method), 108

check_errors() (pymeasure.instruments.keysight.KeysightN5767A method), 113

check_parameters() (pymeasure.experiment.procedure.Procedure method), 37

check_stop() (pymeasure.display.windows.PlotterWindow method), 52

choices (pymeasure.experiment.parameters.ListParameter attribute), 39

clear() (pymeasure.display.manager.Manager method), 47

clear() (pymeasure.instruments.Instrument method), 53

clear_display() (pymeasure.instruments.agilent.Agilent33500 method), 74

clear_errors() (pymeasure.instruments.newport.ESP300 method), 117

clear_plot() (pymeasure.
clear_ramp_set() (PyMeasure. Danfysik.Danfysik8500 method), 36

clear_sequence() (PyMeasure. Danfysik.Danfysik8500 method), 83

close() (PyMeasure. Keithley. Keithley2750 method), 112

close_rows_to_columns() (PyMeasure. Keithley. Keithley2700 method), 106

closed_channels (PyMeasure. Keithley. Keithley2700 attribute), 106

closed_channels (PyMeasure. Keithley. Keithley2750 attribute), 112

collapse_channel_string() (PyMeasure. VirtualBench.VirtualBench method), 128

compliance (PyMeasure. Agilent. Agilent4156.SMU attribute), 68

compliance (PyMeasure. Agilent. Agilent4156.VARD attribute), 69

compliance (PyMeasure. Agilent. Agilent4156.VARX attribute), 69

collapse_channel_string() (PyMeasure. VirtualBench.VirtualBench.MixedSignalOscilloscope method), 124

collapse_channel_string() (PyMeasure. VirtualBench.VirtualBench.FunctionGenerator method), 122

close() (PyMeasure. VirtualBench.VirtualBench.PowerSupply method), 126

Index 165
configure_dc_voltage() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 120
configure_immediate_trigger() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
configure_measurement() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
configure_standard_waveform() (pymeasure.instruments.agilent.Agilent4156.FunctionGenerator method), 122
configure_timing() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
configure_trigger_delay() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
configure_voltage_output() (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply method), 127

constant_value (pymeasure.instruments.agilent.agilent4156.SMU attribute), 68
constant_value (pymeasure.instruments.agilent.agilent4156.VSU attribute), 70

control() (pymeasure.instruments.Instrument static method), 53
control_mode (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 129
convert_timestamp_to_values() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
convert_values_to_datetime() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
convert_values_to_timestamp() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
create_filename() (in module pymeasure.experiment.experiment), 36

Crosshairs (class in pymeasure.display.curves), 44
CSVFormatter (class in pymeasure.experiment.results), 41

current (pymeasure.instruments.danfysik.Danfysik8500 attribute), 83
current (pymeasure.instruments.deltaelektronika.SM7045D attribute), 85
current (pymeasure.instruments.keithley.Keithley2000 attribute), 89
current (pymeasure.instruments.keithley.Keithley2400 attribute), 96
current (pymeasure.instruments.keithley.Keithley2450 attribute), 102
current (pymeasure.instruments.keysight.KeysightN5767A attribute), 113
current (pymeasure.instruments.agilent.Agilent34410A attribute), 64
current (pymeasure.instruments.hp.HP34401A attribute), 88
current_ac_bandwidth (pymeasure.instruments.agilent.Agilent34410A attribute), 89
current_ac_digits (pymeasure.instruments.agilent.Agilent34410A attribute), 89
current_ac_nplc (pymeasure.instruments.agilent.Agilent34410A attribute), 89
current_ac_range (pymeasure.instruments.agilent.Agilent34410A attribute), 89
current_ac_reference (pymeasure.instruments.agilent.Agilent34410A attribute), 90
current_ac_ppm (pymeasure.instruments.agilent.Agilent34410A attribute), 90
current_ac_range (pymeasure.instruments.agilent.Agilent34410A attribute), 90
current_ac_range (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 90

current_ac_range (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 90

current_ac_ppm (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 90

current_nplc (pymeasure.instruments.agilent.Agilent34410A attribute), 90

current_nplc (pymeasure.instruments.agilent.Agilent34410A attribute), 90

current_nplc (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 90


current_ppm (pymeasure.instruments.danfysik.Danfysik8500 attribute), 83
current_range (pymeasure.instruments.keithley.Keithley2000 attribute), 90
current_range (pymeasure.instruments.keithley.Keithley2400 attribute), 102
tribute), 96
current_range (pymeasure.instruments.keithley.Keithley2450 attribute), 102
current_range (pymeasure.instruments.keysight.KeysightN5767A attribute), 113
current_reference (pymeasure.instruments.keithley.Keithley2000 attribute), 90
current_setpoint (pymeasure.instruments.danfysik.Danfysik8500 attribute), 83
dac1 (pymeasure.instruments.ametek.Ametek7270 attribute), 76
dac1 (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
dac1 (pymeasure.instruments.srs.SR830 attribute), 134
dac2 (pymeasure.instruments.ametek.Ametek7270 attribute), 76
dac2 (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
dac2 (pymeasure.instruments.srs.SR830 attribute), 134
dac3 (pymeasure.instruments.ametek.Ametek7270 attribute), 77
dac3 (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
dac3 (pymeasure.instruments.srs.SR830 attribute), 135
dac4 (pymeasure.instruments.ametek.Ametek7270 attribute), 77
dac4 (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
dac4 (pymeasure.instruments.srs.SR830 attribute), 135
Danfysik8500 (class in pymeasure.instruments.danfysik), 82
DanfysikAdapter (class in pymeasure.instruments.danfysik), 82
data (pymeasure.experiment.experiment.Experiment attribute), 36
data (pymeasure.instruments.agilent.Agilent8722ES attribute), 61
data_arb() (pymeasure.instruments.agilent.Agilent33500 method), 74
data_memory_a_condition (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_a_size (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_a_values (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_b_condition (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_b_size (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_b_values (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_memory_select (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
data_variables (pymeasure.instruments.agilent.agilent4156.Agilent4156 attribute), 66
datavolatile_clear() (pymeasure.instruments.agilent.Agilent33500 method), 74
dc_mode() (pymeasure.instruments.lakeshore.LakeShore425 method), 116
define_arbitrary_waveform() (pymeasure.instruments.keithley.Keithley6221 method), 109
define_position() (pymeasure.instruments.newport.esp300.Axis method), 117
delay_time (pymeasure.instruments.agilent.agilent4156.Agilent4156 attribute), 66
determine_valid_channels() (pymeasure.instruments.keithley.Keithley2700 method), 106
dip_search (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 80
disable (pymeasure.instruments.agilent.agilent4156.SMU attribute), 68
disable (pymeasure.instruments.agilent.agilent4156.VMU attribute), 70
disable (pymeasure.instruments.agilent.agilent4156.VSU attribute), 70
disable() (pymeasure.instruments.agilent.Agilent8257D method), 59
disable() (pymeasure.instruments.anritsu.AnritsuMG3692C method), 79
disable() (pymeasure.instruments.danfysik.Danfysik8500 method), 83
disable() (pymeasure.instruments.deltaelektronika.SM7045D method), 85
disable() (pymeasure.instruments.keysight.KeysightN5767A method), 113
disable() (pymeasure.instruments.newport.ESP300 method), 117
disable() (pymeasure.instruments.newport.esp300.Axis method), 117
disable_source() (pymeasure.instruments.keithley.Keithley2400 method), 96
disable_source() (pymeasure.instruments.keithley.Keithley2450 method), 102
disable_source() (pymeasure.instruments.keithley.Keithley6221 method), 109
disable_source() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 138
display () (pymeasure.instruments.agilent.Agilent33500 attribute), 74
display_closed_channels() (pymeasure.instruments.keithley.Keithley2700 attribute), 106
display_enabled (pymeasure.instruments.keithley.Keithley2400 attribute), 96
display_enabled (pymeasure.instruments.keithley.Keithley6221 attribute), 109
display_text (pymeasure.instruments.keithley.Keithley2000 attribute), 90
disp7265 () (class in pymeasure.instruments.signalrecovery), 132
dwell_time (pymeasure.instruments.agilent.Agilent8257D attribute), 59
echo() (pymeasure.instruments.parker.ParkerGV6 method), 131
emit() (pymeasure.display.log.LogHandler method), 46
emit() (pymeasure.experiment.workers.Worker method), 40
enable() (pymeasure.instruments.agilent.Agilent8257D method), 59
enable() (pymeasure.instruments.anritsu.AnritsuMG3692C method), 80
enable() (pymeasure.instruments.danfysik.Danfysik8500 method), 83
enable() (pymeasure.instruments.deltaelektronika.SM7045D method), 85
enable() (pymeasure.instruments.keysight.KeysightN5767A method), 113
enable() (pymeasure.instruments.newport.ESP300Axis method), 117
enable() (pymeasure.instruments.newport.esp300.Axis method), 117
enable() (pymeasure.instruments.parker.ParkerGV6 method), 131
enable_amplitude_modulation() (pymeasure.instruments.agilent.Agilent8257D method), 59
enable_filter() (pymeasure.instruments.keithley.Keithley2000 method), 90
enable_low_freq_out() (pymeasure.instruments.agilent.Agilent8257D method), 59
enable_persistent_switch() (pymeasure.instruments.ami.AMI430 method), 78
enable_pulse_modulation() (pymeasure.instruments.agilent.Agilent8257D method), 59
enable_reference() (pymeasure.instruments.keithley.Keithley2000 method), 90
enable_source() (pymeasure.instruments.keithley.Keithley2400 method), 96
enable_source() (pymeasure.instruments.keithley.Keithley2450 method), 102
enable_source() (pymeasure.instruments.keithley.Keithley6221 method), 109
enable_source() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 138
enabled (pymeasure.instruments.newport.esp300.Axis attribute), 117
error (pymeasure.instruments.keithley.Keithley2400 attribute), 96
error (pymeasure.instruments.keithley.Keithley2450 attribute), 102
error (pymeasure.instruments.keithley.Keithley2700 attribute), 106
error (pymeasure.instruments.keithley.Keithley6221 attribute), 109
error (pymeasure.instruments.newport.ESP300 attribute), 117
errors (pymeasure.instruments.newport.ESP300 attribute), 117
ese2 (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81
ESP300 (class in pymeasure.instruments.newport), 117
esr2 (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81
eval_string() (pymeasure.display.widgets.SequencerWidget static method), 49
execute() (pymeasure.experiment.procedure.Procedure method), 37
expand_channel_string() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 128
Experiment (class in pymeasure.display.manager), 46
Experiment (class in pymeasure.experiment.experiment), 35
ExperimentQueue (class in pymeasure.display.manager), 46
export_signal() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput method), 119
ext_trig_out (pymeasure.instruments.agilent.Agilent33500 attribute), 74
FakeAdapter (class in pymeasure.adapters), 28
field (pymeasure.instruments.ami.AMI430 attribute), 78
field (pymeasure.instruments.fwbell.FWBell5080 attribute), 87
field (pymeasure.instruments.lakeshore.LakeShore425 attribute), 116
fields() (pymeasure.instruments.fwbell.FWBell5080 method), 87
filter (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator attribute), 122
filter_slope (pymeasure.instruments.srs.SR830 attribute), 135
find_img_index() (pymeasure.display.curves.ResultsImage method), 44
FloatInput (class in pymeasure.display.inputs), 44
FloatParameter (class in pymeasure.experiment.parameters), 38
force_trigger() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
format() (pymeasure.experiment.results.CSVFormatter method), 41
format() (pymeasure.experiment.results.Results method), 41
freq_sweep() (pymeasure.instruments.agilent.AgilentE4980 method), 63
frequencies (pymeasure.instruments.agilent.AgilentE4980 attribute), 63
frequencies (pymeasure.instruments.agilent.AgilentE4408B attribute), 62
frequency (pymeasure.instruments.agilent.Agilent33500 attribute), 74
frequency (pymeasure.instruments.agilent.Agilent33500 get_buffer() (pymeasure.instruments.srs.SR830 attribute), 74)
frequency (pymeasure.instruments.agilent.Agilent33521A get_calibration_information() (pymeasure.instruments.agilent.Agilent4156 attribute), 66)
frequency (pymeasure.instruments.agilent.Agilent8257D get_data() (pymeasure.instruments.agilent.Agilent4156 attribute), 66)
frequency (pymeasure.instruments.ametek.Ametek7270 get_library_version() (pymeasure.instruments.agilent.Agilent4156 method), 129)
frequency (pymeasure.instruments.anritsu.AnritsuMG3692C get_procedure() (pymeasure.display.widgets.InputsWidget method), 91)
frequency (pymeasure.instruments.anritsu.AnritsuMG3692C get_time() (pymeasure.instruments.AgilentE4408B attribute), 106)
frequency (pymeasure.instruments.keithley.Keithley2000 get_scaling() (pymeasure.instruments.srs.SR830 attribute), 90)
frequency (pymeasure.instruments.signalrecovery.DSP7265 get_state_of_channels() (pymeasure.instruments.keithley.Keithley2700 attribute), 135)
frequency (pymeasure.instruments.srs.SR830 get_time() (pymeasure.instruments.Mock method), 55)
frequency (pymeasure.instruments.Keithley2000 get_voltage() (pymeasure.instruments.Mock method), 55)
frequency (pymeasure.instruments.Keithley2000 get_wave() (pymeasure.instruments.Mock method), 55)

H
handle_abort() (pymeasure.experiment.workers.Worker method), 40
handle_error() (pymeasure.experiment.workers.Worker method), 40
harmonic (pymeasure.instruments.ametek.Ametek7270 attribute), 77
harmonic (pymeasure.instruments.signalrecovery.DSP7265 attribute), 132
harmonic (pymeasure.instruments.srs.SR830 attribute), 135
has_amplitude_modulation (pymeasure.instruments.agilent.Agilent8257D attribute), 60
has_modulation (pymeasure.instruments.agilent.Agilent8257D attribute), 60
has_next() (pymeasure.display.manager.ExperimentQueue method), 46
has_persistent_switch_enabled() (pymeasure.instruments.ami.AMI430 method), 78
has_pulse_modulation (pymeasure.instruments.agilent.Agilent8257D attribute), 60
has_supported_version() (pymeasure.adapters.VISAAdapter static method), 32
header() (pymeasure.experiment.results.Results method), 41
heater_gas_mode (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 129
heater_range (pymeasure.instruments.lakeshore.LakeShore331 attribute), 115
hold_time (pymeasure.instruments.agilent.Agilent4156.Agilent4156 attribute), 66
home() (pymeasure.instruments.newport.esp300.Axis method), 117
HP33120A (class in pymeasure.instruments.hp), 87
HP34401A (class in pymeasure.instruments.hp), 88
id (pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG attribute), 58
id (pymeasure.instruments.agilent.Agilent33500 attribute), 74
id (pymeasure.instruments.ametek.Ametek7270 attribute), 77
id (pymeasure.instruments.danfysik.Danfysik8500 attribute), 83
id (pymeasure.instruments.fwbell.FWBell5080 attribute), 87
id (pymeasure.instruments.Instrument attribute), 54
id (pymeasure.instruments.keithley.Keithley2000 attribute), 91
id (pymeasure.instruments.keithley.Keithley2400 attribute), 96
id (pymeasure.instruments.keithley.Keithley2450 attribute), 102
id (pymeasure.instruments.keithley.Keithley2700 attribute), 107
id (pymeasure.instruments.keithley.Keithley2750 attribute), 113
id (pymeasure.instruments.keithley.Keithley6221 attribute), 109
id (pymeasure.instruments.keysight.KeysightN5767A attribute), 113
id (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133
id (pymeasure.instruments.yokogawa.Yokogawa7651 attribute), 138
ImageExperiment (class in pymeasure.display.manager), 47
ImageExperimentQueue (class in pymeasure.display.manager), 47
ImageFrame (class in pymeasure.display.widgets), 48
ImageManager (class in pymeasure.display.manager), 47
ImageWidget (class in pymeasure.display.widgets), 49
impedance (pymeasure.instruments.agilent.AgilentE4980 attribute), 63
input (class in pymeasure.display.inputs), 45
input_config (pymeasure.instruments.srs.SR830 attribute), 135
input_coupling (pymeasure.instruments.srs.SR830 attribute), 135
input_grounding (pymeasure.instruments.srs.SR830 attribute), 135
InputsWidget (class in pymeasure.display.widgets), 49
Instrument (class in pymeasure.instruments), 53
IntegerInput (class in pymeasure.display.inputs), 45
IntegerParameter (class in pymeasure.experiment.parameters), 38
integration_time (pymeasure.instruments.keithley.Keithley2000 attribute), 60
internal_frequency (pymeasure.instruments.agilent.Agilent8257D attribute), 60
internal_shape (pymeasure.instruments.agilent.Agilent8257D attribute), 60
is_averaging() (pymeasure.instruments.agilent.Agilent8722ES method), 61
is_buffer_full() (pymeasure.instruments.keithley.Keithley2000 method), 91
is_buffer_full() (pymeasure.instruments.keithley.Keithley2400 method), 96
is_buffer_full() (pymeasure.instruments.keithley.Keithley2450 method), 102
is_buffer_full() (pymeasure.instruments.keithley.Keithley2700 method), 107
is_buffer_full() (pymeasure.instruments.keithley.Keithley6221 method), 109
is_current_stable() (pymeasure.instruments.danfysik.Danfysik8500 method), 83
is_enabled (pymeasure.
PyMeasure Documentation, Release 0.8.0

sure.instruments.agilent.Agilent8257D

is_enabled() (pymeasure.instruments.danfysik.Danfysik8500 method), 60

is_enabled() (pymeasure.instruments.keysight.KeysightN5767A method), 113

is_moving() (pymeasure.instruments.parker.ParkerGV6 method), 131

is_out_of_range() (pymeasure.instruments.srs.SR830 method), 135

is_ready() (pymeasure.instruments.danfysik.Danfysik8500 method), 83

is_running() (pymeasure.display.manager.Manager method), 47

is_sequence_running() (pymeasure.instruments.danfysik.Danfysik8500 method), 83

is_set() (pymeasure.experiment.parameters.Parameter method), 39

ITC503 (class in pymeasure.instruments.oxfordinstruments), 129

J

join() (pymeasure.display.thread.StoppableQThread method), 48

join() (pymeasure.experiment.workers.Worker method), 40

K

Keithley2000 (class in pymeasure.instruments.keithley), 89

Keithley2400 (class in pymeasure.instruments.keithley), 94

Keithley2450 (class in pymeasure.instruments.keithley), 100

Keithley2700 (class in pymeasure.instruments.keithley), 105

Keithley2750 (class in pymeasure.instruments.keithley), 112

Keithley6221 (class in pymeasure.instruments.keithley), 108

KeysightN5767A (class in pymeasure.instruments.keysight), 113

kill() (pymeasure.instruments.parker.ParkerGV6 method), 131

L

labels() (pymeasure.experiment.results.Results method), 41

LakeShore331 (class in pymeasure.instruments.lakeshore), 115

LakeShore425 (class in pymeasure.instruments.lakeshore), 116

LakeShoreUSBAdapter (class in pymeasure.instruments.lakeshore), 114

LDCCurrent (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137

LDCCurrentLimit (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137

LDCPolarity (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137

LDCStatus (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137

left_limit (pymeasure.instruments.newport.exp300.Axis attribute), 117

level_lin (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81

level_log (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81

level_opt_attn (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81

level_scale (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81

list_resources() (in module pymeasure.instruments), 57

Listener (class in pymeasure.experiment.listeners), 36

ListInput (class in pymeasure.display.inputs), 45

ListParameter (class in pymeasure.experiment.parameters), 39

load() (pymeasure.display.manager.ImageManager method), 47

load() (pymeasure.display.manager.Manager method), 47

load() (pymeasure.experiment.results.Results static method), 41

load_sequence() (pymeasure.display.widgets.SequencerWidget method), 50

local() (pymeasure.instruments.danfysik.Danfysik8500 method), 83

local() (pymeasure.instruments.keithley.Keithley2000 method), 91

log_magnitude() (pymeasure.instruments.agilent.Agilent8722ES method), 61

LogHandler (class in pymeasure.display.log), 46

LogWidget (class in pymeasure.display.widgets), 49
PyMeasure Documentation, Release 0.8.0

raw_text

low_freq_out_amplitude (pymeasure.instruments.agilent.Agilent8257D attribute), 60

low_freq_out_source (pymeasure.instruments.agilent.Agilent8257D attribute), 60

M

mag (pymeasure.instruments.ametek.Ametek7270 attribute), 77

mag (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133

magnet_current (pymeasure.instruments.ami.AMI1430 attribute), 79

magnitude (pymeasure.instruments.srs.SR830 attribute), 135

magnitude() (pymeasure.instruments.agilent.Agilent8722ES method), 61

ManagedImageWindow (class in pymeasure.display.windows), 50

ManagedWindow (class in pymeasure.display.windows), 51

Manager (class in pymeasure.display.manager), 47

max_amplitude (pymeasure.instruments.hp.HP33120A attribute), 88

max_current (pymeasure.instruments.deltaelektronika.SM7045D attribute), 85

max_current (pymeasure.instruments.keithley.Keithley2400 attribute), 96

max_current (pymeasure.instruments.keithley.Keithley2450 attribute), 102

max_frequency (pymeasure.instruments.hp.HP33120A attribute), 88

max_offset (pymeasure.instruments.hp.HP33120A attribute), 88

max_resistance (pymeasure.instruments.keithley.Keithley2400 attribute), 96

max_resistance (pymeasure.instruments.keithley.Keithley2450 attribute), 102

max_voltage (pymeasure.instruments.deltaelektronika.SM7045D attribute), 85

max_voltage (pymeasure.instruments.keithley.Keithley2400 attribute), 102

max_voltage (pymeasure.instruments.keithley.Keithley2450 attribute), 102

max_voltage (pymeasure.instruments.keithley.Keithley2450 attribute), 102

mean_current (pymeasure.instruments.keithley.Keithley2400 attribute), 97

mean_current (pymeasure.instruments.keithley.Keithley2450 attribute), 102

mean_resistance (pymeasure.instruments.keithley.Keithley2400 attribute), 97

mean_resistance (pymeasure.instruments.keithley.Keithley2450 attribute), 102

mean_voltage (pymeasure.instruments.keithley.Keithley2400 attribute), 97

mean_voltage (pymeasure.instruments.keithley.Keithley2450 attribute), 102

means (pymeasure.instruments.keithley.Keithley2400 attribute), 97

means (pymeasure.instruments.keithley.Keithley2450 attribute), 102

Measurable (class in pymeasure.experiment.parameters), 39

measure() (pymeasure.instruments.agilent.Agilent4156.Agilent4156 method), 67

measure() (pymeasure.instruments.lakeshore.LakeShore425 method), 116

measure_concurrent_functions (pymeasure.instruments.keithley.Keithley2400 attribute), 97

measure_continuity() (pymeasure.instruments.keithley.Keithley2000 method), 91

measure_current (pymeasure.instruments.deltaelektronika.SM7045D attribute), 85

measure_current() (pymeasure.instruments.keithley.Keithley2000 method), 91

measure_current() (pymeasure.instruments.keithley.Keithley2400 method), 97

measure_current() (pymeasure.instruments.keithley.Keithley2450 method), 102

measure_diode() (pymeasure.
PyMeasure Documentation, Release 0.8.0

measure_frequency() (pymeasure.instruments.keithley.Keithley2000 method), 91
min_current (pymeasure.instruments.keithley.Keithley2400 attribute), 97
min_current (pymeasure.instruments.keithley.Keithley2450 attribute), 103
min_frequency (pymeasure.instruments.hp.HP33120A attribute), 88
min_offset (pymeasure.instruments.hp.HP33120A attribute), 88
min_resistance (pymeasure.instruments.keithley.Keithley2000 attribute), 97
min_resistance (pymeasure.instruments.keithley.Keithley2400 attribute), 97
min_resistance (pymeasure.instruments.keithley.Keithley2450 attribute), 103
min_voltage (pymeasure.instruments.keithley.Keithley2000 attribute), 97
minimums (pymeasure.instruments.keithley.Keithley2450 attribute), 103
Mock (class in pymeasure.instruments), 55
mode (pymeasure.instruments.agilent.AgilentE4980 attribute), 63
mode (pymeasure.instruments.keithley.Keithley2000 attribute), 92
Monitor (class in pymeasure.display.listeners), 46
Monitor (class in pymeasure.experiment.listeners), 37
motion_done (pymeasure.instruments.newport.esp300.Axis attribute), 117
mouseMoved() (pymeasure.display.curves.Crosshairs method), 44
move() (pymeasure.instruments.parker.ParkerGV6 method), 131

N
new_image() (pymeasure.display.widgets.ImageWidget method), 49
next() (pymeasure.display.manager.ExperimentQueue method), 46
next() (pymeasure.display.manager.Manager method), 47

O
offset (pymeasure.instruments.agilent.Agilent33220A attribute),
attribute (pymeasure.instruments.agilent.Agilent3500 attribute), 71
offset (pymeasure.instruments.agilent.Agilent4156.VARD attribute), 69
offset (pymeasure.instruments.hp.HP33120A attribute), 88
open() (pymeasure.instruments.keithley.Keithley2750 method), 113
open_all() (pymeasure.instruments.keithley.Keithley2750 method), 113
open_all_channels() (pymeasure.instruments.keithley.Keithley2700 method), 107
open_channels (pymeasure.instruments.keithley.Keithley2700 attribute), 107
open_rows_to_columns() (pymeasure.instruments.keithley.Keithley2700 method), 107
operation_event_enabled (pymeasure.instruments.keithley.Keithley6221 attribute), 109
operation_events (pymeasure.instruments.keithley.Keithley6221 attribute), 110
options (pymeasure.instruments.keithley.Keithley2700 attribute), 107
output (pymeasure.instruments.agilent.Agilent33220A attribute), 71
output (pymeasure.instruments.agilent.Agilent3500 attribute), 75
output (pymeasure.instruments.anritsu.AnritsuMG3692C attribute), 80
output_conversion() (pymeasure.instruments.srs.SR830 method), 135
output_load (pymeasure.instruments.agilent.Agilent33500 attribute), 75
output_trigger_on_external() (pymeasure.instruments.keithley.Keithley2400 method), 97
output_trigger_on_external() (pymeasure.instruments.keithley.Keithley6221 method), 110
outputs_enabled (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply attribute), 127

P

Parameter (class in pymeasure.experiment.parameters), 39

parameter (pymeasure.display.inputs.Input attribute), 45
parameter_objects() (pymeasure.experiment.procedure.Procedure method), 37
parameter_values() (pymeasure.experiment.procedure.Procedure method), 37
parameters_are_set() (pymeasure.experiment.procedure.Procedure method), 37
ParkerGV6 (class in pymeasure.instruments.parker), 131
parse() (pymeasure.experiment.results.Results method), 41
parse_axis() (pymeasure.display.widgets.ImageFrame method), 49
parse_axis() (pymeasure.display.widgets.PlotFrame method), 49
parse_header() (pymeasure.experiment.results.Results static method), 41
pause() (pymeasure.instruments.ami.AM1430 method), 79
pcolor() (pymeasure.experiment.experiment.Experiment method), 36
peak_search (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81
period (pymeasure.instruments.keithley.Keithley2000 attribute), 92
period_aperture (pymeasure.instruments.keithley.Keithley2000 attribute), 92
period_digits (pymeasure.instruments.keithley.Keithley2000 attribute), 92
period_reference (pymeasure.instruments.keithley.Keithley2000 attribute), 92
period_threshold (pymeasure.instruments.keithley.Keithley2000 attribute), 92
phase (pymeasure.instruments.ametek.Ametek7270 attribute), 77
phase (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133
phase (pymeasure.instruments.srs.SR830 attribute), 135
phase() (pymeasure.instruments.srs.SR830 attribute), 135
phase() (pymeasure.instruments.srs.SR830 method), 61
PhysicalParameter (class in pymeasure.experiment.parameters), 39
plot (pymeasure.display.windows.ManagedImageWindow
attribute), 50

plot (pymeasure.display.windows.ManagedWindow attribute), 51

plot() (pymeasure.experiment.experiment Experiment method), 36

plot_live() (pymeasure.experiment.experiment Experiment method), 36

PlotFrame (class in pymeasure.display.widgets), 49

Plotter (class in pymeasure.display.plotter), 48

PlotterWindow (class in pymeasure.display.windows), 52

PlotWidget (class in pymeasure.display.widgets), 49

points (pymeasure.instruments.agilent.agilent4156.VAR2 attribute), 69

polarity (pymeasure.instruments.danfysik.Danfysik8500 attribute), 83

position (pymeasure.experiment.newport.esp300.Axis attribute), 117

position (pymeasure.instruments.parker.ParkerGV6 attribute), 131

position_error (pymeasure.instruments.parker.ParkerGV6 attribute), 131

power (pymeasure.instruments.agilent.Agilent8257D attribute), 60

power (pymeasure.instruments.anritsu.AnritsuMG3692C attribute), 80

power (pymeasure.instruments.thorlabs.ThorlabsPM100USB attribute), 137

prepare() (pymeasure.display.curves.BufferCurve method), 44

Procedure (class in pymeasure.experiment.procedure), 37

program_sweep() (pymeasure.instruments.oxfordinstruments.ITC503 method), 129

PrologixAdapter (class in pymeasure.adapters), 30

pulse_dutycycle (pymeasure.instruments.agilent.Agilent33220A attribute), 71

pulse_dutycycle (pymeasure.instruments.agilent.Agilent33500 attribute), 75

pulse_frequency (pymeasure.instruments.agilent.Agilent8257D attribute), 60

pulse_hold (pymeasure.instruments.agilent.Agilent33220A attribute), 72

pulse_hold (pymeasure.instruments.agilent.Agilent33500 attribute), 75

pulse_input (pymeasure.instruments.agilent.Agilent8257D attribute), 60

pulse_period (pymeasure.instruments.agilent.Agilent33220A attribute), 72

pulse_period (pymeasure.instruments.agilent.Agilent33500 attribute), 75

pulse_source (pymeasure.instruments.agilent.Agilent8257D attribute), 60

pulse_transition (pymeasure.instruments.agilent.Agilent33220A attribute), 72

pulse_transition (pymeasure.instruments.agilent.Agilent33500 attribute), 75

pymeasure.display.browser (module), 43

pymeasure.display.curves (module), 43

pymeasure.display.inputs (module), 44

pymeasure.display.listeners (module), 46

pymeasure.display.log (module), 46

pymeasure.display.manager (module), 46

pymeasure.display.plotter (module), 48

pymeasure.display.thread (module), 48

pymeasure.display.windows (module), 50

pymeasure.experiment (module), 35

pymeasure.experiment.listeners (module), 36

pymeasure.experiment.parameters (module), 38

pymeasure.experiment.procedure (module), 37

pymeasure.experiment.results (module), 41

pymeasure.experiment.workers (module), 40

pymeasure.instruments (module), 52

pymeasure.instruments.advantest (module), 57

pymeasure.instruments.advantest.advantestR3767CG (module), 58

pymeasure.instruments.agilent (module), 58

pymeasure.instruments.agilent.agilent4156 (module), 64

pymeasure.instruments.ametek (module), 76

pymeasure.instruments.ami (module), 78

pymeasure.instruments.anritsu (module), 79
PyMeasure Documentation, Release 0.8.0

pymeasure.instruments.comedi (module), 57
pymeasure.instruments.danfysik (module), 82
pymeasure.instruments.deltaelektronika (module), 85
pymeasure.instruments.fwbell (module), 86
pymeasure.instruments.hp (module), 87
pymeasure.instruments.keithley (module), 113
pymeasure.instruments.lakeshore (module), 114
pymeasure.instruments.newport (module), 116
pymeasure.instruments.ni (module), 118
pymeasure.instruments.oxfordinstruments (module), 129
pymeasure.instruments.parker (module), 131
pymeasure.instruments.signalrecovery (module), 132
pymeasure.instruments.srs (module), 133
pymeasure.instruments.tektronix (module), 136
pymeasure.instruments.thorlabs (module), 136
pymeasure.instruments.validators (module), 55
pymeasure.instruments.yokogawa (module), 137

Q

QListener (class in pymeasure.display.listeners), 46
query_ac_current () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
query_acquisition_status () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_analog_channel () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_analog_channel_characteristics () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_analog_edge_trigger () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_analog_pulse_width_trigger () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_arbitrary_waveform () (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 122
query_arbitrary_waveform_gain_and_offset () (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 122
query_dc_current () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
query_dc_voltage () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
query_enabled_analog_channels () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_export_signal () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput method), 119
query_generation_status () (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 122
query_line_configuration () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput method), 119
query_measurement () (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
query_standard_waveform () (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
query_timing () (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 125
query_waveform_mode () (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
questionable_event_enabled (pymeasure.instruments.keithley.Keithley6221 attribute), 110
questionable_events (pymeasure.instruments.keithley.Keithley6221 attribute), 110
queue () (pymeasure.display.manager.Manager method), 47
queue () (pymeasure.display.windows.ManagedImageWindow method), 47

Index 177
sure.experiment.procedure.Procedure method), 37
reload() (pymeasure.experiment.results.Results method), 41
remote() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
remote() (pymeasure.instruments.keithley.Keithley2000 method), 92
remote_local_state (pymeasure.instruments.agilent.Agilent33220A attribute), 72
remote_lock() (pymeasure.instruments.keithley.Keithley2000 method), 92
remove() (pymeasure.display.manager.ImageManager method), 47
remove() (pymeasure.display.manager.Manager method), 47
reset() (pymeasure.instruments.fwbell.FWBell5080 method), 87
reset() (pymeasure.instruments.Instrument method), 54
reset() (pymeasure.instruments.keithley.Keithley2000 method), 92
reset() (pymeasure.instruments.keithley.Keithley2400 method), 98
reset() (pymeasure.instruments.keithley.Keithley2450 method), 103
reset() (pymeasure.instruments.keithley.Keithley2700 method), 107
reset() (pymeasure.instruments.keithley.Keithley2750 method), 113
reset() (pymeasure.instruments.keithley.Keithley6221 method), 110
reset() (pymeasure.instruments.keysight.KeysightN5767A method), 113
reset_buffer() (pymeasure.instruments.keithley.Keithley2000 method), 92
reset_buffer() (pymeasure.instruments.keithley.Keithley2400 method), 98
reset_buffer() (pymeasure.instruments.keithley.Keithley2450 method), 103
reset_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 107
reset_buffer() (pymeasure.instruments.keithley.Keithley6221 method), 110
reset_instrument() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput method), 120
reset_instrument() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
reset_instrument() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
reset_instrument() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 126
reset_instrument() (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply method), 127
reset_interlocks() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
reset_time() (pymeasure.instruments.Mock method), 55
resistance (pymeasure.instruments.agilent.Agilent34410A attribute), 64
resistance (pymeasure.instruments.hp.HP34401A attribute), 88
resistance (pymeasure.instruments.keithley.Keithley2000 attribute), 92
resistance (pymeasure.instruments.keithley.Keithley2400 attribute), 98
resistance (pymeasure.instruments.keithley.Keithley2450 attribute), 104
resistance (pymeasure.instruments.keithley.Keithley2700 attribute), 107
resistance (pymeasure.instruments.keithley.Keithley2750 attribute), 113
resistance (pymeasure.instruments.keithley.Keithley6221 attribute), 110
resistance_4w (pymeasure.instruments.agilent.Agilent34410A attribute), 64
resistance_4w (pymeasure.instruments.hp.HP34401A attribute), 88
resistance_4w_digits (pymeasure.instruments.keithley.Keithley2000 attribute), 92
resistance_4w_nplc (pymeasure.instruments.keithley.Keithley2000 attribute), 92
resistance_4w_range (pymeasure.instruments.keithley.Keithley2000 attribute), 92
resistance_4w_reference (pymeasure.instruments.keithley.Keithley2000 attribute), 92
resistance_digits (pymeasure.instruments.keithley.Keithley2000 attribute), 93
resistance_nplc (pymeasure.instruments.keithley.Keithley2400 attribute), 98
sample_frequency (pymeasure.instruments.srs.SR830 attribute), 135
sampling_points (pymeasure.instruments.anritsu.AnritsuMS9710C attribute), 81
save() (pymeasure.instruments.agilent.Agilent4156.Agilent4156 method), 67
save_var() (pymeasure.instruments.agilent.Agilent4156.Agilent4156 method), 67
scan() (pymeasure.instruments.agilent.Agilent8722ES method), 61
scan_continuous() (pymeasure.instruments.agilent.Agilent8722ES method), 61
scan(points) (pymeasure.instruments.agilent.Agilent8722ES attribute), 61
scan_single() (pymeasure.instruments.agilent.Agilent8722ES method), 61

ScientificInput (class in pymeasure.display.inputs), 45
self_calibrate() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
sensitivity (pymeasure.instruments.ametek.Ametek7270 attribute), 77
sensitivity (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133
sensitivity (pymeasure.instruments.srs.SR830 attribute), 135
dsensitivity (pymeasure.instruments.thorlabs.ThorlabsPM100USB method), 137

SequenceEvaluationException, 49
SequencerWidget (class in pymeasure.display.widgets), 49
SerialAdapter (class in pymeasure.adapters), 29
series_resistance (pymeasure.instruments.agilent.Agilent4156.SMU attribute), 68
set_averaging() (pymeasure.instruments.agilent.Agilent8722ES method), 61
set_channel_A_mode() (pymeasure.instruments.ametek.Ametek7270 method), 77
set_defaults() (pymeasure.adapters.PrologixAdapter method), 31
set_defaults() (pymea-
sure.instruments.parker.ParkerGV6
method), 131
set_differential_mode() (pyme-
sure.instruments.ametek.Ametek7270
method), 77
set_fixed_frequency() (pyme-
sure.instruments.agilent.Agilent8722ES
method), 61
set_hardware_limits() (pyme-
sure.instruments.parker.ParkerGV6
method), 132
set_IF_bandwidth() (pyme-
sure.instruments.agilent.Agilent8722ES
method), 61
set_output_voltage() (pyme-
sure.instruments.Mock method), 55
set_parameter() (pyme-
sure.display.inputs.BooleanInput
method), 44
set_parameter() (pyme-
sure.display.inputs.FloatInput
method), 45
set_parameter() (pyme-
sure.display.inputs.IntegerInput
method), 45
set_parameter() (pyme-
sure.display.inputs.ListInput
method), 45
set_parameter() (pyme-
sure.display.inputs.ScientificInput
method), 45
set_parameters() (pyme-
sure.display.windows.ManagedImageWindow
method), 50
set_parameters() (pyme-
sure.display.windows.ManagedWindow
method), 52
set_sequence() (pyme-
sure.instruments.danfysik.Danfysik8500
method), 84
set_ramp_delay() (pyme-
sure.instruments.danfysik.Danfysik8500
method), 84
set_ramp_to_current() (pyme-
sure.instruments.danfysik.Danfysik8500
method), 84
set_scaling() (pyme-
sure.instruments.srs.SR830
method), 136
set_sequence() (pyme-
sure.instruments.danfysik.Danfysik8500
method), 84
set_software_limits() (pyme-
sure.instruments.parker.ParkerGV6
method), 77
set_time() (pymeasure.instruments.Mock
method), 55
set_timed_arm() (pymeasure.instruments.keithley.Keithley2400
method), 98
set_timed_arm() (pymeasure.instruments.keithley.Keithley6221
method), 110
set_trigger_counts() (pymeasure.
instruments.keithley.Keithley2400
method), 98
set_voltage_mode() (pymeasure.
instruments.ametek.Ametek7270
method), 77
setDifferentialMode() (pymeasure.instruments.signalrecovery.DSP7265
method), 133
setpoint_1 (pymeasure.
instruments.lakeshore.LakeShore331
attribute), 115
setpoint_2 (pymeasure.
instruments.lakeshore.LakeShore331
attribute), 115
setting() (pymeasure.instruments.Instrument static
method), 54
setup_im_plot() (pymeasure.
display.windows.ManagedImageWindow
method), 51
setup_plot() (pymeasure.display.plotter.Plotter
method), 48
setup_plot() (pymeasure.
display.windows.ManagedImageWindow
method), 51
setup_plot() (pymeasure.
display.windows.ManagedWindow
method), 52
shape (pymeasure.instruments.agilent.Agilent33220A
attribute), 72
shape (pymeasure.instruments.agilent.Agilent33500
attribute), 75
shape (pymeasure.instruments.hp.HP33120A
attribute), 88
shutdown() (pymeasure.
experiment.procedure.Procedure method), 37
shutdown() (pymeasure.
experiment.workers.Worker
method), 40
shutdown() (pymeasure.
instruments.agilent.Agilent8257D
method), 60
shutdown() (pymeasure.
instruments.ametek.Ametek7270
method), 77
Index 181
shutdown() (pymeasure.instruments.ami.AMI430 method), 79
shutdown() (pymeasure.instruments.anritsu.AnritsuMG3692C method), 80
shutdown() (pymeasure.instruments.deltaelektronika.SM7045D method), 86
shutdown() (pymeasure.instruments.Instrument method), 55
shutdown() (pymeasure.instruments.keithley.Keithley2000 method), 93
shutdown() (pymeasure.instruments.keithley.Keithley2400 method), 98
shutdown() (pymeasure.instruments.keithley.Keithley2450 method), 107
shutdown() (pymeasure.instruments.keithley.Keithley2700 method), 104
shutdown() (pymeasure.instruments.keithley.Keithley2750 method), 113
shutdown() (pymeasure.instruments.keithley.Keithley6221 method), 110
shutdown() (pymeasure.instruments.keithley.Keithley6221 method), 110
shutdown() (pymeasure.instruments.keysight.KeysightN5767A method), 114
shutdown() (pymeasure.instruments.newport.ESP300 method), 117
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench method), 129
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalInput method), 120
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter method), 121
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 126
shutdown() (pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply method), 127
shutdown() (pymeasure.instruments.signalrecovery.DSP7265 method), 133
shutdown() (pymeasure.instruments.yokogawa.Yokogawa7651 method), 139
sine_voltage (pymeasure.instruments.srs.SR830 attribute), 136
single_sweep() (pymeasure.instruments.anritsu.AnritsuMS9710C method), 81
slew_rate (pymeasure.instruments.danfysik.Danfysik8500 attribute), 84
slope (pymeasure.instruments.ametek.Ametek7270 attribute), 77
slope (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133
slot (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137
SM7045D (class in pymeasure.instruments.deltaelektronika), 85
SMU (class in pymeasure.instruments.agilent.agilent4156), 67
source_auto_range (pymeasure.instruments.keithley.Keithley6221 attribute), 110
source_compliance (pymeasure.instruments.keithley.Keithley6221 attribute), 110
source_current (pymeasure.instruments.keithley.Keithley2400 attribute), 98
source_current (pymeasure.instruments.keithley.Keithley6221 attribute), 110
source_current (pymeasure.instruments.keithley.Keithley2450 attribute), 104
source_current (pymeasure.instruments.keithley.Keithley6221 attribute), 110
source_current (pymeasure.instruments.yokogawa.Yokogawa7651 attribute), 139
source_current_range (pymeasure.instruments.keithley.Keithley2400 attribute), 98
source_current_range (pymeasure.instruments.keithley.Keithley2450 attribute), 104
source_current_range (pymeasure.instruments.yokogawa.Yokogawa7651 attribute), 139
source_delay (pymeasure.instruments.keithley.Keithley2400 attribute), 99
source_delay (pymeasure.instruments.keithley.Keithley6221 attribute), 110
source_delay_auto (pymeasure.instruments.thorlabs.ThorlabsPro8000 attribute), 137

Index
sure.instruments.keithley.Keithley2400 attribute), 99
source_enabled (pymea-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
source_enabled (pyme-
sure.instruments.keithley.Keithley2450 at-
tribute), 104
source_enabled (pyme-
sure.instruments.keithley.Keithley6221 at-
tribute), 110
source_enabled (pyme-
sure.instruments.yokogawa.Yokogawa7651 attribute), 139
source_mode (pyme-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
source_mode (pyme-
sure.instruments.keithley.Keithley2450 at-
tribute), 104
source_mode (pyme-
sure.instruments.yokogawa.Yokogawa7651 attribute), 139
source_range (pyme-
sure.instruments.keithley.Keithley6221 at-
tribute), 110
source_voltage (pyme-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
source_voltage (pyme-
sure.instruments.keithley.Keithley2450 at-
tribute), 104
source_voltage (pyme-
sure.instruments.yokogawa.Yokogawa7651 attribute), 139
source_voltage_range (pyme-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
source_voltage_range (pyme-
sure.instruments.keithley.Keithley2450 at-
tribute), 104
source_voltage_range (pyme-
sure.instruments.yokogawa.Yokogawa7651 attribute), 139
spacing (pymeasure.instruments.agilent.agilent4156.VAR1 attribute), 68
span_frequency (pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG attribute), 58
square_dutycycle (pymeasure.instruments.agilent.Agilent33220A attribute), 72
square_dutycycle (pymeasure.instruments.agilent.Agilent33500 attribute), 75
SR830 (class in pymeasure.instruments.srs), 134
srq_event_enabled (pyme-
sure.instruments.keithley.Keithley6221 at-
tribute), 111
standard_devs (pyme-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
standard_devs (pyme-
sure.instruments.keithley.Keithley2450 at-
tribute), 104
standard_event_enabled (pyme-
sure.instruments.keithley.Keithley6221 at-
tribute), 111
start (pymeasure.instruments.agilent.agilent4156.VARX attribute), 69
start() (pymeasure.experiment.experiment.Experiment method), 36
start_buffer() (pymeasure.instruments.keithley.Keithley2000 method), 93
start_buffer() (pymeasure.instruments.keithley.Keithley2400 method), 99
start_buffer() (pymeasure.instruments.keithley.Keithley2450 method), 104
start_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 107
start_buffer() (pymeasure.instruments.keithley.Keithley6221 method), 111
start_frequency (pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG attribute), 58
start_frequency (pymeasure.instruments.agilent.Agilent8257D attribute), 60
start_frequency (pymeasure.instruments.agilent.Agilent8722ES attribute), 61
start_frequency (pymeasure.instruments.agilent.AgilentE4408B attribute), 62
start_power (pymeasure.instruments.agilent.Agilent8257D attribute), 60
start_ramp() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
start_sequence() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
start_step_sweep() (pymeasure.instruments.agilent.Agilent8257D method), 60
startup() (pymeasure.experiment.procedure.Procedure method), 38
startup() (pymeasure.experiment.procedure.UnknownProcedure method), 38
state (pymeasure.instruments.ami.AMI430 attribute), 79
status (pymeasure.instruments.danfysik.Danfysik8500 attribute), 84
status (pymeasure.instruments.parker.ParkerGV6 attribute), 132
status_hex (pymeasure.instruments.danfysik.Danfysik8500 attribute), 84
std_current (pymeasure.instruments.keithley.Keithley2400 attribute), 99
std_current (pymeasure.instruments.keithley.Keithley2450 attribute), 104
std_resistance (pymeasure.instruments.keithley.Keithley2400 attribute), 99
std_resistance (pymeasure.instruments.keithley.Keithley2450 attribute), 104
std_voltage (pymeasure.instruments.keithley.Keithley2400 attribute), 99
std_voltage (pymeasure.instruments.keithley.Keithley2450 attribute), 104
step (pymeasure.instruments.agilent.agilent4156.VARX attribute), 69
step_points (pymeasure.instruments.agilent.agilent8257D attribute), 69
stop() (pymeasure.instruments.agilent.agilent4156.VARX attribute), 69
stop() (pymeasure.instruments.agilent.agilent4156.Agilent4156 method), 67
stop() (pymeasure.instruments.ni.virtualbench.VirtualBench.FunctionGenerator method), 123
stop() (pymeasure.instruments.ni.virtualbench.VirtualBench.MixedSignalOscilloscope method), 126
stop() (pymeasure.instruments.parker.ParkerGV6 method), 132
stop_buffer() (pymeasure.instruments.keithley.Keithley2000 method), 93
stop_buffer() (pymeasure.instruments.keithley.Keithley2400 method), 99
stop_buffer() (pymeasure.instruments.keithley.Keithley2450 method), 104
stop_buffer() (pymeasure.instruments.keithley.Keithley2700 method), 107
stop_buffer() (pymeasure.instruments.advan.test.advan.testR3767CG.Advan.testR3767CG attribute), 58
stop_frequency (pymeasure.instruments.agilent.Agilent8257D attribute), 60
stop_frequency (pymeasure.instruments.agilent.Agilent8722ES attribute), 61
stop_frequency (pymeasure.instruments.agilent.AgilentE4408B attribute), 62
stop_power (pymeasure.instruments.agilent.Agilent8257D attribute), 61
stop_ramp() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
stop_sequence() (pymeasure.instruments.danfysik.Danfysik8500 method), 84
stop_step_sweep() (pymeasure.instruments.agilent.Agilent8257D method), 61
StoppableQThread (class in pymeasure.display.thread), 48
StringInput (class in pymeasure.display.inputs), 46
supply_current (pymeasure.instruments.ami.AMI430 attribute), 79
sweep_status (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 130
sweep_table (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 130
sweep_time (pymeasure.instruments.agilent.Agilent8722ES attribute), 62
sweep_time (pymeasure.instruments.agilent.AgilentE4408B attribute), 62
<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync_sequence()</td>
<td><code>pymeasure.instruments.danfysik.Danfysik8500</code> method</td>
<td>84</td>
</tr>
<tr>
<td>target_current</td>
<td><code>pymeasure.instruments.ami.AMI430</code> attribute</td>
<td>79</td>
</tr>
<tr>
<td>target_field</td>
<td><code>pymeasure.instruments.ami.AMI430</code> attribute</td>
<td>79</td>
</tr>
<tr>
<td>TDS2000</td>
<td><code>class in pymeasure.instruments.tektronix</code></td>
<td>136</td>
</tr>
<tr>
<td>TEDSetTemperature</td>
<td><code>pymeasure.instruments.thorlabs.ThorlabsPro8000</code> attribute</td>
<td>137</td>
</tr>
<tr>
<td>TEDStatus</td>
<td><code>pymeasure.instruments.thorlabs.ThorlabsPro8000</code> attribute</td>
<td>137</td>
</tr>
<tr>
<td>temperature</td>
<td><code>pymeasure.instruments.keithley.Keithley2000</code> attribute</td>
<td>93</td>
</tr>
<tr>
<td>temperature_1</td>
<td><code>pymeasure.instruments.oxfordinstruments.ITC503</code> attribute</td>
<td>130</td>
</tr>
<tr>
<td>temperature_2</td>
<td><code>pymeasure.instruments.oxfordinstruments.ITC503</code> attribute</td>
<td>130</td>
</tr>
<tr>
<td>temperature_3</td>
<td><code>pymeasure.instruments.oxfordinstruments.ITC503</code> attribute</td>
<td>130</td>
</tr>
<tr>
<td>temperature_A</td>
<td><code>pymeasure.instruments.lakeshore.LakeShore331</code> attribute</td>
<td>115</td>
</tr>
<tr>
<td>temperature_B</td>
<td><code>pymeasure.instruments.lakeshore.LakeShore331</code> attribute</td>
<td>115</td>
</tr>
<tr>
<td>temperature_digits</td>
<td><code>pymeasure.instruments.keithley.Keithley2000</code> attribute</td>
<td>93</td>
</tr>
<tr>
<td>temperature_error</td>
<td><code>pymeasure.instruments.oxfordinstruments.ITC503</code> attribute</td>
<td>130</td>
</tr>
<tr>
<td>temperature_nplc</td>
<td><code>pymeasure.instruments.keithley.Keithley2000</code> attribute</td>
<td>93</td>
</tr>
<tr>
<td>temperature_reference</td>
<td><code>pymeasure.instruments.keithley.Keithley2000</code> attribute</td>
<td>93</td>
</tr>
<tr>
<td>temperature_setpoint</td>
<td><code>pymeasure.instruments.oxfordinstruments.ITC503</code> attribute</td>
<td>130</td>
</tr>
<tr>
<td>text_enabled</td>
<td><code>pymeasure.instruments.keithley.Keithley2700</code> attribute</td>
<td>107</td>
</tr>
<tr>
<td>theta</td>
<td><code>pymeasure.instruments.srs.SR830</code> attribute</td>
<td>136</td>
</tr>
<tr>
<td>ThorlabsPM100USB</td>
<td><code>class in pymeasure.instruments.thorlabs</code></td>
<td>137</td>
</tr>
<tr>
<td>ThorlabsPro8000</td>
<td><code>class in pymeasure.instruments.thorlabs</code></td>
<td>137</td>
</tr>
<tr>
<td>time</td>
<td><code>pymeasure.instruments.Mock</code> attribute</td>
<td>55</td>
</tr>
<tr>
<td>time_constant</td>
<td><code>pymeasure.instruments.ametek.Ametek7270</code> attribute</td>
<td>77</td>
</tr>
<tr>
<td>time_constant</td>
<td><code>pymeasure.instruments.signalrecovery.DSP7265</code> attribute</td>
<td>133</td>
</tr>
<tr>
<td>time_constant</td>
<td><code>pymeasure.instruments.srs.SR830</code> attribute</td>
<td>136</td>
</tr>
<tr>
<td>trace()</td>
<td><code>pymeasure.instruments.agilent.AgilentE4408B</code> method</td>
<td>62</td>
</tr>
<tr>
<td>trace_df()</td>
<td><code>pymeasure.instruments.advantest.advantestR3767CG.AdvantestR3767CG</code> attribute</td>
<td>58</td>
</tr>
<tr>
<td>trace_marker</td>
<td><code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> attribute</td>
<td>81</td>
</tr>
<tr>
<td>trace_marker_center</td>
<td><code>pymeasure.instruments.anritsu.AnritsuMS9710C</code> attribute</td>
<td>81</td>
</tr>
<tr>
<td>tracking</td>
<td><code>pymeasure.instruments.ni.virtualbench.VirtualBench.PowerSupply</code> attribute</td>
<td>127</td>
</tr>
<tr>
<td>triad()</td>
<td><code>pymeasure.instruments.keithley.Keithley2400</code> method</td>
<td>99</td>
</tr>
<tr>
<td>triad()</td>
<td><code>pymeasure.instruments.keithley.Keithley2450</code> method</td>
<td>104</td>
</tr>
<tr>
<td>triad()</td>
<td><code>pymeasure.instruments.keithley.Keithley2700</code> method</td>
<td>107</td>
</tr>
<tr>
<td>triad()</td>
<td><code>pymeasure.instruments.keithley.Keithley6221</code> method</td>
<td>111</td>
</tr>
<tr>
<td>trigger()</td>
<td><code>pymeasure.instruments.agilent.Agilent33220A</code> method</td>
<td>72</td>
</tr>
<tr>
<td>trigger()</td>
<td><code>pymeasure.instruments.agilent.Agilent33500</code> method</td>
<td>75</td>
</tr>
<tr>
<td>trigger()</td>
<td><code>pymeasure.instruments.keithley.Keithley2400</code> method</td>
<td>99</td>
</tr>
<tr>
<td>trigger()</td>
<td><code>pymeasure.instruments.keithley.Keithley2450</code> method</td>
<td>105</td>
</tr>
<tr>
<td>trigger()</td>
<td><code>pymeasure.instruments.keithley.Keithley6221</code> method</td>
<td>111</td>
</tr>
<tr>
<td>trigger_count</td>
<td><code>pymeasure.instruments.keithley.Keithley2000</code> attribute</td>
<td>93</td>
</tr>
<tr>
<td>trigger_count</td>
<td><code>pymeasure.instruments.keithley.Keithley2400</code> attribute</td>
<td>99</td>
</tr>
<tr>
<td>trigger_delay</td>
<td><code>pymeasure.instruments.keithley.Keithley2400</code> attribute</td>
<td>99</td>
</tr>
</tbody>
</table>
sure.instruments.keithley.Keithley2000 at-
tribute), 93
trigger_delay (pymeas-
sure.instruments.keithley.Keithley2400 at-
tribute), 99
trigger_immediately() (pymeas-
sure.instruments.keithley.Keithley2400
method), 100
trigger_immediately() (pymeas-
sure.instruments.keithley.Keithley6221
method), 111
trigger_on_bus() (pymeas-
sure.instruments.keithley.Keithley2400
method), 100
trigger_on_bus() (pymeas-
sure.instruments.keithley.Keithley6221
method), 111
trigger_on_external() (pymeas-
sure.instruments.keithley.Keithley2400
method), 100
trigger_on_external() (pymeas-
sure.instruments.keithley.Keithley6221
method), 111
trigger_source (pymeas-
sure.instruments.agilent.Agilent33220A at-
tribute), 72
trigger_source (pymeas-
sure.instruments.agilent.Agilent33500 at-
tribute), 75
trigger_source (pymeas-
sure.instruments.agilent.AgilentE4980 at-
tribute), 63
trigger_state (pymeas-
sure.instruments.agilent.Agilent33220A at-
tribute), 72
tristate_lines() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput
method), 120

V
validate_auto_range_terminal() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter
method), 121
validate_channel() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter
method), 126
validate_channel() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.PowerSupply
method), 127
validate_dmm_function() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter
method), 121
validate_lines() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalInputOutput
method), 120
validate_range() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.DigitalMultimeter
static method), 121
validate_trigger_instance() (pymeas-
sure.instruments.ni.virtualbench.VirtualBench.MixedSignalOsci

U
unique_filename() (in module pymea-
sure.experiment.results), 41
unit (pymeasure.instruments.lakeshore.LakeShore425
attribute), 116
units (pymeasure.instruments.fwbell.FWBell5080
attribute), 87
units (pymeasure.instruments.newport.esp300.Axis
attribute), 118
UnknownProcedure (class in pymea-
sure.experiment.procedure), 38
update() (pymeasure.display.curves.Crosshairs
method), 44
update() (pymeasure.display.curves.ResultsCurve
method), 44
values() (pymeasure.adapters.Adapter method), 28
values() (pymeasure.adapters.FakeAdapter method), 28
values() (pymeasure.adapters.PrologixAdapter method), 31
values() (pymeasure.adapters.SerialAdapter method), 29
values() (pymeasure.adapters.VISAAdapter method), 32
values() (pymeasure.adapters.VXI11Adapter method), 33
values() (pymeasure.instruments.fwbell.FWBell5080 method), 87
values() (pymeasure.instruments.Instrument method), 55
values() (pymeasure.instruments.lakeshore.LakeShoreUSBAdapter method), 114
values() (pymeasure.instruments.signalrecovery.DSP7265 method), 133
VAR1 (class in pymeasure.instruments.agilent.agilent4156), 68
VAR2 (class in pymeasure.instruments.agilent.agilent4156), 69
VARD (class in pymeasure.instruments.agilent.agilent4156), 69
VARX (class in pymeasure.instruments.agilent.agilent4156), 69
VectorParameter (class in pymeasure.experiment.parameters), 40
VirtualBench (class in pymeasure.instruments.ni.virtualbench), 119
VirtualBench.DigitalInputOutput (class in pymeasure.instruments.ni.virtualbench), 119
VirtualBench.DigitalMultimeter (class in pymeasure.instruments.ni.virtualbench), 120
VirtualBench.FunctionGenerator (class in pymeasure.instruments.ni.virtualbench), 122
VirtualBench.MixedSignalOscilloscope (class in pymeasure.instruments.ni.virtualbench), 123
VirtualBench.PowerSupply (class in pymeasure.instruments.ni.virtualbench), 126
VirtualBench_Direct (class in pymeasure.instruments.ni.virtualbench), 129
VISAAdapter (class in pymeasure.adapters), 31
VMU (class in pymeasure.instruments.agilent.agilent4156), 70
voltage (pymeasure.instruments.keithley.Keithley2400 attribute), 100
time (pymeasure.instruments.keysight.KeysightN5767A attribute), 114
voltage (pymeasure.instruments.Mock attribute), 55
voltage (pymeasure.instruments.signalrecovery.DSP7265 attribute), 133
voltage_ac (pymeasure.instruments.agilent.Agilent34410A attribute), 64
voltage_ac (pymeasure.instruments.hp.HP34401A attribute), 88
voltage_ac_bandwidth (pymeasure.instruments.keithley.Keithley2000 attribute), 88
voltage_ac_digits (pymeasure.instruments.keithley.Keithley2000 attribute), 93
voltage_ac_nplc (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_ac_range (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_ac_reference (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_dc (pymeasure.instruments.agilent.Agilent34410A attribute), 64
voltage_dc (pymeasure.instruments.hp.HP34401A attribute), 88
voltage_digits (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_high (pymeasure.instruments.agilent.Agilent33220A attribute), 72
voltage_high (pymeasure.instruments.agilent.Agilent33500 attribute), 75
voltage_limit (pymeasure.instruments.ami.AMI430 attribute), 79
voltage_low (pymeasure.instruments.agilent.Agilent33220A attribute), 72
voltage_low (pymeasure.instruments.agilent.Agilent33500 attribute), 75
voltage_name (pymeasure.instruments.agilent.agilent4156.SMU attribute), 68
voltage_name : (pymeasure.instruments.agilent.agilent4156.VMU attribute), 70
voltage_name : (pymeasure.instruments.agilent.agilent4156.VSU attribute), 70
voltage_nplc : (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_nplc : (pymeasure.instruments.keithley.Keithley2400 attribute), 100
voltage_nplc : (pymeasure.instruments.keithley.Keithley2450 attribute), 105
voltage_range : (pymeasure.instruments.keithley.Keithley2000 attribute), 94
voltage_range : (pymeasure.instruments.keithley.Keithley2400 attribute), 100
voltage_range : (pymeasure.instruments.keithley.Keithley2450 attribute), 105
voltage_reference : (pymeasure.instruments.keithley.Keithley2000 attribute), 94
VSU : (class in pymeasure.instruments.agilent.agilent4156), 70
VXI11Adapter : (class in pymeasure.adapters), 33

W
wait() : (pymeasure.instruments.anritsu.AnritsuMS9710C method), 81
wait_for_buffer() : (pymeasure.instruments.keithley.Keithley2000 method), 94
wait_for_buffer() : (pymeasure.instruments.keithley.Keithley2400 method), 100
wait_for_buffer() : (pymeasure.instruments.keithley.Keithley2450 method), 105
wait_for_buffer() : (pymeasure.instruments.keithley.Keithley2700 method), 107
wait_for_buffer() : (pymeasure.instruments.keithley.Keithley6221 method), 111
wait_for_buffer() : (pymeasure.instruments.srs.SR830 method), 136
wait_for_current() : (pymeasure.instruments.danfysik.Danfysik8500 method), 84
wait_for_data() : (pymeasure.experiment.experiment.Experiment method), 36
wait_for_holding() : (pymeasure.instruments.ami.AMI430 method), 79
wait_for_ready() : (pymeasure.instruments.danfysik.Danfysik8500 method), 85
wait_for_srq() : (pymeasure.adapters.PrologixAdapter method), 31
wave() : (pymeasure.instruments.Mock attribute), 55
waveform_abort() : (pymeasure.instruments.keithley.Keithley6221 method), 111
waveform_amplitude : (pymeasure.instruments.keithley.Keithley6221 attribute), 111
waveform_arm() : (pymeasure.instruments.keithley.Keithley6221 method), 112
waveform_duration_cycles : (pymeasure.instruments.keithley.Keithley6221 attribute), 112
waveform_duration_set_infinity() : (pymeasure.instruments.keithley.Keithley6221 method), 112
waveform_duration_time : (pymeasure.instruments.keithley.Keithley6221 attribute), 112
waveform_dutycycle : (pymeasure.instruments.keithley.Keithley6221 attribute), 112
Yokogawa7651 (class in pymeasure.instruments.yokogawa), 138
ypointer (pymeasure.instruments.oxfordinstruments.ITC503 attribute), 130

Z

zero() (pymeasure.instruments.ami.AM1430 method), 79
zero() (pymeasure.instruments.newport.esp300.Axis method), 118
zero_probe() (pymeasure.instruments.lakeshore.LakeShore425 method), 116