InstrumentKit Library Documentation

Release 0.1.0

Steven Casagrande

June 09, 2016

Contents

Contents:

CHAPTER 1

Introduction

InstrumentKit allows for the control of scientific instruments in a platform-independent manner, abstracted from the details of how the instrument is connected. In particular, InstrumentKit supports connecting to instruments via serial port (including USB-based virtual serial connections), GPIB, USBTMC, TCP/IP or by using the VISA layer.

1.1 Installing

1.1.1 Dependencies

Most of the required and optional dependencies can be obtained using `pip`.

Required Dependencies

Using `pip`, these requirements can be obtained automatically by using the provided `requirements.txt`:

```
$ pip install -r requirements.txt
```

- NumPy
- PySerial
- quantities
- enum34
- future
- python-vxi11
- PyUSB
- python-usbtmc
- PyYAML

Optional Dependencies

- PyVISA (required for accessing instruments via VISA library)

1.2 Getting Started

1.2.1 Instruments and Instrument Classes

Each make and model of instrument that is supported by InstrumentKit is represented by a specific class, as documented in the *InstrumentKit API Reference*. Instruments that offer common functionality, such as multimeters, are represented by base classes, such that specific instruments can be exchanged without affecting code, so long as the proper functionality is provided.

For some instruments, a specific instrument class is not needed, as the *Generic SCPI Instruments* classes can be used to expose functionality of these instruments. If you don't see your specific instrument listed, then, please check in the instrument's manual whether it uses a standard set of SCPI commands.

1.2.2 Connecting to Instruments

Each instrument class in InstrumentKit is constructed using a *communicator* class that wraps a file-like object with additional information about newlines, terminators and other useful details. Most of the time, it is easiest to not worry with creating communicators directly, as convienence methods are provided to quickly connect to instruments over a wide range of common communication protocols and physical connections.

For instance, to connect to a generic SCPI-compliant multimeter using a Galvant Industries GPIB-USB adapter, the `open_gpibusb` method can be used:

```
>>> import instruments as ik
>>> inst = ik.generic_scpi.SCPIMultimeter.open_gpibusb("/dev/ttyUSB0", 1)
```

Similarly, many instruments connected by USB use an FTDI or similar chip to emulate serial ports, and can be connected using the `open_serial` method by specifying the serial port device file (on Linux) or name (on Windows) along with the baud rate of the emulated port:

```
>>> inst = ik.generic_scpi.SCPIMultimeter.open_serial("COM10", 115200)
```

As a convienence, an instrument connection can also be specified using a uniform resource identifier (URI) string:

```
>>> inst = ik.generic_scpi.SCPIMultimeter.open_from_uri("tcpip://192.168.0.10:4100:)
```

Instrument connection URIs of this kind are useful for storing in configuration files, as the same method, `open_from_uri`, is used, regardless of the communication protocol and physical connection being used. InstrumentKit provides special support for this usage, and can load instruments from specifications listed in a YAML-formatted configuration file. See the `load_instruments` function for more details.

1.2.3 Using Connected Instruments

Once connected, functionality of each instrument is exposed by methods and properties of the instrument object. For instance, the name of an instrument can be queried by getting the `name` property:

```
>>> print(inst.name)
```

For details of how to use each instrument, please see the *InstrumentKit API Reference* entry for that instrument's class. If that class does not implement a given command, raw commands and queries can be issued by using the `sendcmd` and `query` methods, respectively:

```
>>> inst.sendcmd("DATA") # Send command with no response
>>> resp = inst.query("*IDN?") # Send command and retrieve response
```

1.3 OS-Specific Instructions

1.3.1 Linux

Raw USB Device Configuration

To enable writing to a USB device in raw or usbtmc mode, the device file must be readable writable by users. As this is not normally the default, you need to add rules to `/etc/udev/rules.d` to override the default permissions. For instance, to add a Tektronix DPO 4104 oscilloscope with world-writable permissions, add the following to rules.d:

```
ATTRS{idVendor}=="0699", ATTRS{idProduct}=="0401", SYMLINK+="tekdpo4104", MODE="0666"
```

Warning: This configuration causes the USB device to be world-writable. Do not do this on a multi-user system with untrusted users.

CHAPTER 2

InstrumentKit API Reference

Contents:

2.1 Instrument Base Classes

2.1.1 `Instrument` - Base class for instrument communication

class `instruments.`**`Instrument`**(*filelike*)

This is the base instrument class from which all others are derived from. It provides the basic implementation for all communication related tasks. In addition, it also contains several class methods for opening connections via the supported hardware channels.

`binblockread`(*data_width*, *fmt=None*)

" Read a binary data block from attached instrument. This requires that the instrument respond in a particular manner as EOL terminators naturally can not be used in binary transfers.

The format is as follows: #{number of following digits:1-9}{num of bytes to be read}{data bytes}

Parameters

- **`data_width`** (`int`) – Specify the number of bytes wide each data point is. One of [1,2,4].
- **`fmt`** (*`str`*) – Format string as specified by the `struct` module, or `None` to choose a format automatically based on the data width. Typically you can just specify `data_width` and leave this default.

classmethod **`open_file`**(*filename*)

Given a file, treats that file as a character device file that can be read from and written to in order to communicate with the instrument. This may be the case, for instance, if the instrument is connected by the Linux `usbtmc` kernel driver.

Parameters **`filename`** (*`str`*) – Name of the character device to open.

Return type *`Instrument`*

Returns Object representing the connected instrument.

classmethod **`open_from_uri`**(*uri*)

Given an instrument URI, opens the instrument named by that URI. Instrument URIs are formatted with a scheme, such as `serial://`, followed by a location that is interpreted differently for each scheme. The following examples URIs demonstrate the currently supported schemes and location formats:

```
serial://COM3
serial:///dev/ttyACM0
tcpip://192.168.0.10:4100
gpib+usb://COM3/15
gpib+serial://COM3/15
gpib+serial:///dev/ttyACM0/15 # Currently non-functional.
visa://USB::0x0699::0x0401::C0000001::0::INSTR
usbtmc://USB::0x0699::0x0401::C0000001::0::INSTR
```

For the `serial` URI scheme, baud rates may be explicitly specified using the query parameter `baud=`, as in the example `serial://COM9?baud=115200`. If not specified, the baud rate is assumed to be 115200.

Parameters **uri** (*str*) – URI for the instrument to be loaded.

Return type *Instrument*

See also:

PySerial documentation for serial port URI format

classmethod open_gpibethernet (*host*, *port*, *gpib_address*)

> **Warning:** The GPIB-Ethernet adapter that this connection would use does not actually exist, and thus this class method should not be used.

classmethod open_gpibusb (*port*, *gpib_address*, *timeout=3*, *write_timeout=3*)

Opens an instrument, connecting via a Galvant Industries GPIB-USB adapter.

Parameters

- **port** (*str*) – Name of the the port or device file to open a connection on. Note that because the GI GPIB-USB adapter identifies as a serial port to the operating system, this should be the name of a serial port.
- **gpib_address** (int) – Address on the connected GPIB bus assigned to the instrument.
- **timeout** (float) – Number of seconds to wait when reading from the instrument before timing out.
- **write_timeout** (float) – Number of seconds to wait when writing to the instrument before timing out.

Return type *Instrument*

Returns Object representing the connected instrument.

See also:

`Serial` for description of `port` and timeouts.

classmethod open_serial (*port*, *baud*, *timeout=3*, *write_timeout=3*)

Opens an instrument, connecting via a physical or emulated serial port. Note that many instruments which connect via USB are exposed to the operating system as serial ports, so this method will very commonly be used for connecting instruments via USB.

Parameters

- **port** (*str*) – Name of the the port or device file to open a connection on. For example, `"COM10"` on Windows or `"/dev/ttyUSB0"` on Linux.
- **baud** (int) – The baud rate at which instrument communicates.

- **timeout** (float) – Number of seconds to wait when reading from the instrument before timing out.
- **write_timeout** (float) – Number of seconds to wait when writing to the instrument before timing out.

Return type *Instrument*

Returns Object representing the connected instrument.

See also:

Serial for description of port, baud rates and timeouts.

classmethod open_tcpip (*host*, *port*)

Opens an instrument, connecting via TCP/IP to a given host and TCP port.

Parameters

- **host** (*str*) – Name or IP address of the instrument.
- **port** (int) – TCP port on which the insturment is listening.

Return type *Instrument*

Returns Object representing the connected instrument.

See also:

connect for description of host and port parameters in the TCP/IP address family.

classmethod open_test (*stdin=None*, *stdout=None*)

Opens an instrument using a loopback communicator for a test connection. The primary use case of this is to instantiate a specific instrument class without requiring an actual physical connection of any kind. This is also very useful for creating unit tests through the parameters of this class method.

Parameters

- **stdin** (io.BytesIO or None) – The stream of data coming from the instrument
- **stdout** (io.BytesIO or None) – Empty data stream that will hold data sent from the Python class to the loopback communicator. This can then be checked for the contents.

Returns Object representing the virtually-connected instrument

classmethod open_usb (*vid*, *pid*)

Opens an instrument, connecting via a raw USB stream.

Note: Note that raw USB a very uncommon of connecting to instruments, even for those that are connected by USB. Most will identify as either serial ports (in which case, *open_serial* should be used), or as USB-TMC devices. On Linux, USB-TMC devices can be connected using *open_file*, provided that the usbtmc kernel module is loaded. On Windows, some such devices can be opened using the VISA library and the *open_visa* method.

Parameters

- **vid** (*str*) – Vendor ID of the USB device to open.
- **pid** (int) – Product ID of the USB device to open.

Return type *Instrument*

Returns Object representing the connected instrument.

classmethod **open_usbtmc** (**args*, ***kwargs*)

Opens an instrument, connecting to a USB-TMC device using the Python `usbtmc` library.

> **Warning:** The operational status of this is unknown. It is suggested that you connect via the other provided class methods. For Linux, if you have the `usbtmc` kernel module, the *`open_file`* class method will work. On Windows, using the *`open_visa`* class method along with having the VISA libraries installed will work.

Returns Object representing the connected instrument

classmethod **open_visa** (*resource_name*)

Opens an instrument, connecting using the VISA library. Note that PyVISA and a VISA implementation must both be present and installed for this method to function.

Parameters **resource_name** (*str*) – Name of a VISA resource representing the given instrument.

Return type *Instrument*

Returns Object representing the connected instrument.

See also:

National Instruments help page on VISA resource names.

classmethod **open_vxi11** (**args*, ***kwargs*)

Opens a vxi11 enabled instrument, connecting using the python library python-vxi11. This package must be present and installed for this method to function.

Return type *Instrument*

Returns Object representing the connected instrument.

query (*cmd*, *size=-1*)

Executes the given query.

Parameters

- **cmd** (*str*) – String containing the query to execute.
- **size** (`int`) – Number of bytes to be read. Default is read until termination character is found.

Returns The result of the query as returned by the connected instrument.

Return type `str`

read (*size=-1*)

Read the last line.

Parameters **size** (`int`) – Number of bytes to be read. Default is read until termination character is found.

Returns The result of the read as returned by the connected instrument.

Return type `str`

sendcmd (*cmd*)

Sends a command without waiting for a response.

Parameters **cmd** (*str*) – String containing the command to be sent.

write (*msg*)

Write data string to the connected instrument. This will call the write method for the attached filelike object. This will typically bypass attaching any termination characters or other communication channel related work.

See also:

Instrument.sendcmd if you wish to send a string to the

instrument, while still having InstrumentKit handle termination characters and other communication channel related work.

Parameters msg (*str*) – String that will be written to the filelike object (`Instrument._file`) attached to this instrument.

URI_SCHEMES = [u'serial', u'tcpip', u'gpib+usb', u'gpib+serial', u'visa', u'file', u'usbtmc', u'vxi11']

address

Gets/sets the target communication of the instrument.

This is useful for situations when running straight from a Python shell and your instrument has enumerated with a different address. An example when this can happen is if you are using a USB to Serial adapter and you disconnect/reconnect it.

Type `int` for GPIB address, `str` for other

prompt

Gets/sets the prompt used for communication.

The prompt refers to a character that is sent back from the instrument after it has finished processing your last command. Typically this is used to indicate to an end-user that the device is ready for input when connected to a serial-terminal interface.

In IK, the prompt is specified that that it (and its associated termination character) are read in. The value read in from the device is also checked against the stored prompt value to make sure that everything is still in sync.

Type `str`

terminator

Gets/sets the terminator used for communication.

For communication options where this is applicable, the value corresponds to the ASCII character used for termination in decimal format. Example: 10 sets the character to NEWLINE.

Type `int`, or `str` for GPIB adapters.

timeout

Gets/sets the communication timeout for this instrument. Note that setting this value after opening the connection is not supported for all connection types.

Type `int`

2.1.2 `Multimeter` - Abstract class for multimeter instruments

class `instruments.abstract_instruments.`**`Multimeter`**(*filelike*)

Abstract base class for multimeter instruments.

All applicable concrete instruments should inherit from this ABC to provide a consistent interface to the user.

measure(*mode*)

Perform a measurement as specified by mode parameter.

input_range

Gets/sets the current input range setting of the multimeter. This is an abstract method.

Type `Quantity` or `Enum`

mode
Gets/sets the measurement mode for the multimeter. This is an abstract method.

Type `Enum`

relative
Gets/sets the status of relative measuring mode for the multimeter. This is an abstract method.

Type `bool`

trigger_mode
Gets/sets the trigger mode for the multimeter. This is an abstract method.

Type `Enum`

2.1.3 FunctionGenerator - Abstract class for function generator instruments

class `instruments.abstract_instruments.`**FunctionGenerator**(*filelike*)
Abstract base class for function generator instruments.

All applicable concrete instruments should inherit from this ABC to provide a consistent interface to the user.

class Function
Enum containg valid output function modes for many function generators

arbitrary = <Function.arbitrary: 'ARB'>

noise = <Function.noise: 'NOIS'>

ramp = <Function.ramp: 'RAMP'>

sinusoid = <Function.sinusoid: 'SIN'>

square = <Function.square: 'SQU'>

triangle = <Function.triangle: 'TRI'>

class `FunctionGenerator.`**VoltageMode**
Enum containing valid voltage modes for many function generators

dBm = <VoltageMode.dBm: 'DBM'>

peak_to_peak = <VoltageMode.peak_to_peak: 'VPP'>

rms = <VoltageMode.rms: 'VRMS'>

`FunctionGenerator.`**amplitude**
Gets/sets the output amplitude of the function generator.

If set with units of dBm, then no voltage mode can be passed.

If set with units of V as a `Quantity` or a `float` without a voltage mode, then the voltage mode is assumed to be peak-to-peak.

Units As specified, or assumed to be V if not specified.

Type Either a `tuple` of a `Quantity` and a `FunctionGenerator.VoltageMode`, or a `Quantity` if no voltage mode applies.

`FunctionGenerator.`**frequency**
Gets/sets the the output frequency of the function generator. This is an abstract property.

Type `Quantity`

FunctionGenerator.**function**
Gets/sets the output function mode of the function generator. This is an abstract property.

Type Enum

FunctionGenerator.**offset**
Gets/sets the output offset voltage of the function generator. This is an abstract property.

Type Quantity

FunctionGenerator.**phase**
Gets/sets the output phase of the function generator. This is an abstract property.

Type Quantity

2.1.4 SignalGenerator - Abstract class for Signal Generators

class instruments.abstract_instruments.signal_generator.**SignalGenerator**(*filelike*)
Python abstract base class for signal generators (eg microwave sources).

This ABC is not for function generators, which have their own separate ABC.

See also:

FunctionGenerator

channel
Gets a specific channel object for the SignalGenerator.

Return type A class inherited from SGChannel

2.1.5 SingleChannelSG - Class for Signal Generators with a Single Channel

class instruments.abstract_instruments.signal_generator.**SingleChannelSG**(*filelike*)
Class for representing a Signal Generator that only has a single output channel. The sole property in this class allows for the user to use the API for SGs with multiple channels and a more compact form since it only has one output.

For example, both of the following calls would work the same:

```
>>> print sg.channel[0].freq # Multi-channel style
>>> print sg.freq # Compact style
```

channel

2.1.6 SGChannel - Abstract class for Signal Generator Channels

class instruments.abstract_instruments.signal_generator.**SGChannel**
Python abstract base class representing a single channel for a signal generator.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the SignalGenerator class.

frequency
Gets/sets the output frequency of the signal generator channel

Type Quantity

output
Gets/sets the output status of the signal generator channel

Type bool

phase
Gets/sets the output phase of the signal generator channel

Type Quantity

power
Gets/sets the output power of the signal generator channel

Type Quantity

2.2 Generic SCPI Instruments

2.2.1 SCPIInstrument - Base class for instruments using the SCPI protocol

class instruments.generic_scpi.**SCPIInstrument**(*filelike*)
Base class for all SCPI-compliant instruments. Inherits from from *Instrument*.

This class does not implement any instrument-specific communication commands. What it does add is several of the generic SCPI star commands. This includes commands such as *IDN?, *OPC?, and *RST.

Example usage:

```
>>> import instruments as ik
>>> inst = ik.generic_scpi.SCPIInstrument.open_tcpip('192.168.0.2', 8888)
>>> print(inst.name)
```

class ErrorCodes
Enumeration describing error codes as defined by SCPI 1999.0. Error codes that are equal to 0 mod 100 are defined to be *generic*.

block_data_error = <ErrorCodes.block_data_error: -160>

block_data_not_allowed = <ErrorCodes.block_data_not_allowed: -168>

character_data_error = <ErrorCodes.character_data_error: -140>

character_data_not_allowed = <ErrorCodes.character_data_not_allowed: -148>

character_data_too_long = <ErrorCodes.character_data_too_long: -144>

command_error = <ErrorCodes.command_error: -100>

command_header_error = <ErrorCodes.command_header_error: -110>

data_type_error = <ErrorCodes.data_type_error: -104>

exponent_too_large = <ErrorCodes.exponent_too_large: -123>

expression_error = <ErrorCodes.expression_error: -170>

expression_not_allowed = <ErrorCodes.expression_not_allowed: -178>

get_not_allowed = <ErrorCodes.get_not_allowed: -105>

header_separator_error = <ErrorCodes.header_separator_error: -111>

header_suffix_out_of_range = <ErrorCodes.header_suffix_out_of_range: -114>

invalid_block_data = <ErrorCodes.invalid_block_data: -161>

invalid_character = <ErrorCodes.invalid_character: -101>

invalid_character_data = <ErrorCodes.invalid_character_data: -141>

invalid_character_in_number = <ErrorCodes.invalid_character_in_number: -121>

invalid_expression = <ErrorCodes.invalid_expression: -171>

invalid_inside_macro_definition = <ErrorCodes.invalid_inside_macro_definition: -183>

invalid_outside_macro_definition = <ErrorCodes.invalid_outside_macro_definition: -181>

invalid_separator = <ErrorCodes.invalid_separator: -103>

invalid_string_data = <ErrorCodes.invalid_string_data: -151>

invalid_suffix = <ErrorCodes.invalid_suffix: -131>

macro_error = <ErrorCodes.macro_error: -180>

macro_parameter_error = <ErrorCodes.macro_parameter_error: -184>

missing_parameter = <ErrorCodes.missing_parameter: -109>

no_error = <ErrorCodes.no_error: 0>

numeric_data_error = <ErrorCodes.numeric_data_error: -120>

numeric_data_not_allowed = <ErrorCodes.numeric_data_not_allowed: -128>

operation_complete = <ErrorCodes.operation_complete: -800>

parameter_not_allowed = <ErrorCodes.parameter_not_allowed: -108>

power_on = <ErrorCodes.power_on: -500>

program_mnemonic_too_long = <ErrorCodes.program_mnemonic_too_long: -112>

request_control_event = <ErrorCodes.request_control_event: -700>

string_data_error = <ErrorCodes.string_data_error: -150>

string_data_not_allowed = <ErrorCodes.string_data_not_allowed: -158>

suffix_error = <ErrorCodes.suffix_error: -130>

suffix_not_allowed = <ErrorCodes.suffix_not_allowed: -138>

suffix_too_long = <ErrorCodes.suffix_too_long: -134>

syntax_error = <ErrorCodes.syntax_error: -102>

too_many_digits = <ErrorCodes.too_many_digits: -124>

undefined_header = <ErrorCodes.undefined_header: -113>

unexpected_number_of_parameters = <ErrorCodes.unexpected_number_of_parameters: -115>

user_request_event = <ErrorCodes.user_request_event: -600>

SCPIInstrument.**check_error_queue**()

Checks and clears the error queue for this device, returning a list of *SCPIInstrument.ErrorCodes* or int elements for each error reported by the connected instrument.

SCPIInstrument.**clear**()

Clear instrument. Consult manual for specifics related to that instrument.

SCPIInstrument.**reset**()

Reset instrument. On many instruments this is a factory reset and will revert all settings to default.

SCPIInstrument.**trigger**()
Send a software trigger event to the instrument. On most instruments this will cause some sort of hardware event to start. For example, a multimeter might take a measurement.

This software trigger usually performs the same action as a hardware trigger to your instrument.

SCPIInstrument.**wait_to_continue**()
Instruct the instrument to wait until it has completed all received commands before continuing.

SCPIInstrument.**display_brightness**
Brightness of the display on the connected instrument, represented as a float ranging from 0 (dark) to 1 (full brightness).

Type float

SCPIInstrument.**display_contrast**
Contrast of the display on the connected instrument, represented as a float ranging from 0 (no contrast) to 1 (full contrast).

Type float

SCPIInstrument.**line_frequency**
Gets/sets the power line frequency setting for the instrument.

Returns The power line frequency

Units Hertz

Type Quantity

SCPIInstrument.**name**
The name of the connected instrument, as reported by the standard SCPI command `*IDN?`.

Return type str

SCPIInstrument.**op_complete**
Check if all operations sent to the instrument have been completed.

Return type bool

SCPIInstrument.**power_on_status**
Gets/sets the power on status for the instrument.

Type bool

SCPIInstrument.**scpi_version**
Returns the version of the SCPI protocol supported by this instrument, as specified by the `SYST:VERS?` command described in section 21.21 of the SCPI 1999 standard.

SCPIInstrument.**self_test_ok**
Gets the results of the instrument's self test. This lets you check if the self test was sucessful or not.

Return type bool

2.2.2 SCPIMultimeter - Generic multimeter using SCPI commands

class instruments.generic_scpi.**SCPIMultimeter**(*filelike*)
This class is used for communicating with generic SCPI-compliant multimeters.

Example usage:

```
>>> import instruments as ik
>>> inst = ik.generic_scpi.SCPIMultimeter.open_tcpip("192.168.1.1")
>>> print(inst.measure(inst.Mode.resistance))
```

class `InputRange`
Valid device range parameters outside of directly specifying the range.

`automatic` = **<InputRange.automatic: 'AUTO'>**

`default` = **<InputRange.default: 'DEF'>**

`maximum` = **<InputRange.maximum: 'MAX'>**

`minimum` = **<InputRange.minimum: 'MIN'>**

class `SCPIMultimeter.Mode`
Enum of valid measurement modes for (most) SCPI compliant multimeters

`capacitance` = **<Mode.capacitance: 'CAP'>**

`continuity` = **<Mode.continuity: 'CONT'>**

`current_ac` = **<Mode.current_ac: 'CURR:AC'>**

`current_dc` = **<Mode.current_dc: 'CURR:DC'>**

`diode` = **<Mode.diode: 'DIOD'>**

`fourpt_resistance` = **<Mode.fourpt_resistance: 'FRES'>**

`frequency` = **<Mode.frequency: 'FREQ'>**

`period` = **<Mode.period: 'PER'>**

`resistance` = **<Mode.resistance: 'RES'>**

`temperature` = **<Mode.temperature: 'TEMP'>**

`voltage_ac` = **<Mode.voltage_ac: 'VOLT:AC'>**

`voltage_dc` = **<Mode.voltage_dc: 'VOLT:DC'>**

class `SCPIMultimeter.Resolution`
Valid measurement resolution parameters outside of directly the resolution.

`default` = **<Resolution.default: 'DEF'>**

`maximum` = **<Resolution.maximum: 'MAX'>**

`minimum` = **<Resolution.minimum: 'MIN'>**

class `SCPIMultimeter.SampleCount`
Valid sample count parameters outside of directly the value.

`default` = **<SampleCount.default: 'DEF'>**

`maximum` = **<SampleCount.maximum: 'MAX'>**

`minimum` = **<SampleCount.minimum: 'MIN'>**

class `SCPIMultimeter.SampleSource`
Valid sample source parameters.

1. **"immediate": The trigger delay time is inserted between successive** samples. After the first measurement is completed, the instrument waits the time specified by the trigger delay and then performs the next sample.
2. **"timer": Successive samples start one sample interval after the** START of the previous sample.

immediate = <SampleSource.immediate: 'IMM'>

timer = <SampleSource.timer: 'TIM'>

class SCPIMultimeter.**TriggerCount**
Valid trigger count parameters outside of directly the value.

default = <TriggerCount.default: 'DEF'>

infinity = <TriggerCount.infinity: 'INF'>

maximum = <TriggerCount.maximum: 'MAX'>

minimum = <TriggerCount.minimum: 'MIN'>

class SCPIMultimeter.**TriggerMode**
Valid trigger sources for most SCPI Multimeters.

"Immediate": This is a continuous trigger. This means the trigger signal is always present.

"External": External TTL pulse on the back of the instrument. It is active low.

"Bus": Causes the instrument to trigger when a `*TRG` command is sent by software. This means calling the trigger() function.

bus = <TriggerMode.bus: 'BUS'>

external = <TriggerMode.external: 'EXT'>

immediate = <TriggerMode.immediate: 'IMM'>

SCPIMultimeter.**measure**(*mode=None*)
Instruct the multimeter to perform a one time measurement. The instrument will use default parameters for the requested measurement. The measurement will immediately take place, and the results are directly sent to the instrument's output buffer.

Method returns a Python quantity consisting of a numpy array with the instrument value and appropriate units. If no appropriate units exist, (for example, continuity), then return type is `float`.

Parameters mode (*Mode*) – Desired measurement mode. If set to `None`, will default to the current mode.

SCPIMultimeter.**input_range**
Gets/sets the device input range for the device range for the currently set multimeter mode.

Example usages:

```
>>> dmm.input_range = dmm.InputRange.automatic
>>> dmm.input_range = 1 * pq.millivolt
```

Units As appropriate for the current mode setting.

Type `Quantity`, or *InputRange*

SCPIMultimeter.**mode**
Gets/sets the current measurement mode for the multimeter.

Example usage:

```
>>> dmm.mode = dmm.Mode.voltage_dc
```

Type *Mode*

SCPIMultimeter.**relative**

SCPIMultimeter.**resolution**

Gets/sets the measurement resolution for the multimeter. When specified as a float it is assumed that the user is providing an appropriate value.

Example usage:

```
>>> dmm.resolution = 3e-06
>>> dmm.resolution = dmm.Resolution.maximum
```

Type int, float or *Resolution*

SCPIMultimeter.**sample_count**

Gets/sets the number of readings (samples) that the multimeter will take per trigger event.

The time between each measurement is defined with the sample_timer property.

Note that if the trigger_count propery has been changed, the number of readings taken total will be a multiplication of sample count and trigger count (see property SCPIMulimeter.trigger_count).

If specified as a *SampleCount* value, the following options apply:

1. "minimum": 1 sample per trigger
2. "maximum": Maximum value as per instrument manual
3. "default": Instrument default as per instrument manual

Note that when using triggered measurements, it is recommended that you disable autorange by either explicitly disabling it or specifying your desired range.

Type int or *SampleCount*

SCPIMultimeter.**sample_source**

Gets/sets the multimeter sample source. This determines whether the trigger delay or the sample timer is used to dtermine sample timing when the sample count is greater than 1.

In both cases, the first sample is taken one trigger delay time period after the trigger event. After that, it depends on which mode is used.

Type *SCPIMultimeter.SampleSource*

SCPIMultimeter.**sample_timer**

Gets/sets the sample interval when the sample counter is greater than one and when the sample source is set to timer (see *SCPIMultimeter.sample_source*).

This command does not effect the delay between the trigger occuring and the start of the first sample. This trigger delay is set with the *trigger_delay* property.

Units As specified, or assumed to be of units seconds otherwise.

Type Quantity

SCPIMultimeter.**trigger_count**

Gets/sets the number of triggers that the multimeter will accept before returning to an "idle" trigger state.

Note that if the sample_count propery has been changed, the number of readings taken total will be a multiplication of sample count and trigger count (see property SCPIMulimeter.sample_count).

If specified as a *TriggerCount* value, the following options apply:

1. "minimum": 1 trigger
2. "maximum": Maximum value as per instrument manual
3. "default": Instrument default as per instrument manual

4.**"infinity": Continuous. Typically when the buffer is filled in this** case, the older data points are overwritten.

Note that when using triggered measurements, it is recommended that you disable autorange by either explicitly disabling it or specifying your desired range.

Type `int` or *TriggerCount*

`SCPIMultimeter.`**`trigger_delay`**

Gets/sets the time delay which the multimeter will use following receiving a trigger event before starting the measurement.

Units As specified, or assumed to be of units seconds otherwise.

Type `Quantity`

`SCPIMultimeter.`**`trigger_mode`**

Gets/sets the SCPI Multimeter trigger mode.

Example usage:

```
>>> dmm.trigger_mode = dmm.TriggerMode.external
```

Type *TriggerMode*

2.2.3 `SCPIFunctionGenerator` - Generic multimeter using SCPI commands

class `instruments.generic_scpi.`**`SCPIFunctionGenerator`**(*filelike*)

This class is used for communicating with generic SCPI-compliant function generators.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> inst = ik.generic_scpi.SCPIFunctionGenerator.open_tcpip("192.168.1.1")
>>> inst.frequency = 1 * pq.kHz
```

`frequency`

Gets/sets the output frequency.

Units As specified, or assumed to be Hz otherwise.

Type `float` or `Quantity`

`function`

Gets/sets the output function of the function generator

Type `SCPIFunctionGenerator.Function`

`offset`

Gets/sets the offset voltage of the function generator.

Set value should be within correct bounds of instrument.

Units As specified (if a `Quantity`) or assumed to be of units volts.

Type `Quantity` with units volts.

`phase`

2.3 Agilent

2.3.1 `Agilent33220a` Function Generator

class `instruments.agilent.`**`Agilent33220a`**(*filelike*)

The Agilent/Keysight 33220a is a 20MHz function/arbitrary waveform generator. This model has been replaced by the Keysight 33500 series waveform generators. This class may or may not work with these newer models.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> inst = ik.agilent.Agilent33220a.open_gpibusb('/dev/ttyUSB0', 1)
>>> inst.function = inst.Function.sinusoid
>>> inst.frequency = 1 * pq.kHz
>>> inst.output = True
```

class **`Function`**

Enum containing valid functions for the Agilent/Keysight 33220a

`dc` = <Function.dc: 'DC'>

`noise` = <Function.noise: 'NOIS'>

`pulse` = <Function.pulse: 'PULS'>

`ramp` = <Function.ramp: 'RAMP'>

`sinusoid` = <Function.sinusoid: 'SIN'>

`square` = <Function.square: 'SQU'>

`user` = <Function.user: 'USER'>

class `Agilent33220a.`**`LoadResistance`**

Enum containing valid load resistance for the Agilent/Keysight 33220a

`high_impedance` = <LoadResistance.high_impedance: 'INF'>

`maximum` = <LoadResistance.maximum: 'MAX'>

`minimum` = <LoadResistance.minimum: 'MIN'>

class `Agilent33220a.`**`OutputPolarity`**

Enum containing valid output polarity modes for the Agilent/Keysight 33220a

`inverted` = <OutputPolarity.inverted: 'INV'>

`normal` = <OutputPolarity.normal: 'NORM'>

`Agilent33220a.`**`duty_cycle`**

Gets/sets the duty cycle of a square wave.

Duty cycle represents the amount of time that the square wave is at a high level.

Type `int`

`Agilent33220a.`**`frequency`**

`Agilent33220a.`**`function`**

Gets/sets the output function of the function generator

Type *`Agilent33220a.Function`*

Agilent33220a.**load_resistance**

Gets/sets the desired output termination load (ie, the impedance of the load attached to the front panel output connector).

The instrument has a fixed series output impedance of 50ohms. This function allows the instrument to compensate of the voltage divider and accurately report the voltage across the attached load.

Units As specified (if a `Quantity`) or assumed to be of units Ω (ohm).

Type `Quantity` or *Agilent33220a.LoadResistance*

Agilent33220a.**output**

Gets/sets the output enable status of the front panel output connector.

The value `True` corresponds to the output being on, while `False` is the output being off.

Type `bool`

Agilent33220a.**output_polarity**

Gets/sets the polarity of the waveform relative to the offset voltage.

Type *OutputPolarity*

Agilent33220a.**output_sync**

Gets/sets the enabled status of the front panel sync connector.

Type `bool`

Agilent33220a.**phase**

Agilent33220a.**ramp_symmetry**

Gets/sets the ramp symmetry for ramp waves.

Symmetry represents the amount of time per cycle that the ramp wave is rising (unless polarity is inverted).

Type `int`

2.3.2 `Agilent34410a` Digital Multimeter

class instruments.agilent.**Agilent34410a**(*filelike*)

The Agilent 34410a is a very popular 6.5 digit DMM. This class should also cover the Agilent 34401a, 34411a, as well as the backwards compatability mode in the newer Agilent/Keysight 34460a/34461a. You can find the full specifications for these instruments on the Keysight website.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> dmm = ik.agilent.Agilent34410a.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(dmm.measure(dmm.Mode.resistance))
```

abort()

Abort all measurements currently in progress.

clear_memory()

Clears the non-volatile memory of the Agilent 34410a.

fetch()

Transfer readings from instrument memory to the output buffer, and thus to the computer. If currently taking a reading, the instrument will wait until it is complete before executing this command. Readings are NOT erased from memory when using fetch. Use the R? command to read and erase data. Note that the data is transfered as ASCII, and thus it is not recommended to transfer a large number of data points using this method.

Return type `list` of `Quantity` elements

init()

Switch device from "idle" state to "wait-for-trigger state". Measurements will begin when specified triggering conditions are met, following the receipt of the INIT command.

Note that this command will also clear the previous set of readings from memory.

r(*count*)

Have the multimeter perform a specified number of measurements and then transfer them using a binary transfer method. Data will be cleared from instrument memory after transfer is complete. Data is transfered from the instrument in 64-bit double floating point precision format.

Parameters **count** (`int`) – Number of samples to take.

Return type `Quantity` with `numpy.array`

read_data(*sample_count*)

Transfer specified number of data points from reading memory (RGD_STORE) to output buffer. First data point sent to output buffer is the oldest. Data is erased after being sent to output buffer.

Parameters **sample_count** (`int`) – Number of data points to be transfered to output buffer. If set to -1, all points in memory will be transfered.

Return type `list` of `Quantity` elements

read_data_nvmem()

Returns all readings in non-volatile memory (NVMEM).

Return type `list` of `Quantity` elements

read_last_data()

Retrieve the last measurement taken. This can be executed at any time, including when the instrument is currently taking measurements. If there are no data points available, the value `9.91000000E+37` is returned.

Units As specified by the data returned by the instrument.

Return type `Quantity`

read_meter()

Switch device from "idle" state to "wait-for-trigger" state. Immediately after the trigger conditions are met, the data will be sent to the output buffer of the instrument.

This is similar to calling *`init`* and then immediately following *`fetch`*.

Return type `Quantity`

data_point_count

Gets the total number of readings that are located in reading memory (RGD_STORE).

Return type `int`

2.4 Holzworth

2.4.1 HS9000 Multichannel frequency synthesizer

class `instruments.holzworth.`**`HS9000`**(*filelike*)

Communicates with a Holzworth HS-9000 series multi-channel frequency synthesizer.

class **`Channel`**(*hs*, *idx_chan*)

Class representing a physical channel on the Holzworth HS9000

> **Warning:** This class should NOT be manually created by the user. It

is designed to be initialized by the *`HS9000`* class.

`query`(*cmd*)

Function used to send a command to the instrument while wrapping the command with the neccessary identifier for the channel.

Parameters **`cmd`** (*`str`*) – Command that will be sent to the instrument after being prefixed with the channel identifier

Returns The result from the query

Return type `str`

`recall_state`()

Recalls the state of the specified channel from memory.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> hs.channel[0].recall_state()

`reset`()

Resets the setting of the specified channel

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> hs.channel[0].reset()

`save_state`()

Saves the current state of the specified channel.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> hs.channel[0].save_state()

`sendcmd`(*cmd*)

Function used to send a command to the instrument while wrapping the command with the neccessary identifier for the channel.

Parameters **`cmd`** (*`str`*) – Command that will be sent to the instrument after being prefixed with the channel identifier

`frequency`

Gets/sets the frequency of the specified channel. When setting, values are bounded between what is returned by *`frequency_min`* and *`frequency_max`*.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> print(hs.channel[0].frequency) >>> print(hs.channel[0].frequency_min) >>> print(hs.channel[0].frequency_max)

Type `Quantity`

Units As specified or assumed to be of units GHz

`frequency_max`

`frequency_min`

output
Gets/sets the output status of the channel. Setting to `True` will turn the channel's output stage on, while a value of `False` will turn it off.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> print(hs.channel[0].output) >>> hs.channel[0].output = True

Type `bool`

phase
Gets/sets the output phase of the specified channel. When setting, values are bounded between what is returned by *phase_min* and *phase_max*.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> print(hs.channel[0].phase) >>> print(hs.channel[0].phase_min) >>> print(hs.channel[0].phase_max)

Type `Quantity`

Units As specified or assumed to be of units degrees

phase_max

phase_min

power
Gets/sets the output power of the specified channel. When setting, values are bounded between what is returned by *power_min* and *power_max*.

Example usage: >>> import instruments as ik >>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080) >>> print(hs.channel[0].power) >>> print(hs.channel[0].power_min) >>> print(hs.channel[0].power_max)

Type `Quantity`

Units `instruments.units.dBm`

power_max

power_min

temperature
Gets the current temperature of the specified channel.

Units As specified by the instrument.

Return type `Quantity`

HS9000.**channel**
Gets a specific channel on the HS9000. The desired channel is accessed like one would access a list.

Example usage:

```
>>> import instruments as ik
>>> hs = ik.holzworth.HS9000.open_tcpip("192.168.0.2", 8080)
>>> print(hs.channel[0].frequency)
```

Returns A channel object for the HS9000

Return type *Channel*

HS9000.**name**
Gets identification string of the HS9000

Returns The string as usually returned by `*IDN?` on SCPI instruments

Return type `str`

HS9000.**ready**
Gets the ready status of the HS9000.

Returns If the instrument is ready for operation

Return type bool

2.5 Hewlett-Packard

2.5.1 HP3456a Digital Voltmeter

class instruments.hp.**HP3456a**(*filelike*)
The *HP3456a* is a 6 1/2 digit bench multimeter.

It supports DCV, ACV, ACV + DCV, 2 wire Ohms, 4 wire Ohms, DCV/DCV Ratio, ACV/DCV Ratio, Offset compensated 2 wire Ohms and Offset compensated 4 wire Ohms measurements.

Measurements can be further extended using a system math mode that allows for pass/fail, statistics, dB/dBm, null, scale and percentage readings.

HP3456a is a HPIB / pre-448.2 instrument.

class MathMode
Enum with the supported math modes

db = <MathMode.db: 9>

dbm = <MathMode.dbm: 4>

null = <MathMode.null: 3>

off = <MathMode.off: 0>

pass_fail = <MathMode.pass_fail: 1>

percent = <MathMode.percent: 8>

scale = <MathMode.scale: 7>

statistic = <MathMode.statistic: 2>

thermistor_c = <MathMode.thermistor_c: 6>

thermistor_f = <MathMode.thermistor_f: 5>

class HP3456a.**Mode**
Enum containing the supported mode codes

acv = <Mode.acv: 'S0F2'>

acvdcv = <Mode.acvdcv: 'S0F3'>

dcv = <Mode.dcv: 'S0F1'>

oc_resistence_2wire = <Mode.oc_resistence_2wire: 'S1F4'>

oc_resistence_4wire = <Mode.oc_resistence_4wire: 'S1F5'>

ratio_acv_dcv = <Mode.ratio_acv_dcv: 'S1F2'>

ratio_acvdcv_dcv = <Mode.ratio_acvdcv_dcv: 'S1F3'>

ratio_dcv_dcv = <Mode.ratio_dcv_dcv: 'S1F1'>

resistance_2wire = <Mode.resistance_2wire: 'S0F4'>

resistance_4wire = <Mode.resistance_4wire: 'S0F5'>

class HP3456a.**Register**

Enum with the register names for all *HP3456a* internal registers.

count = <Register.count: 'C'>

delay = <Register.delay: 'D'>

lower = <Register.lower: 'L'>

mean = <Register.mean: 'M'>

nplc = <Register.nplc: 'I'>

number_of_digits = <Register.number_of_digits: 'G'>

number_of_readings = <Register.number_of_readings: 'N'>

r = <Register.r: 'R'>

upper = <Register.upper: 'U'>

variance = <Register.variance: 'V'>

y = <Register.y: 'Y'>

z = <Register.z: 'Z'>

class HP3456a.**TriggerMode**

Enum with valid trigger modes.

external = <TriggerMode.external: 2>

hold = <TriggerMode.hold: 4>

internal = <TriggerMode.internal: 1>

single = <TriggerMode.single: 3>

class HP3456a.**ValidRange**

Enum with the valid ranges for voltage, resistance, and number of powerline cycles to integrate over.

nplc = <ValidRange.nplc: (0.1, 1.0, 10.0, 100.0)>

resistance = <ValidRange.resistance: (100.0, 1000.0, 10000.0, 100000.0, 1000000.0, 10000000.0, 100000000.0, 1000

voltage = <ValidRange.voltage: (0.1, 1.0, 10.0, 100.0, 1000.0)>

HP3456a.**auto_range**()

Set input range to auto. The *HP3456a* should upscale when a reading is at 120% and downscale when it below 11% full scale. Note that auto ranging can increase the measurement time.

HP3456a.**fetch**(*mode=None*)

Retrieve n measurements after the HP3456a has been instructed to perform a series of similar measurements. Typically the mode, range, nplc, analog filter, autozero is set along with the number of measurements to take. The series is then started at the trigger command.

Example usage:

```
>>> dmm.number_of_digits = 6
>>> dmm.auto_range()
>>> dmm.nplc = 1
>>> dmm.mode = dmm.Mode.resistance_2wire
>>> n = 100
>>> dmm.number_of_readings = n
>>> dmm.trigger()
```

```
>>> time.sleep(n * 0.04)
>>> v = dmm.fetch(dmm.Mode.resistance_2wire)
>>> print len(v)
10
```

Parameters **mode** (*HP3456a.Mode*) – Desired measurement mode. If not specified, the previous set mode will be used, but no measurement unit will be returned.

Returns A series of measurements from the multimeter.

Return type Quantity

HP3456a.**measure**(*mode=None*)

Instruct the HP3456a to perform a one time measurement. The measurement will use the current set registers for the measurement (number_of_readings, number_of_digits, nplc, delay, mean, lower, upper, y and z) and will immediately take place.

Note that using *HP3456a.measure()* will override the *trigger_mode* to *HP3456a.TriggerMode.single*

Example usage:

```
>>> dmm = ik.hp.HP3456a.open_gpibusb("/dev/ttyUSB0", 22)
>>> dmm.number_of_digits = 6
>>> dmm.nplc = 1
>>> print dmm.measure(dmm.Mode.resistance_2wire)
```

Parameters **mode** (*HP3456a.Mode*) – Desired measurement mode. If not specified, the previous set mode will be used, but no measurement unit will be returned.

Returns A measurement from the multimeter.

Return type Quantity

HP3456a.**trigger**()

Signal a single manual trigger event to the *HP3456a*.

HP3456a.**autozero**

Set the autozero mode.

This is used to compensate for offsets in the dc input amplifier circuit of the multimeter. If set, the amplifier"s input circuit is shorted to ground prior to actual measurement in order to take an offset reading. This offset is then used to compensate for drift in the next measurement. When disabled, one offset reading is taken immediately and stored into memory to be used for all successive measurements onwards. Disabling autozero increases the *HP3456a*"s measurement speed, and also makes the instrument more suitable for high impendance measurements since no input switching is done.

HP3456a.**count**

Get the number of measurements taken from *HP3456a.Register.count* when in *HP3456a.MathMode.statistic*.

Return type int

HP3456a.**delay**

Get/set the delay that is waited after a trigger for the input to settle using *HP3456a.Register.delay*.

Type As specified, assumed to be s otherwise

Return type s

HP3456a.**filter**

Set the analog filter mode.

The *HP3456a* has a 3 pole active filter with greater than 60dB attenuation at frequencies of 50Hz and higher. The filter is applied between the input terminals and input amplifier. When in ACV or ACV+DCV functions the filter is applied to the output of the ac converter and input amplifier. In these modes select the filter for measurements below 400Hz.

HP3456a.**input_range**

Set the input range to be used.

The *HP3456a* has separate ranges for `ohm` and for `volt`. The range value sent to the instrument depends on the unit set on the input range value. `auto` selects auto ranging.

Type Quantity

HP3456a.**lower**

Get/set the value in *HP3456a.Register.lower*, which indicates the lowest value measurement made while in *HP3456a.MathMode.statistic*, or the lowest value preset for *HP3456a.MathMode.pass_fail*.

Type float

HP3456a.**math_mode**

Set the math mode.

The *HP3456a* has a number of different math modes that can change measurement output, or can provide additional statistics. Interaction with these modes is done via the *HP3456a.Register*.

Type *HP3456a.MathMode*

HP3456a.**mean**

Get the mean over *HP3456a.Register.count* measurements from *HP3456a.Register.mean* when in *HP3456a.MathMode.statistic*.

Return type float

HP3456a.**mode**

Set the measurement mode.

Type *HP3456a.Mode*

HP3456a.**nplc**

Get/set the number of powerline cycles to integrate per measurement using *HP3456a.Register.nplc*.

Setting higher values increases accuracy at the cost of a longer measurement time. The implicit assumption is that the input reading is stable over the number of powerline cycles to integrate.

Type int

HP3456a.**number_of_digits**

Get/set the number of digits used in measurements using *HP3456a.Register.number_of_digits*.

Set to higher values to increase accuracy at the cost of measurement speed.

Type int

HP3456a.**number_of_readings**

Get/set the number of readings done per trigger/measurement cycle using *HP3456a.Register.number_of_readings*.

Type float

Return type float

HP3456a.**r**
Get/set the value in *HP3456a.Register.r*, which indicates the resistor value used while in *HP3456a.MathMode.dbm* or the number of recalled readings in reading storage mode.

Type float

Return type float

HP3456a.**relative**
Enable or disable HP3456a.MathMode.Null on the instrument.

Type bool

HP3456a.**trigger_mode**
Set the trigger mode.

Note that using *HP3456a.measure()* will override the *trigger_mode* to *HP3456a.TriggerMode.single*.

Type *HP3456a.TriggerMode*

HP3456a.**upper**
Get/set the value in *HP3456a.Register.upper*, which indicates the highest value measurement made while in *HP3456a.MathMode.statistic*, or the highest value preset for *HP3456a.MathMode.pass_fail*.

Type float

Return type float

HP3456a.**variance**
Get the variance over *HP3456a.Register.count* measurements from *HP3456a.Register.variance* when in *HP3456a.MathMode.statistic*.

Return type float

HP3456a.**y**
Get/set the value in *HP3456a.Register.y* to be used in calculations when in *HP3456a.MathMode.scale* or *HP3456a.MathMode.percent*.

Type float

Return type float

HP3456a.**z**
Get/set the value in *HP3456a.Register.z* to be used in calculations when in *HP3456a.MathMode.scale* or the first reading when in *HP3456a.MathMode.statistic*.

Type float

Return type float

2.5.2 HP6624a Power Supply

class instruments.hp.**HP6624a**(*filelike*)
The HP6624a is a multi-output power supply.

This class can also be used for HP662xa, where x=1,2,3,4,7. Note that some models have less channels then the HP6624 and it is up to the user to take this into account. This can be changed with the *channel_count* property.

Example usage:

```
>>> import instruments as ik
>>> psu = ik.hp.HP6624a.open_gpibusb('/dev/ttyUSB0', 1)
>>> psu.channel[0].voltage = 10 # Sets channel 1 voltage to 10V.
```

class Channel (*hp*, *idx*)

Class representing a power output channel on the HP6624a.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *HP6624a* class.

query (*cmd*)

Function used to send a command to the instrument while wrapping the command with the neccessary identifier for the channel.

Parameters **cmd** (*str*) – Command that will be sent to the instrument after being prefixed with the channel identifier

Returns The result from the query

Return type str

reset ()

Reset overvoltage and overcurrent errors to resume operation.

sendcmd (*cmd*)

Function used to send a command to the instrument while wrapping the command with the neccessary identifier for the channel.

Parameters **cmd** (*str*) – Command that will be sent to the instrument after being prefixed with the channel identifier

current

Gets/sets the current of the specified channel. If the device is in constant voltage mode, this sets the current limit.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be A otherwise.

Type float or Quantity

current_sense

Gets the actual output current as measured by the instrument for the specified channel.

Units A (amps)

Return type Quantity

mode

Gets/sets the mode for the specified channel.

output

Gets/sets the outputting status of the specified channel.

This is a toggle setting. True will turn on the channel output while False will turn it off.

Type bool

overcurrent

Gets/sets the overcurrent protection setting for the specified channel.

This is a toggle setting. It is either on or off.

Type bool

overvoltage

Gets/sets the overvoltage protection setting for the specified channel.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

voltage

Gets/sets the voltage of the specified channel. If the device is in constant current mode, this sets the voltage limit.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

voltage_sense

Gets the actual voltage as measured by the sense wires for the specified channel.

Units V (volts)

Return type Quantity

class HP6624a.**Mode**

Enum holding typical valid output modes for a power supply.

However, for the HP6624a I believe that it is only capable of constant-voltage output, so this class current does not do anything and is just a placeholder.

current = <Mode.current: 0>

voltage = <Mode.current: 0>

HP6624a.**clear**()

Taken from the manual:

Return the power supply to its power-on state and all parameters are returned to their initial power-on values except the following:

1. The store/recall registers are not cleared.
2. The power supply remains addressed to listen.
3. The PON bit in the serial poll register is cleared.

HP6624a.**channel**

Gets a specific channel object. The desired channel is specified like one would access a list.

Return type *HP6624a.Channel*

See also:

HP6624a for example using this property.

HP6624a.**channel_count**

Gets/sets the number of output channels available for the connected power supply.

Type int

HP6624a.**current**

Gets/sets the current for all four channels.

Units As specified (if a Quantity) or assumed to be of units Amps.

Type list of Quantity with units Amp

HP6624a.**current_sense**

Gets the actual current as measured by the instrument for all channels.

Units A (amps)

Return type tuple of Quantity

HP6624a.**voltage**
Gets/sets the voltage for all four channels.

Units As specified (if a Quantity) or assumed to be of units Volts.

Type list of Quantity with units Volt

HP6624a.**voltage_sense**
Gets the actual voltage as measured by the sense wires for all channels.

Units V (volts)

Return type tuple of Quantity

2.5.3 HP6632b Power Supply

class instruments.hp.**HP6632b**(*filelike*)
The HP6632b is a system dc power supply with an output rating of 0-20V/0-5A, precision low current measurement and low output noise.

According to the manual this class MIGHT be usable for any HP power supply with a model number

- HP663Xb with X in {1, 2, 3, 4},
- HP661Xc with X in {1,2, 3, 4} and
- HP663X2A for X in {1, 3}, without the additional measurement capabilities.

HOWEVER, it has only been tested by the author with HP6632b supplies.

Example usage:

```
>>> import instruments as ik
>>> psu = ik.hp.HP6632b.open_gpibusb('/dev/ttyUSB0', 6)
>>> psu.voltage = 10             # Sets voltage to 10V.
>>> psu.output = True            # Enable output
>>> psu.voltage
array(10.0) * V
>>> psu.voltage_trigger = 20     # Set transient trigger voltage
>>> psu.init_output_trigger()    # Prime instrument to initiated state, ready for trigger
>>> psu.trigger()                # Send trigger
>>> psu.voltage
array(10.0) * V
```

class ALCBandwidth
Enum containing valid ALC bandwidth modes for the hp6632b

fast = <ALCBandwidth.fast: 60000>

normal = <ALCBandwidth.normal: 15000>

class HP6632b.**DFISource**
Enum containing valid DFI sources for the hp6632b

event_status_bit = <DFISource.event_status_bit: 'ESB'>

off = <DFISource.off: 'OFF'>

operation = <DFISource.operation: 'OPER'>

questionable = <DFISource.questionable: 'QUES'>

request_service_bit = <DFISource.request_service_bit: 'RQS'>

class `HP6632b.DigitalFunction`

Enum containing valid digital function modes for the hp6632b

`data` = **<DigitalFunction.data: ‘DIG’>**

`remote_inhibit` = **<DigitalFunction.remote_inhibit: ‘RIDF’>**

class `HP6632b.ErrorCodes`

Enum containing generic-SCPI error codes along with codes specific to the HP6632b.

`block_data_error` = **<ErrorCodes.block_data_error: -160>**

`block_data_not_allowed` = **<ErrorCodes.block_data_not_allowed: -168>**

`cal_not_enabled` = **<ErrorCodes.cal_not_enabled: 403>**

`cal_password_incorrect` = **<ErrorCodes.cal_password_incorrect: 402>**

`cal_switch_prevents_cal` = **<ErrorCodes.cal_switch_prevents_cal: 401>**

`character_data_error` = **<ErrorCodes.character_data_error: -140>**

`character_data_not_allowed` = **<ErrorCodes.character_data_not_allowed: -148>**

`character_data_too_long` = **<ErrorCodes.character_data_too_long: -144>**

`command_error` = **<ErrorCodes.command_error: -100>**

`command_header_error` = **<ErrorCodes.command_header_error: -110>**

`command_only_applic_rs232` = **<ErrorCodes.command_only_applic_rs232: 602>**

`computed_prog_cal_constants_incorrect` = **<ErrorCodes.computed_prog_cal_constants_incorrect: 405>**

`computed_readback_cal_const_incorrect` = **<ErrorCodes.computed_readback_cal_const_incorrect: 404>**

`curr_or_volt_fetch_incompat_with_last_acq` = **<ErrorCodes.curr_or_volt_fetch_incompat_with_last_a**

`cv_or_cc_status_incorrect` = **<ErrorCodes.cv_or_cc_status_incorrect: 407>**

`data_out_of_range` = **<ErrorCodes.data_out_of_range: -222>**

`data_type_error` = **<ErrorCodes.data_type_error: -104>**

`digital_io_selftest` = **<ErrorCodes.digital_io_selftest: 80>**

`execution_error` = **<ErrorCodes.execution_error: -200>**

`exponent_too_large` = **<ErrorCodes.exponent_too_large: -123>**

`expression_error` = **<ErrorCodes.expression_error: -170>**

`expression_not_allowed` = **<ErrorCodes.expression_not_allowed: -178>**

`front_panel_uart_buffer_overrun` = **<ErrorCodes.front_panel_uart_buffer_overrun: 223>**

`front_panel_uart_framing` = **<ErrorCodes.front_panel_uart_framing: 221>**

`front_panel_uart_overrun` = **<ErrorCodes.front_panel_uart_overrun: 220>**

`front_panel_uart_parity` = **<ErrorCodes.front_panel_uart_parity: 222>**

`front_panel_uart_timeout` = **<ErrorCodes.front_panel_uart_timeout: 224>**

`get_not_allowed` = **<ErrorCodes.get_not_allowed: -105>**

`header_separator_error` = **<ErrorCodes.header_separator_error: -111>**

`header_suffix_out_of_range` = **<ErrorCodes.header_suffix_out_of_range: -114>**

`illegal_macro_label` = **<ErrorCodes.illegal_macro_label: -273>**

illegal_parameter_value = <ErrorCodes.illegal_parameter_value: -224>

incorrect_seq_cal_commands = <ErrorCodes.incorrect_seq_cal_commands: 406>

ingrd_recv_buffer_overrun = <ErrorCodes.ingrd_recv_buffer_overrun: 213>

invalid_block_data = <ErrorCodes.invalid_block_data: -161>

invalid_character = <ErrorCodes.invalid_character: -101>

invalid_character_data = <ErrorCodes.invalid_character_data: -141>

invalid_character_in_number = <ErrorCodes.invalid_character_in_number: -121>

invalid_expression = <ErrorCodes.invalid_expression: -171>

invalid_inside_macro_definition = <ErrorCodes.invalid_inside_macro_definition: -183>

invalid_outside_macro_definition = <ErrorCodes.invalid_outside_macro_definition: -181>

invalid_separator = <ErrorCodes.invalid_separator: -103>

invalid_string_data = <ErrorCodes.invalid_string_data: -151>

invalid_suffix = <ErrorCodes.invalid_suffix: -131>

macro_error_180 = <ErrorCodes.macro_error_180: -180>

macro_error_270 = <ErrorCodes.macro_error_270: -270>

macro_execution_error = <ErrorCodes.macro_execution_error: -272>

macro_parameter_error = <ErrorCodes.macro_parameter_error: -184>

macro_recursion_error = <ErrorCodes.macro_recursion_error: -276>

macro_redefinition_not_allowed = <ErrorCodes.macro_redefinition_not_allowed: -277>

measurement_overrange = <ErrorCodes.measurement_overrange: 604>

missing_parameter = <ErrorCodes.missing_parameter: -109>

no_error = <ErrorCodes.no_error: 0>

numeric_data_error = <ErrorCodes.numeric_data_error: -120>

numeric_data_not_allowed = <ErrorCodes.numeric_data_not_allowed: -128>

operation_complete = <ErrorCodes.operation_complete: -800>

out_of_memory = <ErrorCodes.out_of_memory: -225>

output_mode_must_be_normal = <ErrorCodes.output_mode_must_be_normal: 408>

ovdac_selftest = <ErrorCodes.ovdac_selftest: 15>

parameter_not_allowed = <ErrorCodes.parameter_not_allowed: -108>

power_on = <ErrorCodes.power_on: -500>

program_mnemonic_too_long = <ErrorCodes.program_mnemonic_too_long: -112>

query_deadlocked = <ErrorCodes.query_deadlocked: -430>

query_error = <ErrorCodes.query_error: -400>

query_interrupted = <ErrorCodes.query_interrupted: -410>

query_unterminated = <ErrorCodes.query_unterminated: -420>

query_unterminated_after_indefinite_response = <ErrorCodes.query_unterminated_after_indefinite_

ram_cal_checksum_failed = <ErrorCodes.ram_cal_checksum_failed: 3>

ram_config_checksum_failed = <ErrorCodes.ram_config_checksum_failed: 2>

ram_rd0_checksum_failed = <ErrorCodes.ram_rd0_checksum_failed: 1>

ram_rst_checksum_failed = <ErrorCodes.ram_rst_checksum_failed: 5>

ram_selftest = <ErrorCodes.ram_selftest: 10>

ram_state_checksum_failed = <ErrorCodes.ram_state_checksum_failed: 4>

request_control_event = <ErrorCodes.request_control_event: -700>

rs232_recv_framing_error = <ErrorCodes.rs232_recv_framing_error: 216>

rs232_recv_overrun_error = <ErrorCodes.rs232_recv_overrun_error: 218>

rs232_recv_parity_error = <ErrorCodes.rs232_recv_parity_error: 217>

string_data_error = <ErrorCodes.string_data_error: -150>

string_data_not_allowed = <ErrorCodes.string_data_not_allowed: -158>

suffix_error = <ErrorCodes.suffix_error: -130>

suffix_not_allowed = <ErrorCodes.suffix_not_allowed: -138>

suffix_too_long = <ErrorCodes.suffix_too_long: -134>

syntax_error = <ErrorCodes.syntax_error: -102>

system_error = <ErrorCodes.system_error: -310>

too_many_digits = <ErrorCodes.too_many_digits: -124>

too_many_errors = <ErrorCodes.too_many_errors: -350>

too_many_sweep_points = <ErrorCodes.too_many_sweep_points: 601>

too_much_data = <ErrorCodes.too_much_data: -223>

undefined_header = <ErrorCodes.undefined_header: -113>

unexpected_number_of_parameters = <ErrorCodes.unexpected_number_of_parameters: -115>

user_request_event = <ErrorCodes.user_request_event: -600>

vdac_idac_selftest1 = <ErrorCodes.vdac_idac_selftest1: 11>

vdac_idac_selftest2 = <ErrorCodes.vdac_idac_selftest2: 12>

vdac_idac_selftest3 = <ErrorCodes.vdac_idac_selftest3: 13>

vdac_idac_selftest4 = <ErrorCodes.vdac_idac_selftest4: 14>

class HP6632b.**RemoteInhibit**

Enum containing vlaid remote inhibit modes for the hp6632b.

latching = <RemoteInhibit.latching: 'LATC'>

live = <RemoteInhibit.live: 'LIVE'>

off = <RemoteInhibit.off: 'OFF'>

class HP6632b.**SenseWindow**

Enum containing valid sense window modes for the hp6632b.

hanning = <SenseWindow.hanning: 'HANN'>

rectangular = <SenseWindow.rectangular: 'RECT'>

HP6632b.**abort_output_trigger**()
Set the output trigger system to the idle state.

HP6632b.**check_error_queue**()
Checks and clears the error queue for this device, returning a list of ErrorCodes or int elements for each error reported by the connected instrument.

HP6632b.**init_output_trigger**()
Set the output trigger system to the initiated state. In this state, the power supply will respond to the next output trigger command.

HP6632b.**current_sense_range**
Get/set the sense current range by the current max value.

A current of 20mA or less selects the low-current range, a current value higher than that selects the high-current range. The low current range increases the low current measurement sensitivity and accuracy.

Units As specified, or assumed to be A otherwise.

Type float or Quantity

HP6632b.**current_trigger**
Gets/sets the pending triggered output current.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be A otherwise.

Type float or Quantity

HP6632b.**digital_data**
Get/set digital in+out port to data. Data can be an integer from 0-7.

Type int

HP6632b.**digital_function**
Get/set the inhibit+fault port to digital in+out or vice-versa.

Type *DigitalFunction*

HP6632b.**display_brightness**

HP6632b.**display_contrast**

HP6632b.**init_output_continuous**
Get/set the continuous output trigger. In this state, the power supply will remain in the initiated state, and respond continuously on new incoming triggers by applying the set voltage and current trigger levels.

Type bool

HP6632b.**line_frequency**

HP6632b.**output_dfi**
Get/set the discrete fault indicator (DFI) output from the dc source. The DFI is an open-collector logic signal connected to the read panel FLT connection, that can be used to signal external devices when a fault is detected.

Type bool

HP6632b.**output_dfi_source**
Get/set the source for discrete fault indicator (DFI) events.

Type *DFISource*

HP6632b.**output_protection_delay**
Get/set the time between programming of an output change that produces a constant current condition and the recording of that condigition in the Operation Status Condition register. This command also delays over current protection, but not overvoltage protection.

Units As specified, or assumed to be s otherwise.

Type float or Quantity

HP6632b.**output_remote_inhibit**
Get/set the remote inhibit signal. Remote inhibit is an external, chassis-referenced logic signal routed through the rear panel INH connection, which allows an external device to signal a fault.

Type *RemoteInhibit*

HP6632b.**sense_sweep_interval**
Get/set the digitizer sample spacing. Can be set from 15.6 us to 31200 seconds, the interval will be rounded to the nearest 15.6 us increment.

Units As specified, or assumed to be s otherwise.

Type float or Quantity

HP6632b.**sense_sweep_points**
Get/set the number of points in a measurement sweep.

Type int

HP6632b.**sense_window**
Get/set the measurement window function.

Type *SenseWindow*

HP6632b.**voltage_alc_bandwidth**
Get the "automatic level control bandwidth" which for the HP66332A and HP6631-6634 determines if the output capacitor is in circuit. `Normal` denotes that it is, and `Fast` denotes that it is not.

Type *ALCBandwidth*

HP6632b.**voltage_trigger**
Gets/sets the pending triggered output voltage.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

2.5.4 HP6652a Single Output Power Supply

class instruments.hp.**HP6652a**(*filelike*)
The HP6652a is a single output power supply.

Because it is a single channel output, this object inherits from both PowerSupply and PowerSupplyChannel.

According to the manual, this class MIGHT be usable for any HP power supply with a model number HP66XYA, where X is in {4,5,7,8,9} and Y is a digit(?). (e.g. HP6652A and HP6671A)

HOWEVER, it has only been tested by the author with an HP6652A power supply.

Example usage:

```
>>> import time
>>> import instruments as ik
>>> psu = ik.hp.HP6652a.open_serial('/dev/ttyUSB0', 57600)
>>> psu.voltage = 3 # Sets output voltage to 3V.
>>> psu.output = True
>>> psu.voltage
array(3.0) * V
>>> psu.voltage_sense < 5
True
>>> psu.output = False
>>> psu.voltage_sense < 1
True
>>> psu.display_textmode=True
>>> psu.display_text("test GOOD")
'TEST GOOD'
>>> time.sleep(5)
>>> psu.display_textmode=False
```

display_text(*text_to_display*)

Sends up to 12 (uppercase) alphanumerics to be sent to the front-panel LCD display. Some punctuation is allowed, and can affect the number of characters allowed. See the programming manual for the HP6652A for more details.

Because the maximum valid number of possible characters is 15 (counting the possible use of punctuation), the text will be truncated to 15 characters before the command is sent to the instrument.

If an invalid string is sent, the command will fail silently. Any lowercase letters in the text_to_display will be converted to uppercase before the command is sent to the instrument.

No attempt to validate punctuation is currently made.

Because the string cannot be read back from the instrument, this method returns the actual string value sent.

Parameters **text_to_display** (*'str'*) – The text that you wish to have displayed on the front-panel LCD

Returns Returns the version of the provided string that will be send to the instrument. This means it will be truncated to a maximum of 15 characters and changed to all upper case.

Return type `str`

reset()

Reset overvoltage and overcurrent errors to resume operation.

channel

Return the channel (which in this case is the entire instrument, since there is only 1 channel on the HP6652a.)

Return type 'tuple' of length 1 containing a reference back to the parent HP6652a object.

current

Gets/sets the output current.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be A otherwise.

Type `float` or `Quantity`

current_sense

Gets the actual output current as measured by the sense wires.

Units A (amps)

Return type Quantity

display_textmode
Gets/sets the display mode.

This is a toggle setting. True will allow text to be sent to the front-panel LCD with the display_text() method. False returns to the normal display mode.

See also:

display_text()

Type bool

mode
Unimplemented.

name
The name of the connected instrument, as reported by the standard SCPI command *IDN?.

Return type str

output
Gets/sets the output status.

This is a toggle setting. True will turn on the instrument output while False will turn it off.

Type bool

overcurrent
Gets/sets the overcurrent protection setting.

This is a toggle setting. It is either on or off.

Type bool

overvoltage
Gets/sets the overvoltage protection setting in volts.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

voltage
Gets/sets the output voltage.

Note there is no bounds checking on the value specified.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

voltage_sense
Gets the actual output voltage as measured by the sense wires.

Units V (volts)

Return type Quantity

2.6 Keithley

2.6.1 Keithley195 Digital Multimeter

class instruments.keithley.**Keithley195**(*filelike*)

The Keithley 195 is a 5 1/2 digit auto-ranging digital multimeter. You can find the full specifications list in the Keithley 195 user's guide.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> dmm = ik.keithley.Keithley195.open_gpibusb('/dev/ttyUSB0', 12)
>>> print dmm.measure(dmm.Mode.resistance)
```

class Mode

Enum containing valid measurement modes for the Keithley 195

current_ac = <Mode.current_ac: 4>

current_dc = <Mode.current_dc: 3>

resistance = <Mode.resistance: 2>

voltage_ac = <Mode.voltage_ac: 1>

voltage_dc = <Mode.voltage_dc: 0>

class Keithley195.**TriggerMode**

Enum containing valid trigger modes for the Keithley 195

ext_continuous = <TriggerMode.ext_continuous: 6>

ext_one_shot = <TriggerMode.ext_one_shot: 7>

get_continuous = <TriggerMode.get_continuous: 2>

get_one_shot = <TriggerMode.get_one_shot: 3>

talk_continuous = <TriggerMode.talk_continuous: 0>

talk_one_shot = <TriggerMode.talk_one_shot: 1>

x_continuous = <TriggerMode.x_continuous: 4>

x_one_shot = <TriggerMode.x_one_shot: 5>

class Keithley195.**ValidRange**

Enum containing valid range settings for the Keithley 195

current_ac = <ValidRange.current_ac: (2e-05, 0.0002, 0.002, 0.02, 0.2, 2, 2)>

current_dc = <ValidRange.current_dc: (2e-05, 0.0002, 0.002, 0.02, 0.2, 2)>

resistance = <ValidRange.resistance: (20, 200, 2000, 20000.0, 200000.0, 2000000.0, 20000000.0)>

voltage_ac = <ValidRange.voltage_ac: (0.02, 0.2, 2, 20, 200, 700)>

voltage_dc = <ValidRange.voltage_dc: (0.02, 0.2, 2, 20, 200, 1000)>

Keithley195.**auto_range**()

Turn on auto range for the Keithley 195.

This is the same as calling `Keithley195.input_range = 'auto'`

Keithley195.**get_status_word**()

Retreive the status word from the instrument. This contains information regarding the various settings of the instrument.

The function *parse_status_word* is designed to parse the return string from this function.

Returns String containing setting information of the instrument

Return type str

Keithley195.**measure**(*mode=None*)

Instruct the Keithley 195 to perform a one time measurement. The instrument will use default parameters for the requested measurement. The measurement will immediately take place, and the results are directly sent to the instrument's output buffer.

Method returns a Python quantity consisting of a numpy array with the instrument value and appropriate units.

With the 195, it is HIGHLY recommended that you seperately set the mode and let the instrument settle into the new mode. This can sometimes take longer than the 2 second delay added in this method. In our testing the 2 seconds seems to be sufficient but we offer no guarentee.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> dmm = ik.keithley.Keithley195.open_gpibusb('/dev/ttyUSB0', 12)
>>> print(dmm.measure(dmm.Mode.resistance))
```

Parameters **mode** (*Keithley195.Mode*) – Desired measurement mode. This must always be specified in order to provide the correct return units.

Returns A measurement from the multimeter.

Return type Quantity

static Keithley195.**parse_status_word**(*statusword*)

Parse the status word returned by the function *get_status_word*.

Returns a dict with the following keys: {trigger,mode,range,eoi,buffer,rate,srqmode,relative,de selftest,dataformat,datacontrol,filter,terminator}

Parameters **statusword** – Byte string to be unpacked and parsed

Type str

Returns A parsed version of the status word as a Python dictionary

Return type dict

Keithley195.**trigger**()

Tell the Keithley 195 to execute all commands that it has received.

Do note that this is different from the standard SCPI *TRG command (which is not supported by the 195 anyways).

Keithley195.**input_range**

Gets/sets the range of the Keithley 195 input terminals. The valid range settings depends on the current mode of the instrument. They are listed as follows:

1. voltage_dc = (20e-3, 200e-3, 2, 20, 200, 1000)
2. voltage_ac = (20e-3, 200e-3, 2, 20, 200, 700)

3.current_dc = `(20e-6, 200e-6, 2e-3, 20e-3, 200e-3, 2)`

4.current_ac = `(20e-6, 200e-6, 2e-3, 20e-3, 200e-3, 2)`

5.resistance = `(20, 200, 2000, 20e3, 200e3, 2e6, 20e6)`

All modes will also accept the string `auto` which will set the 195 into auto ranging mode.

Return type `Quantity` or `str`

`Keithley195.`**`mode`**

Gets/sets the measurement mode for the Keithley 195. The base model only has DC voltage and resistance measurements. In order to use AC voltage, DC current, and AC current measurements your unit must be equiped with option 1950.

Example use:

```
>>> import instruments as ik
>>> dmm = ik.keithley.Keithley195.open_gpibusb('/dev/ttyUSB0', 12)
>>> dmm.mode = dmm.Mode.resistance
```

Type *`Keithley195.Mode`*

`Keithley195.`**`relative`**

Gets/sets the zero command (relative measurement) mode of the Keithley 195.

As stated in the manual: The zero mode serves as a means for a baseline suppression. When the correct zero command is send over the bus, the instrument will enter the zero mode, as indicated by the front panel ZERO indicator light. All reading displayed or send over the bus while zero is enabled are the difference between the stored baseline adn the actual voltage level. For example, if a 100mV baseline is stored, 100mV will be subtracted from all subsequent readings as long as the zero mode is enabled. The value of the stored baseline can be as little as a few microvolts or as large as the selected range will permit.

See the manual for more information.

Type `bool`

`Keithley195.`**`trigger_mode`**

Gets/sets the trigger mode of the Keithley 195.

There are two different trigger settings for four different sources. This means there are eight different settings for the trigger mode.

The two types are continuous and one-shot. Continuous has the instrument continuously sample the resistance. One-shot performs a single resistance measurement.

The three trigger sources are on talk, on GET, and on "X". On talk refers to addressing the instrument to talk over GPIB. On GET is when the instrument receives the GPIB command byte for "group execute trigger". On "X" is when one sends the ASCII character "X" to the instrument. This character is used as a general execute to confirm commands send to the instrument. In InstrumentKit, "X" is sent after each command so it is not suggested that one uses on "X" triggering. Last, is external triggering. This is the port on the rear of the instrument. Refer to the manual for electrical characteristics of this port.

Type *`Keithley195.TriggerMode`*

2.6.2 `Keithley580` Microohm Meter

class `instruments.keithley.`**`Keithley580`**(*filelike*)

The Keithley Model 580 is a 4 1/2 digit resolution autoranging micro-ohmmeter with a +- 20,000 count LCD. It is designed for low resistance measurement requirements from 10uΩ to 200kΩ.

The device needs some processing time (manual reports 300-500ms) after a command has been transmitted.

class `Drive`

Enum containing valid drive modes for the Keithley 580

`dc` **= <Drive.dc: 1>**

`pulsed` **= <Drive.pulsed: 0>**

class `Keithley580.`**`Polarity`**

Enum containing valid polarity modes for the Keithley 580

`negative` **= <Polarity.negative: 1>**

`positive` **= <Polarity.positive: 0>**

class `Keithley580.`**`TriggerMode`**

Enum containing valid trigger modes for the Keithley 580

`get_continuous` **= <TriggerMode.get_continuous: 2>**

`get_one_shot` **= <TriggerMode.get_one_shot: 3>**

`talk_continuous` **= <TriggerMode.talk_continuous: 0>**

`talk_one_shot` **= <TriggerMode.talk_one_shot: 1>**

`trigger_continuous` **= <TriggerMode.trigger_continuous: 4>**

`trigger_one_shot` **= <TriggerMode.trigger_one_shot: 5>**

`Keithley580.`**`auto_range`**`()`

Turn on auto range for the Keithley 580.

This is the same as calling the `Keithley580.set_resistance_range` method and setting the parameter to "AUTO".

`Keithley580.`**`get_status_word`**`()`

The keithley will not always respond with the statusword when asked. We use a simple heuristic here: request it up to 5 times, using a 1s delay to allow the keithley some thinking time.

Return type `str`

`Keithley580.`**`measure`**`()`

Perform a measurement with the Keithley 580.

The usual mode parameter is ignored for the Keithley 580 as the only valid mode is resistance.

Return type `Quantity`

static `Keithley580.`**`parse_measurement`**`(`*measurement*`)`

Parse the measurement string returned by the instrument.

Returns a dict with the following keys: `{status,polarity,drycircuit,drive,resistance}`

Parameters **`measurement`** – String to be unpacked and parsed

Type `str`

Return type `dict`

`Keithley580.`**`parse_status_word`**`(`*statusword*`)`

Parse the status word returned by the function *`get_status_word`*.

Returns a `dict` with the following keys: `{drive,polarity,drycircuit,operate,range,relative,eoi,t` `sqrondata,sqronerror,linefreq,terminator}`

Parameters **`statusword`** – Byte string to be unpacked and parsed

Type str

Return type dict

Keithley580.**query**(*msg*, *size=-1*)

Keithley580.**sendcmd**(*msg*)

Keithley580.**set_calibration_value**(*value*)
Sets the calibration value. This is not currently implemented.

Parameters **value** – Calibration value to write

Keithley580.**store_calibration_constants**()
Instructs the instrument to store the calibration constants. This is not currently implemented.

Keithley580.**trigger**()
Tell the Keithley 580 to execute all commands that it has received.

Do note that this is different from the standard SCPI *TRG command (which is not supported by the 580 anyways).

Keithley580.**drive**
Gets/sets the instrument drive to either pulsed or DC.

Example use:

```
>>> import instruments as ik
>>> keithley = ik.keithley.Keithley580.open_gpibusb('/dev/ttyUSB0', 1)
>>> keithley.drive = keithley.Drive.pulsed
```

Type *Keithley580.Drive*

Keithley580.**dry_circuit_test**
Gets/sets the 'dry circuit test' mode of the Keithley 580.

This mode is used to minimize any physical and electrical changes in the contact junction by limiting the maximum source voltage to 20mV. By limiting the voltage, the measuring circuit will leave the resistive surface films built up on the contacts undisturbed. This allows for measurement of the resistance of these films.

See the Keithley 580 manual for more information.

Type bool

Keithley580.**input_range**
Gets/sets the range of the Keithley 580 input terminals. The valid ranges are one of {AUTO|2e-1|2|20|200|2000|2e4|2e5}

Type Quantity or str

Keithley580.**operate**
Gets/sets the operating mode of the Keithley 580. If set to true, the instrument will be in operate mode, while false sets the instruments into standby mode.

Type bool

Keithley580.**polarity**
Gets/sets instrument polarity.

Example use:

```
>>> import instruments as ik
>>> keithley = ik.keithley.Keithley580.open_gpibusb('/dev/ttyUSB0', 1)
>>> keithley.polarity = keithley.Polarity.positive
```

Type *Keithley580.Polarity*

Keithley580.**relative**

Gets/sets the relative measurement mode of the Keithley 580.

As stated in the manual: The relative function is used to establish a baseline reading. This reading is subtracted from all subsequent readings. The purpose of making relative measurements is to cancel test lead and offset resistances or to store an input as a reference level.

Once a relative level is established, it remains in effect until another relative level is set. The relative value is only good for the range the value was taken on and higher ranges. If a lower range is selected than that on which the relative was taken, inaccurate results may occur. Relative cannot be activated when "OL" is displayed.

See the manual for more information.

Type bool

Keithley580.**trigger_mode**

Gets/sets the trigger mode of the Keithley 580.

There are two different trigger settings for three different sources. This means there are six different settings for the trigger mode.

The two types are continuous and one-shot. Continuous has the instrument continuously sample the resistance. One-shot performs a single resistance measurement.

The three trigger sources are on talk, on GET, and on "X". On talk refers to addressing the instrument to talk over GPIB. On GET is when the instrument receives the GPIB command byte for "group execute trigger". Last, on "X" is when one sends the ASCII character "X" to the instrument. This character is used as a general execute to confirm commands send to the instrument. In InstrumentKit, "X" is sent after each command so it is not suggested that one uses on "X" triggering.

Type *Keithley580.TriggerMode*

2.6.3 Keithley2182 Nano-voltmeter

class instruments.keithley.**Keithley2182**(*filelike*)

The Keithley 2182 is a nano-voltmeter. You can find the full specifications list in the user's guide.

Example usage:

```
>>> import instruments as ik
>>> meter = ik.keithley.Keithley2182.open_gpibusb("/dev/ttyUSB0", 10)
>>> print meter.measure(meter.Mode.voltage_dc)
```

class Channel(*parent*, *idx*)

Class representing a channel on the Keithley 2182 nano-voltmeter.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *Keithley2182* class.

measure (*mode=None*)
Performs a measurement of the specified channel. If no mode parameter is specified then the current mode is used.

Parameters mode (`Keithley2182.Mode`) – Mode that the measurement will be performed in

Returns The value of the measurement

Return type `Quantity`

input_range

mode

relative

trigger_mode

class `Keithley2182.`**Mode**
Enum containing valid measurement modes for the Keithley 2182

temperature = <Mode.temperature: 'TEMP'>

voltage_dc = <Mode.voltage_dc: 'VOLT'>

class `Keithley2182.`**TriggerMode**
Enum containing valid trigger modes for the Keithley 2182

bus = <TriggerMode.bus: 'BUS'>

external = <TriggerMode.external: 'EXT'>

immediate = <TriggerMode.immediate: 'IMM'>

manual = <TriggerMode.manual: 'MAN'>

timer = <TriggerMode.timer: 'TIM'>

`Keithley2182.`**fetch**()
Transfer readings from instrument memory to the output buffer, and thus to the computer. If currently taking a reading, the instrument will wait until it is complete before executing this command. Readings are NOT erased from memory when using fetch. Use the `R?` command to read and erase data. Note that the data is transfered as ASCII, and thus it is not recommended to transfer a large number of data points using GPIB.

Returns Measurement readings from the instrument output buffer.

Return type `list` of `Quantity` elements

`Keithley2182.`**measure**(*mode=None*)
Perform and transfer a measurement of the desired type.

Parameters mode – Desired measurement mode. If left at default the measurement will occur with the current mode.

Type *`Keithley2182.Mode`*

Returns Returns a single shot measurement of the specified mode.

Return type `Quantity`

Units Volts, Celsius, Kelvin, or Fahrenheit

`Keithley2182.`**channel**
Gets a specific Keithley 2182 channel object. The desired channel is specified like one would access a list.

Although not default, the 2182 has up to two channels.

For example, the following would print the measurement from channel 1:

```
>>> meter = ik.keithley.Keithley2182.open_gpibusb("/dev/ttyUSB0", 10)
>>> print meter.channel[0].measure()
```

Return type Keithley2182.Channel

Keithley2182.**input_range**

Keithley2182.**relative**

Gets/sets the relative measurement function of the Keithley 2182.

This is used to enable or disable the relative function for the currently set mode. When enabling, the current reading is used as a baseline which is subtracted from future measurements.

If relative is already on, the stored value is refreshed with the currently read value.

See the manual for more information.

Type bool

Keithley2182.**units**

Gets the current measurement units of the instrument.

Return type UnitQuantity

2.6.4 Keithley6220 Constant Current Supply

class instruments.keithley.**Keithley6220**(*filelike*)

The Keithley 6220 is a single channel constant current supply.

Because this is a constant current supply, most features that a regular power supply have are not present on the 6220.

Example usage:

```
>>> import quantities as pq
>>> import instruments as ik
>>> ccs = ik.keithley.Keithley6220.open_gpibusb("/dev/ttyUSB0", 10)
>>> ccs.current = 10 * pq.milliamp # Sets current to 10mA
>>> ccs.disable() # Turns off the output and sets the current to 0A
```

disable()

Set the output current to zero and disable the output.

channel

For most power supplies, this would return a channel specific object. However, the 6220 only has a single channel, so this function simply returns a tuple containing itself. This is for compatibility reasons if a multichannel supply is replaced with the single-channel 6220.

For example, the following commands are the same and both set the current to 10mA:

```
>>> ccs.channel[0].current = 0.01
>>> ccs.current = 0.01
```

current

Gets/sets the output current of the source. Value must be between -105mA and +105mA.

Units As specified, or assumed to be A otherwise.

Type float or Quantity

current_max

current_min

voltage
This property is not supported by the Keithley 6220.

2.6.5 Keithley6514 Electrometer

class instruments.keithley.**Keithley6514**(*filelike*)
The Keithley 6514 is an electrometer capable of doing sensitive current, charge, voltage and resistance measurements.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> dmm = ik.keithley.Keithley6514.open_gpibusb('/dev/ttyUSB0', 12)
```

class ArmSource
Enum containing valid trigger arming sources for the Keithley 6514

bus = <ArmSource.bus: 'BUS'>

immediate = <ArmSource.immediate: 'IMM'>

manual = <ArmSource.manual: 'MAN'>

nstest = <ArmSource.nstest: 'NST'>

pstest = <ArmSource.pstest: 'PST'>

stest = <ArmSource.stest: 'STES'>

timer = <ArmSource.timer: 'TIM'>

tlink = <ArmSource.tlink: 'TLIN'>

class Keithley6514.**Mode**
Enum containing valid measurement modes for the Keithley 6514

charge = <Mode.charge: 'CHAR'>

current = <Mode.current: 'CURR:DC'>

resistance = <Mode.resistance: 'RES'>

voltage = <Mode.voltage: 'VOLT:DC'>

class Keithley6514.**TriggerMode**
Enum containing valid trigger modes for the Keithley 6514

immediate = <TriggerMode.immediate: 'IMM'>

tlink = <TriggerMode.tlink: 'TLINK'>

class Keithley6514.**ValidRange**
Enum containing valid measurement ranges for the Keithley 6514

charge = <ValidRange.charge: (2e-08, 2e-07, 2e-06, 2e-05)>

current = <ValidRange.current: (2e-11, 2e-10, 2e-09, 2e-08, 2e-07, 2e-06, 2e-05, 0.0002, 0.002, 0.02)>

resistance = <ValidRange.resistance: (2000.0, 20000.0, 200000.0, 2000000.0, 20000000.0, 200000000.0, 200000000

voltage = <ValidRange.voltage: (2, 20, 200)>

`Keithley6514.`**`auto_config`**`(`*mode*`)`

This command causes the device to do the following:

- Switch to the specified mode
- Reset all related controls to default values
- Set trigger and arm to the 'immediate' setting
- Set arm and trigger counts to 1
- Set trigger delays to 0
- Place unit in idle state
- Disable all math calculations
- Disable buffer operation
- Enable autozero

`Keithley6514.`**`fetch`**`()`

Request the latest post-processed readings using the current mode. (So does not issue a trigger) Returns a tuple of the form (reading, timestamp)

`Keithley6514.`**`read_measurements`**`()`

Trigger and acquire readings using the current mode. Returns a tuple of the form (reading, timestamp)

`Keithley6514.`**`arm_source`**

Gets/sets the arm source of the Keithley 6514.

`Keithley6514.`**`auto_range`**

Gets/sets the auto range setting

Type `bool`

`Keithley6514.`**`input_range`**

Gets/sets the upper limit of the current range.

Type `Quantity`

`Keithley6514.`**`mode`**

Gets/sets the measurement mode of the Keithley 6514.

`Keithley6514.`**`trigger_mode`**

Gets/sets the trigger mode of the Keithley 6514.

`Keithley6514.`**`unit`**

`Keithley6514.`**`zero_check`**

Gets/sets the zero checking status of the Keithley 6514.

`Keithley6514.`**`zero_correct`**

Gets/sets the zero correcting status of the Keithley 6514.

2.7 Lakeshore

2.7.1 `Lakeshore340` Cryogenic Temperature Controller

class `instruments.lakeshore.`**`Lakeshore340`**`(`*filelike*`)`

The Lakeshore340 is a multi-sensor cryogenic temperature controller.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> inst = ik.lakeshore.Lakeshore340.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(inst.sensor[0].temperature)
>>> print(inst.sensor[1].temperature)
```

class Sensor (*parent*, *idx*)

Class representing a sensor attached to the Lakeshore 340.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *Lakeshore340* class.

temperature

Gets the temperature of the specified sensor.

Units Kelvin

Type Quantity

Lakeshore340.**sensor**

Gets a specific sensor object. The desired sensor is specified like one would access a list.

For instance, this would query the temperature of the first sensor:

```
>>> bridge = Lakeshore340.open_serial("COM5")
>>> print(bridge.sensor[0].temperature)
```

The Lakeshore 340 supports up to 2 sensors (index 0-1).

Return type *Sensor*

2.7.2 Lakeshore370 AC Resistance Bridge

class instruments.lakeshore.**Lakeshore370** (*filelike*)

The Lakeshore 370 is a multichannel AC resistance bridge for use in low temperature dilution refridgerator setups.

Example usage:

```
>>> import instruments as ik
>>> bridge = ik.lakeshore.Lakeshore370.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(bridge.channel[0].resistance)
```

class Channel (*parent*, *idx*)

Class representing a sensor attached to the Lakeshore 370.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *Lakeshore370* class.

resistance

Gets the resistance of the specified sensor.

Units Ohm

Return type Quantity

Lakeshore370.**channel**

Gets a specific channel object. The desired channel is specified like one would access a list.

For instance, this would query the resistance of the first channel:

```
>>> import instruments as ik
>>> bridge = ik.lakeshore.Lakeshore370.open_serial("COM5")
>>> print(bridge.channel[0].resistance)
```

The Lakeshore 370 supports up to 16 channels (index 0-15).

Return type `Channel`

2.7.3 `Lakeshore475` Gaussmeter

class `instruments.lakeshore.`**`Lakeshore475`**(*filelike*)

The Lakeshore475 is a DSP Gaussmeter with field ranges from 35mG to 350kG.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> gm = ik.lakeshore.Lakeshore475.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(gm.field)
>>> gm.field_units = pq.tesla
>>> gm.field_setpoint = 0.05 * pq.tesla
```

class **`Filter`**

Enum containing valid filter modes for the Lakeshore 475

`lowpass` = **<Filter.lowpass: 3>**

`narrow` = **<Filter.narrow: 2>**

`wide` = **<Filter.wide: 1>**

class `Lakeshore475.`**`Mode`**

Enum containing valid measurement modes for the Lakeshore 475

`dc` = **<Mode.dc: 1>**

`peak` = **<Mode.peak: 3>**

`rms` = **<Mode.rms: 2>**

class `Lakeshore475.`**`PeakDisplay`**

Enum containing valid peak displays for the Lakeshore 475

`both` = **<PeakDisplay.both: 3>**

`negative` = **<PeakDisplay.negative: 2>**

`positive` = **<PeakDisplay.positive: 1>**

class `Lakeshore475.`**`PeakMode`**

Enum containing valid peak modes for the Lakeshore 475

`periodic` = **<PeakMode.periodic: 1>**

`pulse` = **<PeakMode.pulse: 2>**

`Lakeshore475.`**`change_measurement_mode`**(*mode*, *resolution*, *filter_type*, *peak_mode*, *peak_disp*)

Change the measurement mode of the Gaussmeter.

Parameters

- **`mode`** (`Lakeshore475.Mode`) – The desired measurement mode.

- **resolution** (int) – Digit resolution of the measured field. One of {3|4|5}.
- **filter_type** (*Lakeshore475.Filter*) – Specify the signal filter used by the instrument. Available types include wide band, narrow band, and low pass.
- **peak_mode** (*Lakeshore475.PeakMode*) – Peak measurement mode to be used.
- **peak_disp** (*Lakeshore475.PeakDisplay*) – Peak display mode to be used.

Lakeshore475.**control_mode**
Gets/sets the control mode setting. False corresponds to the field control ramp being disables, while True enables the closed loop PI field control.

Type bool

Lakeshore475.**control_slope_limit**
Gets/sets the I value for the field control ramp.

Units As specified (if a Quantity) or assumed to be of units volt / minute.

Type Quantity

Lakeshore475.**field**
Read field from connected probe.

Type Quantity

Lakeshore475.**field_control_params**
Gets/sets the parameters associated with the field control ramp. These are (in this order) the P, I, ramp rate, and control slope limit.

Type tuple of 2 float and 2 Quantity

Lakeshore475.**field_setpoint**
Gets/sets the final setpoint of the field control ramp.

Units As specified (if a Quantity) or assumed to be of units Gauss.

Type Quantity with units Gauss

Lakeshore475.**field_units**
Gets/sets the units of the Gaussmeter.

Acceptable units are Gauss, Tesla, Oersted, and Amp/meter.

Type UnitQuantity

Lakeshore475.**i_value**
Gets/sets the I value for the field control ramp.

Type float

Lakeshore475.**p_value**
Gets/sets the P value for the field control ramp.

Type float

Lakeshore475.**ramp_rate**
Gets/sets the ramp rate value for the field control ramp.

Units As specified (if a Quantity) or assumed to be of current field units / minute.

Type Quantity

Lakeshore475.**temp_units**
Gets/sets the temperature units of the Gaussmeter.

Acceptable units are celcius and kelvin.

Type `UnitQuantity`

2.8 Newport

2.8.1 `NewportESP301` Motor Controller

class `instruments.newport.`**`NewportESP301`**(*filelike*)

Handles communication with the Newport ESP-301 multiple-axis motor controller using the protocol documented in the user's guide.

Due to the complexity of this piece of equipment, and relative lack of documentation and following of normal SCPI guidelines, this class more than likely contains bugs and non-complete behaviour.

`define_program`(**args*, ***kwds*)

Erases any existing programs with a given program ID and instructs the device to record the commands within this `with` block to be saved as a program with that ID.

For instance:

```
>>> controller = NewportESP301.open_serial("COM3")
>>> with controller.define_program(15):
...     controller.axis[0].move(0.001, absolute=False)
...
>>> controller.run_program(15)
```

Parameters **`program_id`** (`int`) – An integer label for the new program. Must be in `range(1, 101)`.

`execute_bulk_command`(**args*, ***kwds*)

Context manager to execute multiple commands in a single communication with device

Example:

```
with self.execute_bulk_command():
    execute commands as normal...
```

Parameters **`errcheck`** (*`bool`*) – Boolean to check for errors after each command that is sent to the instrument.

`reset`()

Causes the device to perform a hardware reset. Note that this method is only effective if the watchdog timer is enabled by the physical jumpers on the ESP-301. Please see the user's guide for more information.

`run_program`(*program_id*)

Runs a previously defined user program with a given program ID.

Parameters **`program_id`** (`int`) – ID number for previously saved user program

`search_for_home`(*axis=1*, *search_mode=0*, *errcheck=True*)

Searches the specified axis for home using the method specified by `search_mode`.

Parameters

- **`axis`** (`int`) – Axis ID for which home should be searched for. This value is 1-based indexing.

- **search_mode** (NewportESP301HomeSearchMode) – Method to detect when Home has been found.
- **errcheck** (*bool*) – Boolean to check for errors after each command that is sent to the instrument.

axis
Gets the axes of the motor controller as a sequence. For instance, to move along a given axis:

```
>>> controller = NewportESP301.open_serial("COM3")
>>> controller.axis[0].move(-0.001, absolute=False)
```

Note that the axes are numbered starting from zero, so that Python idioms can be used more easily. This is not the same convention used in the Newport ESP-301 user's manual, and so care must be taken when converting examples.

Type *NewportESP301Axis*

class instruments.newport.**NewportESP301Axis** (*controller*, *axis_id*)
Encapsulates communication concerning a single axis of an ESP-301 controller. This class should not be instantiated by the user directly, but is returned by *NewportESP301.axis*.

abort_motion ()
Abort motion

disable ()
Turns motor axis off.

enable ()
Turns motor axis on.

get_status ()

Returns Dictionary containing values: 'units' 'position' 'desired_position' 'desired_velocity' 'is_motion_done'

Return type dict

move (*position*, *absolute=True*, *wait=False*, *block=False*)

Parameters

- **position** (float or Quantity) – Position to set move to along this axis.
- **absolute** (*bool*) – If True, the position pos is interpreted as relative to the zero-point of the encoder. If False, pos is interpreted as relative to the current position of this axis.
- **wait** (*bool*) – If True, will tell axis to not execute other commands until movement is finished
- **block** (*bool*) – If True, will block code until movement is finished

move_indefinitely ()
Move until told to stop

move_to_hardware_limit ()
move to hardware travel limit

read_setup ()

Returns dictionary containing: 'units' 'motor_type' 'feedback_configuration' 'full_step_resolution' 'position_display_resolution' 'current' 'max_velocity' 'encoder_resolution' 'acceleration' 'deceleration' 'velocity' 'max_acceleration' 'homing_velocity' 'jog_high_velocity' 'jog_low_velocity' 'estop_deceleration' 'jerk' 'proportional_gain' 'derivative_gain' 'integral_gain' 'integral_saturation_gain' 'home' 'microstep_factor' 'acceleration_feed_forward' 'trajectory' 'hardware_limit_configuration'

Return type dict of `quantities.Quantity`, float and int

search_for_home(*search_mode=0*)

Searches this axis only for home using the method specified by `search_mode`.

Parameters **search_mode** (`NewportESP301HomeSearchMode`) – Method to detect when Home has been found.

setup_axis(***kwargs*)

Setup a non-newport DC servo motor stage. Necessary parameters are.

- 'motor_type' = type of motor see 'QM' in Newport documentation
- 'current' = motor maximum current (A)
- 'voltage' = motor voltage (V)
- 'units' = set units (see NewportESP301Units)(U)
- 'encoder_resolution' = value of encoder step in terms of (U)
- 'max_velocity' = maximum velocity (U/s)
- 'max_base_velocity' = maximum working velocity (U/s)
- 'homing_velocity' = homing speed (U/s)
- 'jog_high_velocity' = jog high speed (U/s)
- 'jog_low_velocity' = jog low speed (U/s)
- 'max_acceleration' = maximum acceleration (U/s^2)
- 'acceleration' = acceleration (U/s^2)
- 'deceleration' = set deceleration (U/s^2)
- 'error_threshold' = set error threshold (U)
- 'proportional_gain' = PID proportional gain (optional)
- 'derivative_gain' = PID derivative gain (optional)
- 'interal_gain' = PID internal gain (optional)
- 'integral_saturation_gain' = PID integral saturation (optional)
- 'trajectory' = trajectory mode (optional)
- 'position_display_resolution' (U per step)
- 'feedback_configuration'
- 'full_step_resolution' = (U per step)
- 'home' = (U)
- 'acceleration_feed_forward' = bewtween 0 to 2e9
- 'reduce_motor_torque' = (time(ms),percentage)

stop_motion()

Stop all motion on axis. With programmed deceleration rate

wait_for_motion (*poll_interval=0.01*, *max_wait=None*)
Blocks until all movement along this axis is complete, as reported by *is_motion_done*.

Parameters

- **poll_interval** (float) – How long (in seconds) to sleep between checking if the motion is complete.
- **max_wait** (float) – Maximum amount of time to wait before raising a IOError. If None, this method will wait indefinitely.

wait_for_position (*position*)
Wait for axis to reach position before executing next command

Parameters **position** (float or Quantity) – Position to wait for on axis

wait_for_stop ()
Waits for axis motion to stop before next command is executed

acceleration
Gets/sets the axis acceleration

Units As specified (if a Quantity) or assumed to be of current newport unit

Type Quantity or float

acceleration_feed_forward
Gets/sets the axis acceleration_feed_forward setting

Type int

axis_id
Get axis number of Newport Controller

Type int

current
Gets/sets the axis current (amps)

Units As specified (if a Quantity) or assumed to be of current newport A

Type Quantity or float

deceleration
Gets/sets the axis deceleration

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s^2}$

Type Quantity or float

derivative_gain
Gets/sets the axis derivative_gain

Type float

desired_position
Gets desired position on axis in units

Units As specified (if a Quantity) or assumed to be of current newport unit

Type Quantity or float

desired_velocity
Gets the axis desired velocity in unit/s

Units As specified (if a Quantity) or assumed to be of current newport unit/s

Type `Quantity` or `float`

encoder_position
Gets the encoder position

Type

encoder_resolution
Gets/sets the resolution of the encode. The minimum number of units per step. Encoder functionality must be enabled.

Units The number of units per encoder step

Type `Quantity` or `float`

error_threshold
Gets/sets the axis error threshold

Units units

Type `Quantity` or `float`

estop_deceleration
Gets/sets the axis estop deceleration

Units As specified (if a `Quantity`) or assumed to be of current newport $\frac{unit}{s^2}$

Type `Quantity` or float

feedback_configuration
Gets/sets the axis Feedback configuration

Type `int`

full_step_resolution
Gets/sets the axis resolution of the encode. The minimum number of units per step. Encoder functionality must be enabled.

Units The number of units per encoder step

Type `Quantity` or `float`

hardware_limit_configuration
Gets/sets the axis hardware_limit_configuration

Type `int`

home
Gets/sets the axis home position. Default should be 0 as that sets current position as home

Units As specified (if a `Quantity`) or assumed to be of current newport unit

Type `Quantity` or `float`

homing_velocity
Gets/sets the axis homing velocity

Units As specified (if a `Quantity`) or assumed to be of current newport $\frac{unit}{s}$

Type `Quantity` or `float`

integral_gain
Gets/sets the axis integral_gain

Type `float`

integral_saturation_gain
Gets/sets the axis integral_saturation_gain

Type float

is_motion_done
True if and only if all motion commands have completed. This method can be used to wait for a motion command to complete before sending the next command.

Type bool

jerk
Gets/sets the jerk rate for the controller

Units As specified (if a Quantity) or assumed to be of current newport unit

Type Quantity or float

jog_high_velocity
Gets/sets the axis jog high velocity

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s}$

Type Quantity or float

jog_low_velocity
Gets/sets the axis jog low velocity

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s}$

Type Quantity or float

left_limit
Gets/sets the axis left travel limit

Units The limit in units

Type Quantity or float

max_acceleration
Gets/sets the axis max acceleration

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s^2}$

Type Quantity or float

max_base_velocity
Gets/sets the maximum base velocity for stepper motors

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s}$

Type Quantity or float

max_deceleration
Gets/sets the axis max decceleration. Max deaceleration is always the same as acceleration.

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s^2}$

Type Quantity or float

max_velocity
Gets/sets the axis maximum velocity

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s}$

Type Quantity or float

micro_inch = UnitQuantity('micro-inch', 1e-06 * in, 'uin')

microstep_factor
Gets/sets the axis microstep_factor

Type int

motor_type
Gets/sets the axis motor type * 0 = undefined * 1 = DC Servo * 2 = Stepper motor * 3 = commutated stepper motor * 4 = commutated brushless servo motor

Type int

Return type NewportESP301MotorType

position
Gets real position on axis in units

Units As specified (if a Quantity) or assumed to be of current newport unit

Type Quantity or float

position_display_resolution
Gets/sets the position display resolution

Type int

proportional_gain
Gets/sets the axis proportional_gain

Type float

right_limit
Gets/sets the axis right travel limit

Units units

Type Quantity or float

trajectory
Gets/sets the axis trajectory

Type int

units
Get the units that all commands are in reference to.

Type Quantity with units corresponding to units of axis connected or int which corresponds to Newport unit number

velocity
Gets/sets the axis velocity

Units As specified (if a Quantity) or assumed to be of current newport $\frac{unit}{s}$

Type Quantity or float

voltage
Gets/sets the axis voltage

Units As specified (if a Quantity) or assumed to be of current newport V

Type Quantity or float

class instruments.newport.**NewportESP301HomeSearchMode**
Enum containing different search modes code

home_index_signals = <NewportESP301HomeSearchMode.home_index_signals: 1>

home_signal_only = <NewportESP301HomeSearchMode.home_signal_only: 2>

neg_index_signals = <NewportESP301HomeSearchMode.neg_index_signals: 6>

neg_limit_signal = <NewportESP301HomeSearchMode.neg_limit_signal: 4>

pos_index_signals = <NewportESP301HomeSearchMode.pos_index_signals: 5>

pos_limit_signal = <NewportESP301HomeSearchMode.pos_limit_signal: 3>

zero_position_count = <NewportESP301HomeSearchMode.zero_position_count: 0>

2.8.2 NewportError

class instruments.newport.**NewportError**(*errcode=None*, *timestamp=None*)

Raised in response to an error with a Newport-brand instrument.

static get_message(*code*)

Returns the error string for a given error code

Parameters **code** (*str*) – Error code as returned by instrument

Returns Full error code string

Return type str

axis

Gets the axis with which this error is concerned, or None if the error was not associated with any particlar axis.

Type int

errcode

Gets the error code reported by the device.

Type int

messageDict = {'x29': 'DIGITAL I/O INTERLOCK DETECTED', 'x32': 'INVALID TRAJECTORY MODE FOR MO

start_time = datetime.datetime(2016, 6, 9, 21, 38, 9, 313307)

timestamp

Geturns the timestamp reported by the device as the time at which this error occured.

Type datetime

2.9 Other Instruments

2.9.1 NewportESP301

2.9.2 PhaseMatrixFSW0020

Units

Units are identified to the Phase Matrix FSW-0020 using the Quantity class implemented by the quantities package. To support the FSW-0020, we provide several additional unit quantities, listed here.

2.10 Oxford

2.10.1 OxfordITC503 Temperature Controller

class instruments.oxford.**OxfordITC503** (*filelike*)

The Oxford ITC503 is a multi-sensor temperature controller.

Example usage:

```
>>> import instruments as ik
>>> itc = ik.oxford.OxfordITC503.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(itc.sensor[0].temperature)
>>> print(itc.sensor[1].temperature)
```

class Sensor (*parent*, *idx*)

Class representing a probe sensor on the Oxford ITC 503.

Warning: This class should NOT be manually created by the user. It is designed to be initialized by the *OxfordITC503* class.

temperature

Read the temperature of the attached probe to the specified channel.

Units Kelvin

Type Quantity

OxfordITC503.**sensor**

Gets a specific sensor object. The desired sensor is specified like one would access a list.

For instance, this would query the temperature of the first sensor:

```
>>> itc = ik.oxford.OxfordITC503.open_gpibusb('/dev/ttyUSB0', 1)
>>> print(itc.sensor[0].temperature)
```

Type *OxfordITC503.Sensor*

2.11 PhaseMatrix

2.11.1 PhaseMatrixFSW0020 Signal Generator

class instruments.phasematrix.**PhaseMatrixFSW0020** (*filelike*)

Communicates with a Phase Matrix FSW-0020 signal generator via the "Native SPI" protocol, supported on all FSW firmware versions.

Example:

```
>>> import instruments as ik
>>> import quantities as pq
>>> inst = ik.phasematrix.PhaseMatrixFSW0020.open_serial("/dev/ttyUSB0", baud=115200)
>>> inst.frequency = 1 * pq.GHz
>>> inst.power = 0 * ik.units.dBm  # Can omit units and will assume dBm
>>> inst.output = True
```

reset ()

Causes the connected signal generator to perform a hardware reset. Note that no commands will be accepted by the generator for at least 5μs.

am_modulation
Gets/sets the amplitude modulation status of the FSW0020

Type bool

blanking
Gets/sets the blanking status of the FSW0020

Type bool

frequency
Gets/sets the output frequency of the signal generator. If units are not specified, the frequency is assumed to be in gigahertz (GHz).

Type Quantity

Units frequency, assumed to be GHz

output
Gets/sets the channel output status of the FSW0020. Setting this property to True will turn the output on.

Type bool

phase

power
Gets/sets the output power of the signal generator. If units are not specified, the power is assumed to be in decibel-milliwatts (dBm).

Type Quantity

Units log-power, assumed to be dBm

pulse_modulation
Gets/sets the pulse modulation status of the FSW0020

Type bool

ref_output
Gets/sets the reference output status of the FSW0020

Type bool

2.12 Picowatt

2.12.1 PicowattAVS47 Resistance Bridge

class instruments.picowatt.**PicowattAVS47**(*filelike*)
The Picowatt AVS 47 is a resistance bridge used to measure the resistance of low-temperature sensors.

Example usage:

```
>>> import instruments as ik
>>> bridge = ik.picowatt.PicowattAVS47.open_gpibusb('/dev/ttyUSB0', 1)
>>> print bridge.sensor[0].resistance
```

class InputSource
Enum containing valid input source modes for the AVS 47

actual = <InputSource.actual: 1>

ground = <InputSource.ground: 0>

reference = <InputSource.reference: 2>

class `PicowattAVS47.`**`Sensor`**(*parent*, *idx*)

Class representing a sensor on the PicowattAVS47

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *PicowattAVS47* class.

`resistance`

Gets the resistance. It first ensures that the next measurement reading is up to date by first sending the "ADC" command.

Units Ω (ohms)

Return type `Quantity`

`PicowattAVS47.`**`display`**

Gets/sets the sensor that is displayed on the front panel.

Valid display sensor values are 0 through 7 (inclusive).

Type `int`

`PicowattAVS47.`**`excitation`**

Gets/sets the excitation sensor number.

Valid excitation sensor values are 0 through 7 (inclusive).

Type `int`

`PicowattAVS47.`**`input_source`**

Gets/sets the input source.

Type *PicowattAVS47.InputSource*

`PicowattAVS47.`**`mux_channel`**

Gets/sets the multiplexer sensor number. It is recommended that you ground the input before switching the multiplexer channel.

Valid mux channel values are 0 through 7 (inclusive).

Type `int`

`PicowattAVS47.`**`remote`**

Gets/sets the remote mode state.

Enabling the remote mode allows all settings to be changed by computer interface and locks-out the front panel.

Type `bool`

`PicowattAVS47.`**`sensor`**

Gets a specific sensor object. The desired sensor is specified like one would access a list.

Return type *Sensor*

See also:

PicowattAVS47 for an example using this property.

2.13 Qubitekk

2.13.1 CC1 Coincidence Counter

class instruments.qubitekk.**CC1** (*filelike*)

The CC1 is a hand-held coincidence counter.

It has two setting values, the dwell time and the coincidence window. The coincidence window determines the amount of time (in ns) that the two detections may be from each other and still be considered a coincidence. The dwell time is the amount of time that passes before the counter will send the clear signal.

More information can be found at : http://www.qubitekk.com

class Channel (*cc1*, *idx*)

Class representing a channel on the Qubitekk CC1.

count

Gets the counts of this channel.

Return type int

CC1.**clear_counts** ()

Clears the current total counts on the counters.

CC1.**acknowledge**

Gets/sets the acknowledge message state. If True, the CC1 will echo back every command sent, then print the response (either Unable to comply, Unknown command or the response to a query). If False, the CC1 will only print the response.

Units None

Type boolean

CC1.**channel**

Gets a specific channel object. The desired channel is specified like one would access a list.

For instance, this would print the counts of the first channel:

```
>>> cc = ik.qubitekk.CC1.open_serial('COM8', 19200, timeout=1)
>>> print(cc.channel[0].count)
```

Return type *CC1.Channel*

CC1.**delay**

Get/sets the delay value (in nanoseconds) on Channel 1.

When setting, N may be 0, 2, 4, 6, 8, 10, 12, or 14ns.

Return type quantities.ns

Returns the delay value

CC1.**dwell_time**

Gets/sets the length of time before a clear signal is sent to the counters.

Units As specified (if a Quantity) or assumed to be of units seconds.

Type Quantity

CC1.**firmware**

Gets the firmware version

Return type tuple\`(Major:\`int, Minor:int, Patch‘int‘)

CC1.**gate**
Gets/sets the gate enable status

Type bool

CC1.**subtract**
Gets/sets the subtract enable status

Type bool

CC1.**trigger_mode**
Gets/sets the trigger mode setting for the CC1. This can be set to `continuous` or `start/stop` modes.

Type CC1.TriggerMode

CC1.**window**
Gets/sets the length of the coincidence window between the two signals.

Units As specified (if a `Quantity`) or assumed to be of units nanoseconds.

Type Quantity

2.14 Rigol

2.14.1 RigolDS1000Series Oscilloscope

class instruments.rigol.**RigolDS1000Series**(*filelike*)
The Rigol DS1000-series is a popular budget oriented oscilloscope that has featured wide adoption across hobbyist circles.

> **Warning:** This instrument is not complete, and probably not even functional!

class AcquisitionType
Enum containing valid acquisition types for the Rigol DS1000

average = <AcquisitionType.average: 'AVER'>

normal = <AcquisitionType.normal: 'NORM'>

peak_detect = <AcquisitionType.peak_detect: 'PEAK'>

class RigolDS1000Series.**Channel**(*parent*, *idx*)
Class representing a channel on the Rigol DS1000.

This class inherits from *DataSource*.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *RigolDS1000Series* class.

query(*cmd*)
Passes a command from the `Channel` class to the parent *RigolDS1000Series*, appending the required channel identification.

Parameters cmd (*str*) – The command string to send to the instrument
Returns The result as returned by the instrument
Return type str

sendcmd(*cmd*)
Passes a command from the `Channel` class to the parent *RigolDS1000Series*, appending the required channel identification.

Parameters **cmd** (*str*) – The command string to send to the instrument

bw_limit

coupling

display

filter

invert

vernier

class RigolDS1000Series.**Coupling**
Enum containing valid coupling modes for the Rigol DS1000

ac = <Coupling.ac: 'AC'>

dc = <Coupling.dc: 'DC'>

ground = <Coupling.ground: 'GND'>

class RigolDS1000Series.**DataSource**(*parent*, *name*)
Class representing a data source (channel, math, or ref) on the Rigol DS1000

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *RigolDS1000Series* class.

read_waveform(*bin_format=True*)

name

RigolDS1000Series.**force_trigger**()

RigolDS1000Series.**release_panel**()
Releases any lockout of the local control panel.

RigolDS1000Series.**run**()
Starts running the oscilloscope trigger.

RigolDS1000Series.**stop**()
Stops running the oscilloscope trigger.

RigolDS1000Series.**acquire_averages**
Gets/sets the number of averages the oscilloscope should take per acquisition.

Type int

RigolDS1000Series.**acquire_type**

RigolDS1000Series.**channel**

RigolDS1000Series.**math**

RigolDS1000Series.**panel_locked**

RigolDS1000Series.**ref**

2.15 Stanford Research Systems

2.15.1 SRS345 Function Generator

class instruments.srs.**SRS345**(*filelike*)

The SRS DS345 is a 30MHz function generator.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> srs = ik.srs.SRS345.open_gpib('/dev/ttyUSB0', 1)
>>> srs.frequency = 1 * pq.MHz
>>> print(srs.offset)
>>> srs.function = srs.Function.triangle
```

class Function

Enum containing valid output function modes for the SRS 345

arbitrary = <Function.arbitrary: 5>

noise = <Function.noise: 4>

ramp = <Function.ramp: 3>

sinusoid = <Function.sinusoid: 0>

square = <Function.square: 1>

triangle = <Function.triangle: 2>

SRS345.**frequency**

Gets/sets the output frequency.

Units As specified, or assumed to be Hz otherwise.

Type float or Quantity

SRS345.**function**

Gets/sets the output function of the function generator.

Type *Function*

SRS345.**offset**

Gets/sets the offset voltage for the output waveform.

Units As specified, or assumed to be V otherwise.

Type float or Quantity

SRS345.**phase**

Gets/sets the phase for the output waveform.

Units As specified, or assumed to be degrees (°) otherwise.

Type float or Quantity

2.15.2 SRS830 Lock-In Amplifier

class instruments.srs.**SRS830**(*filelike*, *outx_mode=None*)

Communicates with a Stanford Research Systems 830 Lock-In Amplifier.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> srs = ik.srs.SRS830.open_gpibusb('/dev/ttyUSB0', 1)
>>> srs.frequency = 1000 * pq.hertz # Lock-In frequency
>>> data = srs.take_measurement(1, 10) # 1Hz sample rate, 10 samples total
```

class BufferMode

Enum for the SRS830 buffer modes.

loop = <BufferMode.loop: 1>

one_shot = <BufferMode.one_shot: 0>

class SRS830.**Coupling**

Enum for the SRS830 channel coupling settings.

ac = <Coupling.ac: 0>

dc = <Coupling.dc: 1>

class SRS830.**FreqSource**

Enum for the SRS830 frequency source settings.

external = <FreqSource.external: 0>

internal = <FreqSource.internal: 1>

class SRS830.**Mode**

Enum containing valid modes for the SRS 830

aux1 = <Mode.aux1: 'aux1'>

aux2 = <Mode.aux2: 'aux2'>

aux3 = <Mode.aux3: 'aux3'>

aux4 = <Mode.aux4: 'aux4'>

ch1 = <Mode.ch1: 'ch1'>

ch2 = <Mode.ch2: 'ch2'>

none = <Mode.none: 'none'>

r = <Mode.r: 'r'>

ref = <Mode.ref: 'ref'>

theta = <Mode.theta: 'theta'>

x = <Mode.x: 'x'>

xnoise = <Mode.xnoise: 'xnoise'>

y = <Mode.y: 'y'>

ynoise = <Mode.ynoise: 'ynoise'>

SRS830.**auto_offset**(*mode*)

Sets a specific channel mode to auto offset. This is the same as pressing the auto offset key on the display.

It sets the offset of the mode specified to zero.

Parameters mode (*Mode* or str) – Target mode of auto_offset function. Valid inputs are {X|Y|R}.

SRS830.**auto_phase**()

Sets the lock-in to auto phase. This does the same thing as pushing the auto phase button.

Do not send this message again without waiting the correct amount of time for the lock-in to finish.

SRS830.**clear_data_buffer**()

Clears the data buffer of the SRS830.

SRS830.**data_snap**(*mode1*, *mode2*)

Takes a snapshot of the current parameters are defined by variables mode1 and mode2.

For combinations (X,Y) and (R,THETA), they are taken at the same instant. All other combinations are done sequentially, and may not represent values taken from the same timestamp.

Returns a list of floats, arranged in the order that they are given in the function input parameters.

Parameters

- **mode1** (*Mode* or `str`) – Mode to take data snap for channel 1. Valid inputs are given by: {X|Y|R|THETA|AUX1|AUX2|AUX3|AUX4|REF|CH1|CH2}
- **mode2** (*Mode* or `str`) – Mode to take data snap for channel 2. Valid inputs are given by: {X|Y|R|THETA|AUX1|AUX2|AUX3|AUX4|REF|CH1|CH2}

Return type `list`

SRS830.**init**(*sample_rate*, *buffer_mode*)

Wrapper function to prepare the SRS830 for measurement. Sets both the data sampling rate and the end of buffer mode

Parameters

- **sample_rate** (`Quantity` or `str`) – The desired sampling rate. Acceptable set values are 2^n where $n \in \{-4... + 9\}$ in units Hertz or the string `trigger`.
- **buffer_mode** (`SRS830.BufferMode`) – This sets the behaviour of the instrument when the data storage buffer is full. Setting to `one_shot` will stop acquisition, while `loop` will repeat from the start.

SRS830.**pause**()

Has the instrument pause data capture.

SRS830.**read_data_buffer**(*channel*)

Reads the entire data buffer for a specific channel. Transfer is done in ASCII mode. Although binary would be faster, this is not currently implemented.

Returns a list of floats containing instrument's measurements.

Parameters **channel** (*SRS830.Mode* or `str`) – Channel data buffer to read from. Valid channels are given by {CH1|CH2}.

Return type `list`

SRS830.**set_channel_display**(*channel*, *display*, *ratio*)

Sets the display of the two channels. Channel 1 can display X, R, X Noise, Aux In 1, Aux In 2 Channel 2 can display Y, Theta, Y Noise, Aux In 3, Aux In 4

Channel 1 can have ratio of None, Aux In 1, Aux In 2 Channel 2 can have ratio of None, Aux In 3, Aux In 4

Parameters

- **channel** (*Mode* or `str`) – Channel you wish to set the display of. Valid input is one of {CH1|CH2}.

- **display** (*Mode* or str) – Setting the channel will be changed to. Valid input is one of {X|Y|R|THETA|XNOISE|YNOISE|AUX1|AUX2|AUX3|AUX4}
- **ratio** (*Mode* or str) – Desired ratio setting for this channel. Valid input is one of {NONE|AUX1|AUX2|AUX3|AUX4}

SRS830.**set_offset_expand**(*mode*, *offset*, *expand*)

Sets the channel offset and expand parameters. Offset is a percentage, and expand is given as a multiplication factor of 1, 10, or 100.

Parameters

- **mode** (*SRS830.Mode* or str) – The channel mode that you wish to change the offset and/or the expand of. Valid modes are X, Y, and R.
- **offset** (float) – Offset of the mode, given as a percent. offset = <-105...+105>.
- **expand** (int) – Expansion factor for the measurement. Valid input is {1|10|100}.

SRS830.**start_data_transfer**()

Wrapper function to start the actual data transfer. Sets the transfer mode to FAST2, and triggers the data transfer to start after a delay of 0.5 seconds.

SRS830.**start_scan**()

After setting the data transfer on via the dataTransfer function, this is used to start the scan. The scan starts after a delay of 0.5 seconds.

SRS830.**take_measurement**(*sample_rate*, *num_samples*)

Wrapper function that allows you to easily take measurements with a specified sample rate and number of desired samples.

Function will call time.sleep() for the required amount of time it will take the instrument to complete this sampling operation.

Returns a list containing two items, each of which are lists containing the channel data. The order is [[Ch1 data], [Ch2 data]].

Parameters

- **sample_rate** (int) – Set the desired sample rate of the measurement. See *sample_rate* for more information.
- **num_samples** (int) – Number of samples to take.

Return type list

SRS830.**amplitude**

Gets/set the amplitude of the internal reference signal.

Set value should be 0.004 <= newval <= 5.000

Units As specified (if a Quantity) or assumed to be of units volts. Value should be specified as peak-to-peak.

Type Quantity with units volts peak-to-peak.

SRS830.**amplitude_max**

SRS830.**amplitude_min**

SRS830.**buffer_mode**

Gets/sets the end of buffer mode.

This sets the behaviour of the instrument when the data storage buffer is full. Setting to one_shot will stop acquisition, while loop will repeat from the start.

Type *SRS830.BufferMode*

SRS830.**coupling**
Gets/sets the input coupling to either 'ac' or 'dc'.

Type *SRS830.Coupling*

SRS830.**data_transfer**
Gets/sets the data transfer status.

Note that this function only makes use of 2 of the 3 data transfer modes supported by the SRS830. The supported modes are FAST0 and FAST2. The other, FAST1, is for legacy systems which this package does not support.

Type bool

SRS830.**frequency**
Gets/sets the lock-in amplifier reference frequency.

Units As specified (if a Quantity) or assumed to be of units Hertz.

Type Quantity with units Hertz.

SRS830.**frequency_source**

Gets/sets the frequency source used. This is either an external source, or uses the internal reference.

Type *SRS830.FreqSource*

SRS830.**input_shield_ground**
Function sets the input shield grounding to either 'float' or 'ground'.

Type bool

SRS830.**num_data_points**
Gets the number of data sets in the SRS830 buffer.

Type int

SRS830.**phase**
Gets/set the phase of the internal reference signal.

Set value should be -360deg <= newval < +730deg.

Units As specified (if a Quantity) or assumed to be of units degrees.

Type Quantity with units degrees.

SRS830.**phase_max**

SRS830.**phase_min**

SRS830.**sample_rate**
Gets/sets the data sampling rate of the lock-in.

Acceptable set values are 2^n where $n \in \{-4... + 9\}$ or the string trigger.

Type Quantity with units Hertz.

2.15.3 SRSCTC100 Cryogenic Temperature Controller

class instruments.srs.**SRSCTC100** (*filelike*)
Communicates with a Stanford Research Systems CTC-100 cryogenic temperature controller.

class **Channel** (*ctc*, *chan_name*)
Represents an input or output channel on an SRS CTC-100 cryogenic temperature controller.

get_log ()
Gets all of the log data points currently saved in the instrument memory.

Returns Tuple of all the log data points. First value is time, second is the measurement value.

Return type Tuple of 2x `Quantity`, each comprised of a numpy array (`numpy.dnarray`).

get_log_point (*which='next'*, *units=None*)
Get a log data point from the instrument.

Parameters

- **which** (*str*) – Which data point you want. Valid examples include `first`, and `next`. Consult the instrument manual for the complete list
- **units** (`UnitQuantity`) – Units to attach to the returned data point. If left with the value of `None` then the instrument will be queried for the current units setting.

Returns The log data point with units

Return type `Quantity`

average
Gets the average measurement for the specified channel as determined by the statistics gathering.

Type `Quantity`

name
Gets/sets the name of the channel that will be used by the instrument to identify the channel in programming and on the display.

Type `str`

sensor_type
Gets the type of sensor attached to the specified channel.

Type *SRSCTC100.SensorType*

stats_enabled
Gets/sets enabling the statistics for the specified channel.

Type `bool`

stats_points
Gets/sets the number of sample points to use for the channel statistics.

Type `int`

std_dev
Gets the standard deviation for the specified channel as determined by the statistics gathering.

Type `Quantity`

units
Gets the appropriate units for the specified channel.

Units can be one of `celsius`, `watt`, `volt`, `ohm`, or `dimensionless`.

Type `UnitQuantity`

value
Gets the measurement value of the channel. Units depend on what kind of sensor and/or channel you have specified. Units can be one of `celsius`, `watt`, `volt`, `ohm`, or `dimensionless`.

Type `Quantity`

class SRSCTC100.**SensorType**
Enum containing valid sensor types for the SRS CTC-100

diode = <SensorType.diode: 'Diode'>

rox = <SensorType.rox: 'ROX'>

rtd = <SensorType.rtd: 'RTD'>

thermistor = <SensorType.thermistor: 'Thermistor'>

SRSCTC100.**channel_units**()

Returns a dictionary from channel names to channel units, using the `getOutput.units` command. Unknown units and dimensionless quantities are presented the same way by the instrument, and so both are reported using `pq.dimensionless`.

Return type `dict` with channel names as keys and units as values

SRSCTC100.**clear_log**()

Clears the SRS CTC100 log

Not sure if this works.

SRSCTC100.**errcheck**()

Performs an error check query against the CTC100. This function does not return anything, but will raise an `IOError` if the error code received by the instrument is not zero.

Returns Nothing

SRSCTC100.**query**(*cmd*, *size=-1*)

SRSCTC100.**sendcmd**(*cmd*)

SRSCTC100.**channel**

Gets a specific measurement channel on the SRS CTC100. This is accessed like one would access a `dict`. Here you must use the actual channel names to address a specific channel. This is different from most other instruments in InstrumentKit because the CRC100 channel names can change by the user.

The list of current valid channel names can be accessed by the `SRSCTC100._channel_names()` function.

Type *SRSCTC100.Channel*

SRSCTC100.**display_figures**

Gets/sets the number of significant figures to display. Valid range is 0-6 inclusive.

Type `int`

SRSCTC100.**error_check_toggle**

Gets/sets if errors should be checked for after every command.

Bool

2.15.4 SRSDG645 Digital Delay Generator

class instruments.srs.**SRSDG645**(*filelike*)

Communicates with a Stanford Research Systems DG645 digital delay generator, using the SCPI commands documented in the user's guide.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> srs = ik.srs.SRSDG645.open_gpibusb('/dev/ttyUSB0', 1)
>>> srs.channel["B"].delay = (srs.channel["A"], pq.Quantity(10, 'ns'))
>>> srs.output["AB"].level_amplitude = pq.Quantity(4.0, "V")
```

class **Channels**
Enumeration of valid delay channels for the DDG.

A = <Channels.A: 2>

B = <Channels.B: 3>

C = <Channels.C: 4>

D = <Channels.D: 5>

E = <Channels.E: 6>

F = <Channels.F: 7>

G = <Channels.G: 8>

H = <Channels.H: 9>

T0 = <Channels.T0: 0>

T1 = <Channels.T1: 1>

class SRSDG645.**DisplayMode**
Enumeration of possible modes for the physical front-panel display.

adv_triggering_enable = <DisplayMode.adv_triggering_enable: 4>

burst_T0_config = <DisplayMode.burst_T0_config: 14>

burst_count = <DisplayMode.burst_count: 9>

burst_delay = <DisplayMode.burst_delay: 8>

burst_mode = <DisplayMode.burst_mode: 7>

burst_period = <DisplayMode.burst_period: 10>

channel_delay = <DisplayMode.channel_delay: 11>

channel_levels = <DisplayMode.channel_levels: 12>

channel_polarity = <DisplayMode.channel_polarity: 13>

prescale_config = <DisplayMode.prescale_config: 6>

trigger_holdoff = <DisplayMode.trigger_holdoff: 5>

trigger_line = <DisplayMode.trigger_line: 3>

trigger_rate = <DisplayMode.trigger_rate: 0>

trigger_single_shot = <DisplayMode.trigger_single_shot: 2>

trigger_threshold = <DisplayMode.trigger_threshold: 1>

class SRSDG645.**LevelPolarity**
Polarities for output levels.

negative = <LevelPolarity.negative: 0>

positive = <LevelPolarity.positive: 1>

class SRSDG645.**Output** (*parent*, *idx*)
An output from the DDG.

level_amplitude
Amplitude (in voltage) of the output level for this output.
Type float or Quantity

Units As specified, or V by default.

polarity
Polarity of this output.

Type *SRSDG645.LevelPolarity*

class SRSDG645.**Outputs**
Enumeration of valid outputs from the DDG.

AB = <Outputs.AB: 1>

CD = <Outputs.CD: 2>

EF = <Outputs.EF: 3>

GH = <Outputs.GH: 4>

T0 = <Outputs.T0: 0>

class SRSDG645.**TriggerSource**
Enumeration of the different allowed trigger sources and modes.

external_falling = <TriggerSource.external_falling: 2>

external_rising = <TriggerSource.external_rising: 1>

internal = <TriggerSource.internal: 0>

line = <TriggerSource.line: 6>

single_shot = <TriggerSource.single_shot: 5>

ss_external_falling = <TriggerSource.ss_external_falling: 4>

ss_external_rising = <TriggerSource.ss_external_rising: 3>

SRSDG645.**channel**
Gets a specific channel object.

The desired channel is accessed by passing an EnumValue from *Channels*. For example, to access channel A:

```
>>> import instruments as ik
>>> inst = ik.srs.SRSDG645.open_gpibusb('/dev/ttyUSB0', 1)
>>> inst.channel[inst.Channels.A]
```

See the example in *SRSDG645* for a more complete example.

Return type _SRSDG645Channel

SRSDG645.**display**
Gets/sets the front-panel display mode for the connected DDG. The mode is a tuple of the display mode and the channel.

Type tuple of an *SRSDG645.DisplayMode* and an *SRSDG645.Channels*

SRSDG645.**enable_adv_triggering**
Gets/sets whether advanced triggering is enabled.

Type bool

SRSDG645.**holdoff**
Gets/sets the trigger holdoff time.

Type Quantity or float

Units As passed, or s if not specified.

SRSDG645.**output**
Gets the specified output port.

Type *SRSDG645.Output*

SRSDG645.**trigger_rate**
Gets/sets the rate of the internal trigger.

Type Quantity or float

Units As passed or Hz if not specified.

SRSDG645.**trigger_source**
Gets/sets the source for the trigger.

Type *SRSDG645.TriggerSource*

2.16 Tektronix

2.16.1 TekAWG2000 Arbitrary Wave Generator

class instruments.tektronix.**TekAWG2000**(*filelike*)
Communicates with a Tektronix AWG2000 series instrument using the SCPI commands documented in the user's guide.

class Channel(*tek*, *idx*)
Class representing a physical channel on the Tektronix AWG 2000

Warning: This class should NOT be manually created by the user. It

is designed to be initialized by the *TekAWG2000* class.

amplitude
Gets/sets the amplitude of the specified channel.
Units As specified (if a Quantity) or assumed to be of units Volts.
Type Quantity with units Volts peak-to-peak.

frequency
Gets/sets the frequency of the specified channel when using the built-in function generator.
::units: As specified (if a Quantity) or assumed to be of units Hertz.

Type Quantity with units Hertz.

name
Gets the name of this AWG channel
Type str

offset
Gets/sets the offset of the specified channel.
Units As specified (if a Quantity) or assumed to be of units Volts.
Type Quantity with units Volts.

polarity
Gets/sets the polarity of the specified channel.
Type *TekAWG2000.Polarity*

shape
Gets/sets the waveform shape of the specified channel. The AWG will use the internal function generator for these shapes.

Type *TekAWG2000.Shape*

class TekAWG2000.**Polarity**
Enum containing valid polarity modes for the AWG2000

inverted = <Polarity.inverted: 'INVERTED'>

normal = <Polarity.normal: 'NORMAL'>

class TekAWG2000.**Shape**
Enum containing valid waveform shape modes for hte AWG2000

pulse = <Shape.pulse: 'PULSE'>

ramp = <Shape.ramp: 'RAMP'>

sine = <Shape.sine: 'SINUSOID'>

square = <Shape.square: 'SQUARE'>

triangle = <Shape.triangle: 'TRIANGLE'>

TekAWG2000.**upload_waveform**(*yzero*, *ymult*, *xincr*, *waveform*)
Uploads a waveform from the PC to the instrument.

Parameters

- **yzero** (float or int) – Y-axis origin offset
- **ymult** (float or int) – Y-axis data point multiplier
- **xincr** (float or int) – X-axis data point increment
- **waveform** (*numpy.ndarray*) – Numpy array of values representing the waveform to be uploaded. This array should be normalized. This means that all absolute values contained within the array should not exceed 1.

TekAWG2000.**channel**
Gets a specific channel on the AWG2000. The desired channel is accessed like one would access a list.

Example usage:

```
>>> import instruments as ik
>>> inst = ik.tektronix.TekAWG2000.open_gpibusb("/dev/ttyUSB0", 1)
>>> print(inst.channel[0].frequency)
```

Returns A channel object for the AWG2000

Return type *TekAWG2000.Channel*

TekAWG2000.**waveform_name**
Gets/sets the destination waveform name for upload.

This is the file name that will be used on the AWG for any following waveform data that is uploaded.

Type str

2.16.2 TekDPO4104 Oscilloscope

class instruments.tektronix.**TekDPO4104**(*filelike*)
The Tektronix DPO4104 is a multi-channel oscilloscope with analog bandwidths ranging from 100MHz to 1GHz.

This class inherits from *SCPIInstrument*.

Example usage:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekDPO4104.open_tcpip("192.168.0.2", 8888)
>>> [x, y] = tek.channel[0].read_waveform()
```

class Coupling

Enum containing valid coupling modes for the channels on the Tektronix DPO 4104

ac = <Coupling.ac: 'AC'>

dc = <Coupling.dc: 'DC'>

ground = <Coupling.ground: 'GND'>

TekDPO4104.**force_trigger**()

Forces a trigger event to occur on the attached oscilloscope. Note that this is distinct from the standard SCPI `*TRG` functionality.

TekDPO4104.**aquisition_continuous**

Gets/sets whether the aquisition is continuous ("run/stop mode") or whether aquisiton halts after the next sequence ("single mode").

Type `bool`

TekDPO4104.**aquisition_length**

Gets/sets the aquisition length of the oscilloscope

Type `int`

TekDPO4104.**aquisition_running**

Gets/sets the aquisition state of the attached instrument. This property is `True` if the aquisition is running, and is `False` otherwise.

Type `bool`

TekDPO4104.**channel**

Gets a specific oscilloscope channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> tek = ik.tektronix.TekDPO4104.open_tcpip("192.168.0.2", 8888)
>>> [x, y] = tek.channel[0].read_waveform()
```

Return type *_TekDPO4104Channel*

TekDPO4104.**data_source**

Gets/sets the the data source for waveform transfer.

TekDPO4104.**data_width**

Gets/sets the data width (number of bytes wide per data point) for waveforms transfered to/from the oscilloscope.

Valid widths are 1 or 2.

Type `int`

TekDPO4104.**math**

Gets a data source object corresponding to the MATH channel.

Return type *_TekDPO4104DataSource*

TekDPO4104.**ref**

Gets a specific oscilloscope reference channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekDPO4104.open_tcpip("192.168.0.2", 8888)
>>> [x, y] = tek.ref[0].read_waveform()
```

Return type *_TekDPO4104DataSource*

TekDPO4104.**y_offset**

Gets/sets the Y offset of the currently selected data source.

class instruments.tektronix.**_TekDPO4104DataSource** (*tek*, *name*)

Class representing a data source (channel, math, or ref) on the Tektronix DPO 4104.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *TekDPO4104* class.

read_waveform (*bin_format=True*)

Read waveform from the oscilloscope. This function is all inclusive. After reading the data from the oscilloscope, it unpacks the data and scales it accordingly. Supports both ASCII and binary waveform transfer.

Function returns a tuple (x,y), where both x and y are numpy arrays.

Parameters **bin_format** (*bool*) – If True, data is transfered in a binary format. Otherwise, data is transferred in ASCII.

name

Gets the name of this data source, as identified over SCPI.

Type str

y_offset

class instruments.tektronix.**_TekDPO4104Channel** (*parent*, *idx*)

Class representing a channel on the Tektronix DPO 4104.

This class inherits from *_TekDPO4104DataSource*.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *TekDPO4104* class.

coupling

Gets/sets the coupling setting for this channel.

Type *TekDPO4104.Coupling*

2.16.3 TekDPO70000 Oscilloscope

class instruments.tektronix.**TekDPO70000** (*filelike*)

The Tektronix DPO70000 series is a multi-channel oscilloscope with analog bandwidths ranging up to 33GHz.

This class inherits from *SCPIInstrument*.

Example usage:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekDPO70000.open_tcpip("192.168.0.2", 8888)
>>> [x, y] = tek.channel[0].read_waveform()
```

class AcquisitionMode
Enum containing valid acquisition modes for the Tektronix 70000 series oscilloscopes.

average = <AcquisitionMode.average: 'AVE'>

envelope = <AcquisitionMode.envelope: 'ENV'>

hi_res = <AcquisitionMode.hi_res: 'HIR'>

peak_detect = <AcquisitionMode.peak_detect: 'PEAK'>

sample = <AcquisitionMode.sample: 'SAM'>

waveform_db = <AcquisitionMode.waveform_db: 'WFMDB'>

class TekDPO70000.**AcquisitionState**
Enum containing valid acquisition states for the Tektronix 70000 series oscilloscopes.

off = <AcquisitionState.off: 'OFF'>

on = <AcquisitionState.on: 'ON'>

run = <AcquisitionState.run: 'RUN'>

stop = <AcquisitionState.stop: 'STOP'>

class TekDPO70000.**BinaryFormat**
Enum containing valid binary formats for the Tektronix 70000 series oscilloscopes (int, unsigned-int, floating-point).

float = <BinaryFormat.float: 'FP'>

int = <BinaryFormat.int: 'RI'>

uint = <BinaryFormat.uint: 'RP'>

class TekDPO70000.**ByteOrder**
Enum containing valid byte order (big-/little-endian) for the Tektronix 70000 series oscilloscopes.

big_endian = <ByteOrder.big_endian: 'MSB'>

little_endian = <ByteOrder.little_endian: 'LSB'>

class TekDPO70000.**Channel** (*parent*, *idx*)
Class representing a channel on the Tektronix DPO 70000.

This class inherits from *TekDPO70000.DataSource*.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *TekDPO70000* class.

class Coupling
Enum containing valid coupling modes for the oscilloscope channel

ac = <Coupling.ac: 'AC'>

dc = <Coupling.dc: 'DC'>

dc_reject = <Coupling.dc_reject: 'DCREJ'>

ground = <Coupling.ground: 'GND'>

`TekDPO70000.Channel.`**`query`**`(`*cmd, size=-1*`)`
Wraps queries sent from property factories in this class with identifiers for the specified channel.

Parameters

- **cmd** (`str`) – Query command to send to the instrument
- **size** (`int`) – Number of characters to read from the response. Default value reads until a termination character is found.

Returns The query response

Return type `str`

`TekDPO70000.Channel.`**`sendcmd`**`(`*cmd*`)`
Wraps commands sent from property factories in this class with identifiers for the specified channel.

Parameters **cmd** (`str`) – Command to send to the instrument

`TekDPO70000.Channel.`**`bandwidth`**

`TekDPO70000.Channel.`**`coupling`**
Gets/sets the coupling for the specified channel.

Example usage:

```
>>> import instruments as ik
>>> inst = ik.tektronix.TekDPO70000.open_tcpip("192.168.0.1", 8080)
>>> channel = inst.channel[0]
>>> channel.coupling = channel.Coupling.ac
```

`TekDPO70000.Channel.`**`deskew`**

`TekDPO70000.Channel.`**`label`**
Just a human readable label for the channel.

`TekDPO70000.Channel.`**`label_xpos`**
The x position, in divisions, to place the label.

`TekDPO70000.Channel.`**`label_ypos`**
The y position, in divisions, to place the label.

`TekDPO70000.Channel.`**`offset`**
The vertical offset in units of volts. Voltage is given by `offset+scale*(5*raw/2^15 - position)`.

`TekDPO70000.Channel.`**`position`**
The vertical position, in divisions from the center graticule, ranging from `-8` to `8`. Voltage is given by `offset+scale*(5*raw/2^15 - position)`.

`TekDPO70000.Channel.`**`scale`**
Vertical channel scale in units volts/division. Voltage is given by `offset+scale*(5*raw/2^15 - position)`.

`TekDPO70000.Channel.`**`termination`**

class `TekDPO70000.`**`DataSource`**`(`*parent, name*`)`
Class representing a data source (channel, math, or ref) on the Tektronix DPO 70000.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *`TekDPO70000`* class.

`read_waveform``(`*bin_format=True*`)`

`name`

class TekDPO70000.**HorizontalMode**

Enum containing valid horizontal scan modes for the Tektronix 70000 series oscilloscopes.

auto = <HorizontalMode.auto: 'AUTO'>

constant = <HorizontalMode.constant: 'CONST'>

manual = <HorizontalMode.manual: 'MAN'>

class TekDPO70000.**Math**(*parent*, *idx*)

Class representing a math channel on the Tektronix DPO 70000.

This class inherits from *TekDPO70000.DataSource*.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *TekDPO70000* class.

class FilterMode

Enum containing valid filter modes for a math channel on the TekDPO70000 series oscilloscope.

centered = <FilterMode.centered: 'CENT'>

shifted = <FilterMode.shifted: 'SHIF'>

class TekDPO70000.Math.**Mag**

Enum containing valid amplitude units for a math channel on the TekDPO70000 series oscilloscope.

db = <Mag.db: 'DB'>

dbm = <Mag.dbm: 'DBM'>

linear = <Mag.linear: 'LINEA'>

class TekDPO70000.Math.**Phase**

Enum containing valid phase units for a math channel on the TekDPO70000 series oscilloscope.

degrees = <Phase.degrees: 'DEG'>

group_delay = <Phase.group_delay: 'GROUPD'>

radians = <Phase.radians: 'RAD'>

class TekDPO70000.Math.**SpectralWindow**

Enum containing valid spectral windows for a math channel on the TekDPO70000 series oscilloscope.

blackman_harris = <SpectralWindow.blackman_harris: 'BLACKMANH'>

flattop2 = <SpectralWindow.flattop2: 'FLATTOP2'>

gaussian = <SpectralWindow.gaussian: 'GAUSS'>

hamming = <SpectralWindow.hamming: 'HAMM'>

hanning = <SpectralWindow.hanning: 'HANN'>

kaiser_besse = <SpectralWindow.kaiser_besse: 'KAISERB'>

rectangular = <SpectralWindow.rectangular: 'RECTANG'>

tek_exponential = <SpectralWindow.tek_exponential: 'TEKEXP'>

TekDPO70000.Math.**query**(*cmd*, *size=-1*)

Wraps queries sent from property factories in this class with identifiers for the specified math channel.

Parameters

- **cmd** (*str*) – Query command to send to the instrument

- **size** (int) – Number of characters to read from the response. Default value reads until a termination character is found.

Returns The query response

Return type str

TekDPO70000.Math.**sendcmd**(*cmd*)
Wraps commands sent from property factories in this class with identifiers for the specified math channel.

Parameters **cmd** (*str*) – Command to send to the instrument

TekDPO70000.Math.**autoscale**
Enables or disables the auto-scaling of new math waveforms.

TekDPO70000.Math.**define**
A text string specifying the math to do, ex. CH1+CH2

TekDPO70000.Math.**filter_mode**

TekDPO70000.Math.**filter_risetime**

TekDPO70000.Math.**label**
Just a human readable label for the channel.

TekDPO70000.Math.**label_xpos**
The x position, in divisions, to place the label.

TekDPO70000.Math.**label_ypos**
The y position, in divisions, to place the label.

TekDPO70000.Math.**num_avg**
The number of acquisistions over which exponential averaging is performed.

TekDPO70000.Math.**position**
The vertical position, in divisions from the center graticule.

TekDPO70000.Math.**scale**
The scale in volts per division. The range is from `100e-36` to `100e+36`.

TekDPO70000.Math.**spectral_center**
The desired frequency of the spectral analyzer output data span in Hz.

TekDPO70000.Math.**spectral_gatepos**
The gate position. Units are represented in seconds, with respect to trigger position.

TekDPO70000.Math.**spectral_gatewidth**
The time across the 10-division screen in seconds.

TekDPO70000.Math.**spectral_lock**

TekDPO70000.Math.**spectral_mag**
Whether the spectral magnitude is linear, db, or dbm.

TekDPO70000.Math.**spectral_phase**
Whether the spectral phase is degrees, radians, or group delay.

TekDPO70000.Math.**spectral_reflevel**
The value that represents the topmost display screen graticule. The units depend on spectral_mag.

TekDPO70000.Math.**spectral_reflevel_offset**

TekDPO70000.Math.**spectral_resolution_bandwidth**
The desired resolution bandwidth value. Units are represented in Hertz.

TekDPO70000.Math.**spectral_span**
Specifies the frequency span of the output data vector from the spectral analyzer.

TekDPO70000.Math.**spectral_suppress**
The magnitude level that data with magnitude values below this value are displayed as zero phase.

TekDPO70000.Math.**spectral_unwrap**
Enables or disables phase wrapping.

TekDPO70000.Math.**spectral_window**

TekDPO70000.Math.**threshhold**
The math threshhold in volts

TekDPO70000.Math.**unit_string**
Just a label for the units...doesn"t actually change anything.

class TekDPO70000.**SamplingMode**
Enum containing valid sampling modes for the Tektronix 70000 series oscilloscopes.

equivalent_time_allowed = <SamplingMode.equivalent_time_allowed: 'ET'>

interpolation_allowed = <SamplingMode.interpolation_allowed: 'IT'>

real_time = <SamplingMode.real_time: 'RT'>

class TekDPO70000.**StopAfter**
Enum containing valid stop condition modes for the Tektronix 70000 series oscilloscopes.

run_stop = <StopAfter.run_stop: 'RUNST'>

sequence = <StopAfter.sequence: 'SEQ'>

class TekDPO70000.**TriggerState**
Enum containing valid trigger states for the Tektronix 70000 series oscilloscopes.

armed = <TriggerState.armed: 'ARMED'>

auto = <TriggerState.auto: 'AUTO'>

dpo = <TriggerState.dpo: 'DPO'>

partial = <TriggerState.partial: 'PARTIAL'>

ready = <TriggerState.ready: 'READY'>

class TekDPO70000.**WaveformEncoding**
Enum containing valid waveform encoding modes for the Tektronix 70000 series oscilloscopes.

ascii = <WaveformEncoding.ascii: 'ASCII'>

binary = <WaveformEncoding.binary: 'BINARY'>

TekDPO70000.**force_trigger**()
Forces a trigger event to happen for the oscilloscope.

TekDPO70000.**run**()
Enables the trigger for the oscilloscope.

TekDPO70000.**select_fastest_encoding**()
Sets the encoding for data returned by this instrument to be the fastest encoding method consistent with the current data source.

TekDPO70000.**stop**()
Disables the trigger for the oscilloscope.

TekDPO70000.**HOR_DIVS = 10**

TekDPO70000.**VERT_DIVS = 10**

TekDPO70000.**acquire_enhanced_enob**
Valid values are AUTO and OFF.

TekDPO70000.**acquire_enhanced_state**

TekDPO70000.**acquire_interp_8bit**
Valid values are AUTO, ON and OFF.

TekDPO70000.**acquire_magnivu**

TekDPO70000.**acquire_mode**

TekDPO70000.**acquire_mode_actual**

TekDPO70000.**acquire_num_acquisitions**
The number of waveform acquisitions that have occurred since starting acquisition with the ACQuire:STATE RUN command

TekDPO70000.**acquire_num_avgs**
The number of waveform acquisitions to average.

TekDPO70000.**acquire_num_envelop**
The number of waveform acquisitions to be enveloped

TekDPO70000.**acquire_num_frames**
The number of frames acquired when in FastFrame Single Sequence and acquisitions are running.

TekDPO70000.**acquire_num_samples**
The minimum number of acquired samples that make up a waveform database (WfmDB) waveform for single sequence mode and Mask Pass/Fail Completion Test. The default value is 16,000 samples. The range is 5,000 to 2,147,400,000 samples.

TekDPO70000.**acquire_sampling_mode**

TekDPO70000.**acquire_state**
This command starts or stops acquisitions.

TekDPO70000.**acquire_stop_after**
This command sets or queries whether the instrument continually acquires acquisitions or acquires a single sequence.

TekDPO70000.**channel**

TekDPO70000.**data_framestart**

TekDPO70000.**data_framestop**

TekDPO70000.**data_source**
Gets/sets the data source for the oscilloscope. This will return the actual Channel/Math/DataSource object as if it was accessed through the usual *TekDPO70000.channel*, *TekDPO70000.math*, or *TekDPO70000.ref* properties.

Type *TekDPO70000.Channel* or *TekDPO70000.Math*

TekDPO70000.**data_start**
The first data point that will be transferred, which ranges from 1 to the record length.

TekDPO70000.**data_stop**
The last data point that will be transferred.

TekDPO70000.**data_sync_sources**

TekDPO70000.**horiz_acq_duration**
The duration of the acquisition.

TekDPO70000.**horiz_acq_length**
The record length.

TekDPO70000.**horiz_delay_mode**

TekDPO70000.**horiz_delay_pos**
The percentage of the waveform that is displayed left of the center graticule.

TekDPO70000.**horiz_delay_time**
The base trigger delay time setting.

TekDPO70000.**horiz_interp_ratio**
The ratio of interpolated points to measured points.

TekDPO70000.**horiz_main_pos**
The percentage of the waveform that is displayed left of the center graticule.

TekDPO70000.**horiz_mode**

TekDPO70000.**horiz_pos**
The position of the trigger point on the screen, left is 0%, right is 100%.

TekDPO70000.**horiz_record_length**
The recond length in samples. See *horiz_mode*; manual mode lets you change the record length, while the length is readonly for auto and constant mode.

TekDPO70000.**horiz_record_length_lim**
The recond length limit in samples.

TekDPO70000.**horiz_roll**
Valid arguments are AUTO, OFF, and ON.

TekDPO70000.**horiz_sample_rate**
The sample rate in samples per second.

TekDPO70000.**horiz_scale**
The horizontal scale in seconds per division. The horizontal scale is readonly when *horiz_mode* is manual.

TekDPO70000.**horiz_unit**

TekDPO70000.**math**

TekDPO70000.**outgoing_binary_format**
Controls the data type of samples when transferring waveforms from the instrument to the host using binary encoding.

TekDPO70000.**outgoing_byte_order**
Controls whether binary data is returned in little or big endian.

TekDPO70000.**outgoing_n_bytes**
The number of bytes per sample used in representing outgoing waveforms in binary encodings.

Must be either 1, 2, 4 or 8.

TekDPO70000.**outgoing_waveform_encoding**
Controls the encoding used for outgoing waveforms (instrument → host).

TekDPO70000.**ref**

TekDPO70000.**trigger_state**

2.16.4 TekTDS224 Oscilloscope

class instruments.tektronix.**TekTDS224** (*filelike*)

The Tektronix TDS224 is a multi-channel oscilloscope with analog bandwidths of 100MHz.

This class inherits from *SCPIInstrument*.

Example usage:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekTDS224.open_gpibusb("/dev/ttyUSB0", 1)
>>> [x, y] = tek.channel[0].read_waveform()
```

class Coupling

Enum containing valid coupling modes for the Tek TDS224

ac = <Coupling.ac: 'AC'>

dc = <Coupling.dc: 'DC'>

ground = <Coupling.ground: 'GND'>

TekTDS224.**channel**

Gets a specific oscilloscope channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekTDS224.open_tcpip('192.168.0.2', 8888)
>>> [x, y] = tek.channel[0].read_waveform()
```

Return type _TekTDS224Channel

TekTDS224.**data_source**

Gets/sets the the data source for waveform transfer.

TekTDS224.**data_width**

Gets/sets the byte-width of the data points being returned by the instrument. Valid widths are 1 or 2.

Type int

TekTDS224.**force_trigger**

TekTDS224.**math**

Gets a data source object corresponding to the MATH channel.

Return type _TekTDS224DataSource

TekTDS224.**ref**

Gets a specific oscilloscope reference channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> import instruments as ik
>>> tek = ik.tektronix.TekTDS224.open_tcpip('192.168.0.2', 8888)
>>> [x, y] = tek.ref[0].read_waveform()
```

Return type _TekTDS224DataSource

2.16.5 `TekTDS5xx` Oscilloscope

class `instruments.tektronix.`**`TekTDS5xx`**(*filelike*)

Support for the TDS5xx series of oscilloscopes

Implemented from:

TDS Family Digitizing Oscilloscopes
(TDS 410A, 420A, 460A, 520A, 524A, 540A, 544A,
620A, 640A, 644A, 684A, 744A & 784A)
Tektronix Document: 070-8709-07

class **`Bandwidth`**
Bandwidth in MHz

`FULL` = <Bandwidth.FULL: 'FUL'>

`OneHundred` = <Bandwidth.OneHundred: 'HUN'>

`Twenty` = <Bandwidth.Twenty: 'TWE'>

`TwoHundred` = <Bandwidth.TwoHundred: 'TWO'>

class `TekTDS5xx.`**`Coupling`**
Available coupling options for input sources and trigger

`ac` = <Coupling.ac: 'AC'>

`dc` = <Coupling.dc: 'DC'>

`ground` = <Coupling.ground: 'GND'>

class `TekTDS5xx.`**`Edge`**
Available Options for trigger slope

`Falling` = <Edge.Falling: 'FALL'>

`Rising` = <Edge.Rising: 'RIS'>

class `TekTDS5xx.`**`Impedance`**
Available options for input source impedance

`Fifty` = <Impedance.Fifty: 'FIF'>

`OneMeg` = <Impedance.OneMeg: 'MEG'>

class `TekTDS5xx.`**`Source`**
Available Data sources

`CH1` = <Source.CH1: 'CH1'>

`CH2` = <Source.CH2: 'CH2'>

`CH3` = <Source.CH3: 'CH3'>

`CH4` = <Source.CH4: 'CH4'>

`Math1` = <Source.Math1: 'MATH1'>

`Math2` = <Source.Math2: 'MATH2'>

`Math3` = <Source.Math3: 'MATH3'>

`Ref1` = <Source.Ref1: 'REF1'>

`Ref2` = <Source.Ref2: 'REF2'>

Ref3 = <Source.Ref3: ‘REF3’>

Ref4 = <Source.Ref4: ‘REF4’>

class TekTDS5xx.**Trigger**

Available Trigger sources (AUX not Available on TDS520A/TDS540A)

AUX = <Trigger.AUX: ‘AUX’>

CH1 = <Trigger.CH1: ‘CH1’>

CH2 = <Trigger.CH2: ‘CH2’>

CH3 = <Trigger.CH3: ‘CH3’>

CH4 = <Trigger.CH4: ‘CH4’>

LINE = <Trigger.LINE: ‘LINE’>

TekTDS5xx.**get_hardcopy**()

Gets a screenshot of the display

Return type string

TekTDS5xx.**channel**

Gets a specific oscilloscope channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> tek = ik.tektronix.TekTDS5xx.open_tcpip('192.168.0.2', 8888)
>>> [x, y] = tek.channel[0].read_waveform()
```

Return type _TekTDS5xxChannel

TekTDS5xx.**clock**

Get/Set oscilloscope clock

Type datetime.datetime

TekTDS5xx.**data_source**

Gets/sets the the data source for waveform transfer.

Type *TekTDS5xx.Source* or _TekTDS5xxDataSource

Return type ‘_TekTDS5xxDataSource‘

TekTDS5xx.**data_width**

Gets/Sets the data width for waveform transfers

Type int

TekTDS5xx.**display_clock**

Get/Set the visibility of clock on the display

Type bool

TekTDS5xx.**force_trigger**

TekTDS5xx.**horizontal_scale**

Get/Set Horizontal Scale

Type float

TekTDS5xx.**math**

Gets a data source object corresponding to the MATH channel.

Return type _TekTDS5xxDataSource

TekTDS5xx.**measurement**
Gets a specific oscilloscope measurement object. The desired channel is specified like one would access a list.

Return type _TDS5xxMeasurement

TekTDS5xx.**ref**
Gets a specific oscilloscope reference channel object. The desired channel is specified like one would access a list.

For instance, this would transfer the waveform from the first channel:

```
>>> tek = ik.tektronix.TekTDS5xx.open_tcpip('192.168.0.2', 8888)
>>> [x, y] = tek.ref[0].read_waveform()
```

Return type _TekTDS5xxDataSource

TekTDS5xx.**sources**
Returns list of all active sources

Return type list

TekTDS5xx.**trigger_coupling**
Get/Set trigger coupling

Type *TekTDS5xx.Coupling*

TekTDS5xx.**trigger_level**
Get/Set trigger level

Type float

TekTDS5xx.**trigger_slope**
Get/Set trigger slope

Type *TekTDS5xx.Edge*

TekTDS5xx.**trigger_source**
Get/Set trigger source

Type *TekTDS5xx.Trigger*

2.17 ThorLabs

2.17.1 PM100USB USB Power Meter

class instruments.thorlabs.**PM100USB**(*filelike*)
Instrument class for the ThorLabs PM100USB power meter. Note that as this is an SCPI-compliant instrument, the properties and methods of *SCPIInstrument* may be used as well.

class MeasurementConfiguration
Enum containing valid measurement modes for the PM100USB

current = <MeasurementConfiguration.current: 'CURR'>

energy = <MeasurementConfiguration.energy: 'ENER'>

energy_density = <MeasurementConfiguration.energy_density: 'EDEN'>

frequency = <MeasurementConfiguration.frequency: 'FREQ'>

power = <MeasurementConfiguration.power: 'POW'>

power_density = <MeasurementConfiguration.power_density: 'PDEN'>

resistance = <MeasurementConfiguration.resistance: 'RES'>

temperature = <MeasurementConfiguration.temperature: 'TEMP'>

voltage = <MeasurementConfiguration.voltage: 'VOLT'>

class PM100USB.**Sensor**(*parent*)

Class representing a sensor on the ThorLabs PM100USB

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *PM100USB* class.

calibration_message

Gets the calibration message of the sensor channel

Type str

flags

Gets any sensor flags set on the sensor channel

Type collections.namedtuple

name

Gets the name associated with the sensor channel

Type str

serial_number

Gets the serial number of the sensor channel

Type str

type

Gets the sensor type of the sensor channel

Type str

class PM100USB.**SensorFlags**

Enum containing valid sensor flags for the PM100USB

has_temperature_sensor = <SensorFlags.has_temperature_sensor: 256>

is_energy_sensor = <SensorFlags.is_energy_sensor: 2>

is_power_sensor = <SensorFlags.is_power_sensor: 1>

response_settable = <SensorFlags.response_settable: 16>

tau_settable = <SensorFlags.tau_settable: 64>

wavelength_settable = <SensorFlags.wavelength_settable: 32>

PM100USB.**read**(*size=-1*)

Reads a measurement from this instrument, according to its current configuration mode.

Parameters **size** (int) – Number of bytes to read from the instrument. Default of -1 reads until a termination character is found.

Units As specified by *measurement_configuration*.

Return type Quantity

PM100USB.**averaging_count**

Integer specifying how many samples to collect and average over for each measurement, with each sample taking approximately 3 ms.

PM100USB.**cache_units**

If enabled, then units are not checked every time a measurement is made, reducing by half the number of round-trips to the device.

> **Warning:** Setting this to True may cause incorrect values to be returned, if any commands are sent to the device either by its local panel, or by software other than InstrumentKit.

Type bool

PM100USB.**flag = <SensorFlags.has_temperature_sensor: 256>**

PM100USB.**measurement_configuration**

Returns the current measurement configuration.

Return type *PM100USB.MeasurementConfiguration*

PM100USB.**sensor**

Returns information about the currently connected sensor.

Type *PM100USB.Sensor*

2.17.2 ThorLabsAPT ThorLabs APT Controller

class instruments.thorlabs.**ThorLabsAPT**(*filelike*)

Generic ThorLabs APT hardware device controller. Communicates using the ThorLabs APT communications protocol, whose documentation is found in the thorlabs source folder.

class APTChannel(*apt*, *idx_chan*)

Represents a channel within the hardware device. One device can have many channels, each labeled by an index.

enabled

Gets/sets the enabled status for the specified APT channel

Type bool

ThorLabsAPT.**identify**()

Causes a light on the APT instrument to blink, so that it can be identified.

ThorLabsAPT.**channel**

Gets the list of channel objects attached to the APT controller.

A specific channel object can then be accessed like one would access a list.

Type tuple of *APTChannel*

ThorLabsAPT.**destination**

Gets the destination for the APT controller

Type int

ThorLabsAPT.**model_number**

Gets the model number for the APT controller

Type str

ThorLabsAPT.**n_channels**

Gets/sets the number of channels attached to the APT controller

Type int

ThorLabsAPT.**name**

Gets the name of the APT controller. This is a human readable string containing the model, serial number, hardware version, and firmware version.

Type str

ThorLabsAPT.**serial_number**

Gets the serial number for the APT controller

Type str

class instruments.thorlabs.**APTPiezoStage**(*filelike*)

Class representing a Thorlabs APT piezo stage

class **PiezoChannel**(*apt*, *idx_chan*)

Class representing a single piezo channel within a piezo stage on the Thorlabs APT controller.

change_position_control_mode(*closed*, *smooth=True*)

Changes the position control mode of the piezo channel

Parameters

- **closed** (*bool*) – True for closed, False for open
- **smooth** (*bool*) – True for smooth, False for otherwise. Default is True.

output_position

Gets/sets the output position for the piezo channel.

Type str

position_control_closed

Gets the status if the position control is closed or not.

True means that the position control is closed, False otherwise

Tyep bool

class instruments.thorlabs.**APTStrainGaugeReader**(*filelike*)

Class representing a Thorlabs APT strain gauge reader.

Warning: This is not currently implemented

class **StrainGaugeChannel**(*apt*, *idx_chan*)

Class representing a single strain gauge channel attached to a *APTStrainGaugeReader* on the Thorlabs APT controller.

Warning: This is not currently implemented

class instruments.thorlabs.**APTMotorController**(*filelike*)

Class representing a Thorlabs APT motor controller

class **MotorChannel**(*apt*, *idx_chan*)

Class representing a single motor attached to a Thorlabs APT motor controller (*APTMotorController*).

go_home()

Instructs the specified motor channel to return to its home position

move(*pos*, *absolute=True*)

Instructs the specified motor channel to move to a specific location. The provided position can be either an absolute or relative position.

Parameters

- **pos** (Quantity) – The position to move to. Provided value will be converted to encoder counts.

- **absolute** (*bool*) – Specify if the position is a relative or absolute position. `True` means absolute, while `False` is for a relative move.

Units pos As specified, or assumed to of units encoder counts

set_scale (*motor_model*)

Sets the scale factors for this motor channel, based on the model of the attached motor and the specifications of the driver of which this is a channel.

Parameters motor_model (*str*) – Name of the model of the attached motor, as indicated in the APT protocol documentation (page 14, v9).

position

Gets the current position of the specified motor channel

Type `Quantity`

position_encoder

Gets the position of the encoder of the specified motor channel

Type `Quantity`

Units Encoder `counts`

scale_factors = (array(1) * dimensionless, array(1) * dimensionless, array(1) * dimensionless)

status_bits

Gets the status bits for the specified motor channel.

Type `dict`

2.17.3 SC10 Optical Beam Shutter Controller

class `instruments.thorlabs.`**SC10** (*filelike*)

The SC10 is a shutter controller, to be used with the Thorlabs SH05 and SH1. The user manual can be found here: http://www.thorlabs.com/thorcat/8600/SC10-Manual.pdf

class Mode

Enum containing valid output modes of the SC10

auto = <Mode.auto: 2>

external = <Mode.external: 5>

manual = <Mode.manual: 1>

repeat = <Mode.repeat: 4>

single = <Mode.single: 3>

`SC10.`**default** ()

Restores instrument to factory settings.

Returns 1 if successful, zero otherwise.

Return type `int`

`SC10.`**restore** ()

Loads the settings from memory.

Returns 1 if successful, zero otherwise.

Return type `int`

`SC10.`**save** ()

Stores the parameters in static memory

Returns 1 if successful, zero otherwise.

Return type `int`

`SC10.save_mode()`

Stores output trigger mode and baud rate settings in memory.

Returns 1 if successful, zero otherwise.

Return type `int`

`SC10.baud_rate`

Gets/sets the instrument baud rate.

Valid baud rates are 9600 and 115200.

Type `int`

`SC10.closed`

Gets the shutter closed status.

`True` represents the shutter is closed, and `False` for the shutter is open.

Return type `bool`

`SC10.enable`

Gets/sets the shutter enable status, False for disabled, True if enabled

If output enable is on (`True`), there is a voltage on the output.

Return type `bool`

`SC10.interlock`

Gets the interlock tripped status.

Returns `True` if the interlock is tripped, and `False` otherwise.

Return type `bool`

`SC10.mode`

Gets/sets the output mode of the SC10

Return type *`SC10.Mode`*

`SC10.name`

Gets the name and version number of the device.

Returns Name and verison number of the device

Return type `str`

`SC10.open_time`

Gets/sets the amount of time that the shutter is open, in ms

Units As specified (if a `Quantity`) or assumed to be of units milliseconds.

Type `Quantity`

`SC10.out_trigger`

Gets/sets the out trigger source.

0 trigger out follows shutter output, 1 trigger out follows controller output

Type `int`

`SC10.repeat`

Gets/sets the repeat count for repeat mode. Valid range is [1,99] inclusive.

Type `int`

SC10.**shut_time**

Gets/sets the amount of time that the shutter is closed, in ms

Units As specified (if a `Quantity`) or assumed to be of units milliseconds.

Type `Quantity`

SC10.**trigger**

Gets/sets the trigger source.

0 for internal trigger, 1 for external trigger

Type `int`

2.17.4 LCC25 Liquid Crystal Controller

class instruments.thorlabs.**LCC25** (*filelike*)

The LCC25 is a controller for the thorlabs liquid crystal modules. it can set two voltages and then oscillate between them at a specific repetition rate.

The user manual can be found here: http://www.thorlabs.com/thorcat/18800/LCC25-Manual.pdf

class Mode

Enum containing valid output modes of the LCC25

normal = <Mode.normal: 0>

voltage1 = <Mode.voltage1: 1>

voltage2 = <Mode.voltage2: 2>

LCC25.**default** ()

Restores instrument to factory settings.

Returns 1 if successful, 0 otherwise

Return type `int`

LCC25.**get_settings** (*slot*)

Gets the current settings to memory.

Returns 1 if successful, zero otherwise.

Parameters **slot** (`int`) – Memory slot to use, valid range `[1,4]`

Return type `int`

LCC25.**save** ()

Stores the parameters in static memory

Returns 1 if successful, zero otherwise.

Return type `int`

LCC25.**set_settings** (*slot*)

Saves the current settings to memory.

Returns 1 if successful, zero otherwise.

Parameters **slot** (`int`) – Memory slot to use, valid range `[1,4]`

Return type `int`

LCC25.**test_mode**()
Puts the LCC in test mode - meaning it will increment the output voltage from the minimum value to the maximum value, in increments, waiting for the dwell time

Returns 1 if successful, zero otherwise.

Return type int

LCC25.**dwell**
Gets/sets the dwell time for voltages for the test mode.

Units As specified (if a Quantity) or assumed to be of units milliseconds.

Return type Quantity

LCC25.**enable**
Gets/sets the output enable status.

If output enable is on (True), there is a voltage on the output.

Return type bool

LCC25.**extern**
Gets/sets the use of the external TTL modulation.

Value is True for external TTL modulation and False for internal modulation.

Return type bool

LCC25.**frequency**
Gets/sets the frequency at which the LCC oscillates between the two voltages.

Units As specified (if a Quantity) or assumed to be of units Hertz.

Return type Quantity

LCC25.**increment**
Gets/sets the voltage increment for voltages for the test mode.

Units As specified (if a Quantity) or assumed to be of units Volts.

Return type Quantity

LCC25.**max_voltage**
Gets/sets the maximum voltage value for the test mode. If the maximum voltage is less than the minimum voltage, nothing happens.

Units As specified (if a Quantity) or assumed to be of units Volts.

Return type Quantity

LCC25.**min_voltage**
Gets/sets the minimum voltage value for the test mode.

Units As specified (if a Quantity) or assumed to be of units Volts.

Return type Quantity

LCC25.**mode**
Gets/sets the output mode of the LCC25

Return type *LCC25.Mode*

LCC25.**name**
Gets the name and version number of the device

Return type str

LCC25.**remote**
Gets/sets front panel lockout status for remote instrument operation.

Value is `False` for normal operation and `True` to lock out the front panel buttons.

Return type `bool`

LCC25.**voltage1**
Gets/sets the voltage value for output 1.

Units As specified (if a `Quantity`) or assumed to be of units Volts.

Return type `Quantity`

LCC25.**voltage2**
Gets/sets the voltage value for output 2.

Units As specified (if a `Quantity`) or assumed to be of units Volts.

Return type `Quantity`

2.17.5 TC200 Temperature Controller

class instruments.thorlabs.**TC200** (*filelike*)
The TC200 is is a controller for the voltage across a heating element. It can also read in the temperature off of a thermistor and implements a PID control to keep the temperature at a set value.

The user manual can be found here: http://www.thorlabs.com/thorcat/12500/TC200-Manual.pdf

class Mode
Enum containing valid output modes of the TC200.

cycle = <Mode.cycle: 1>

normal = <Mode.normal: 0>

class TC200.**Sensor**
Enum containing valid temperature sensor types for the TC200.

ntc10k = <Sensor.ntc10k: 'ntc10k'>

ptc100 = <Sensor.ptc100: 'ptc100'>

ptc1000 = <Sensor.ptc1000: 'ptc1000'>

th10k = <Sensor.th10k: 'th10k'>

TC200.**name**()
Gets the name and version number of the device

Returns the name string of the device

Return type str

TC200.**beta**
Gets/sets the beta value of the thermistor curve.

Value within [2000, 6000]

Returns the gain (in nnn)

Type `int`

TC200.**d**
Gets/sets the d-gain. Valid numbers are [0, 250]

Returns the d-gain (in nnn)

Type `int`

`TC200.`**`degrees`**
Gets/sets the units of the temperature measurement.

Returns The temperature units (degC/F/K) the TC200 is measuring in

Type `UnitTemperature`

`TC200.`**`enable`**
Gets/sets the heater enable status.

If output enable is on (`True`), there is a voltage on the output.

Type `bool`

`TC200.`**`i`**
Gets/sets the i-gain. Valid numbers are [1,250]

Returns the i-gain (in nnn)

Return type `int`

`TC200.`**`max_power`**
Gets/sets the maximum power

Returns The maximum power

Units Watts (linear units)

Type `Quantity`

`TC200.`**`max_temperature`**
Gets/sets the maximum temperature

Returns the maximum temperature (in deg C)

Units As specified or assumed to be degree Celsius. Returns with units degC.

Return type `Quantity`

`TC200.`**`mode`**
Gets/sets the output mode of the TC200

Type *`TC200.Mode`*

`TC200.`**`p`**
Gets/sets the p-gain. Valid numbers are [1,250].

Returns the p-gain (in nnn)

Return type `int`

`TC200.`**`pid`**
Gets/sets all three PID values at the same time. See *`TC200.p`*, *`TC200.i`*, and *`TC200.d`* for individual restrictions.

If `None` is specified then the corresponding PID value is not changed.

Returns List of integers of PID values. In order [P, I, D].

Type `list` or `tuple`

Return type `list`

TC200.**sensor**

Gets/sets the current thermistor type. Used for converting resistances to temperatures.

Returns The thermistor type

Type *TC200.Sensor*

TC200.**status**

Gets the the status code of the TC200

Return type int

TC200.**temperature**

Gets the actual temperature of the sensor

Units As specified (if a Quantity) or assumed to be of units degrees C.

Type Quantity or int

Returns the temperature (in degrees C)

Return type Quantity

TC200.**temperature_set**

Gets/sets the actual temperature of the sensor

Units As specified (if a Quantity) or assumed to be of units degrees C.

Type Quantity or int

Returns the temperature (in degrees C)

Return type Quantity

2.18 Toptica

2.18.1 TopMode Diode Laser

class instruments.toptica.**TopMode**(*filelike*)

Communicates with a Toptica Topmode instrument.

The TopMode is a diode laser with active stabilization, produced by Toptica.

Example usage:

```
>>> import instruments as ik
>>> tm = ik.toptica.TopMode.open_serial('/dev/ttyUSB0', 115200)
>>> print(tm.laser[0].wavelength)
```

class CharmStatus

Enum containing valid charm statuses for the lasers

failure = <CharmStatus.failure: 3>

in_progress = <CharmStatus.in_progress: 1>

success = <CharmStatus.success: 2>

un_initialized = <CharmStatus.un_initialized: 0>

class TopMode.**Laser**(*parent*, *idx*)

Class representing a laser on the Toptica Topmode.

> **Warning:** This class should NOT be manually created by the user. It

is designed to be initialized by the `Topmode` class.

correction()
Run the correction against the specified laser

charm_status
Gets the 'charm status' of the laser
- **Returns** The 'charm status' of the specified laser
- **Type** `bool`

correction_status
Gets the correction status of the laser
- **Returns** The correction status of the specified laser
- **Type** *CharmStatus*

current_control_status
Gets the current control status of the laser
- **Returns** The current control status of the specified laser
- **Type** `bool`

enable
Gets/sets the enable/disable status of the laser. Value of `True` is for enabled, and `False` for disabled.
- **Returns** Enable status of the specified laser
- **Type** `bool`

first_mode_hop_time
Gets the date and time of the first mode hop
- **Returns** The datetime of the first mode hop for the specified laser
- **Type** `datetime`

intensity
Gets the intensity of the laser. This property is unitless.
- **Returns** the intensity of the specified laser
- **Units** Unitless
- **Type** `float`

latest_mode_hop_time
Gets the date and time of the latest mode hop
- **Returns** The datetime of the latest mode hop for the specified laser
- **Type** `datetime`

lock_start
Gets the date and time of the start of mode-locking
- **Returns** The datetime of start of mode-locking for specified laser
- **Type** `datetime`

mode_hop
Gets whether the laser has mode-hopped
- **Returns** Mode-hop status of the specified laser
- **Type** `bool`

model
Gets the model type of the laser
- **Returns** The model of the specified laser
- **Type** `str`

on_time
Gets the 'on time' value for the laser

Returns The 'on time' value for the specified laser
Units Seconds (s)
Type Quantity

production_date
Gets the production date of the laser

Returns The production date of the specified laser
Type str

serial_number
Gets the serial number of the laser

Returns The serial number of the specified laser
Type str

tec_status
Gets the TEC status of the laser

Returns The TEC status of the specified laser
Type bool

temperature_control_status
Gets the temperature control status of the laser

Returns The temperature control status of the specified laser
Type bool

wavelength
Gets the wavelength of the laser

Returns The wavelength of the specified laser
Units Nanometers (nm)
Type Quantity

TopMode.**display**(*param*)
Sends a display command to the Topmode.

Parameters **param** (*str*) – Parameter that will be sent with a display request

Returns Response to the display request

TopMode.**execute**(*command*)
Sends an execute command to the Topmode. This is used to automatically append (exec ' + command +) to your command.

Parameters **command** (*str*) – The command to be executed.

TopMode.**reboot**()
Reboots the system (note that the serial connect might have to be re-opened after this)

TopMode.**reference**(*param*)
Sends a reference commands to the Topmode. This is effectively a query request. It will append the required (param-ref ' + param +).

Parameters **param** (*str*) – Parameter that should be queried

Returns Response to the reference request

Return type str

TopMode.**set**(*param*, *value*)
Sends a param-set command to the Topmode. This is used to automatically handle appending "param-set!" and the rest of the param-set message structure to your message.

Parameters

- **param** (*str*) – Parameter that will be set
- **value** (str, tuple, list, or bool) – Value that the parameter will be set to

TopMode.**current_status**
Gets the current controller board health status

Returns False if there has been a failure for the current controller board, True otherwise

Type bool

TopMode.**enable**
is the laser lasing? :return:

TopMode.**fpga_status**
Gets the FPGA health status

Returns False if there has been a failure for the FPGA, True otherwise

Type bool

TopMode.**interlock**
Gets the interlock switch open state

Returns True if interlock switch is open, False otherwise

Type bool

TopMode.**laser**
Gets a specific Topmode laser object. The desired laser is specified like one would access a list.

For example, the following would print the wavelength from laser 1:

```
>>> import instruments as ik
>>> import quantities as pq
>>> tm = ik.toptica.TopMode.open_serial('/dev/ttyUSB0', 115200)
>>> print(tm.laser[0].wavelength)
```

Return type *Laser*

TopMode.**locked**
Gets the key switch lock status

Returns True if key switch is locked, False otherwise

Type bool

TopMode.**temperature_status**
Gets the temperature controller board health status

Returns False if there has been a failure for the temperature controller board, True otherwise

Type bool

2.19 Yokogawa

2.19.1 Yokogawa7651 Power Supply

class instruments.yokogawa.**Yokogawa7651** (*filelike*)
The Yokogawa 7651 is a single channel DC power supply.

Example usage:

```
>>> import instruments as ik
>>> import quantities as pq
>>> inst = ik.yokogawa.Yokogawa7651.open_gpibusb("/dev/ttyUSB0", 1)
>>> inst.voltage = 10 * pq.V
```

class **`Channel`** (*parent*, *name*)

Class representing the only channel on the Yokogawa 7651.

This class inherits from `PowerSupplyChannel`.

> **Warning:** This class should NOT be manually created by the user. It is designed to be initialized by the *`Yokogawa7651`* class.

`current`

Sets the current of the specified channel. This device has an max setting of 100mA.

Querying the current is not supported by this instrument.

- **Units** As specified (if a `Quantity`) or assumed to be of units Amps.
- **Type** `Quantity` with units Amp

`mode`

Sets the output mode for the power supply channel. This is either constant voltage or constant current.

Querying the mode is not supported by this instrument.

- **Type** *`Yokogawa7651.Mode`*

`output`

Sets the output status of the specified channel. This either enables or disables the output.

Querying the output status is not supported by this instrument.

- **Type** `bool`

`voltage`

Sets the voltage of the specified channel. This device has a voltage range of 0V to +30V.

Querying the voltage is not supported by this instrument.

- **Units** As specified (if a `Quantity`) or assumed to be of units Volts.
- **Type** `Quantity` with units Volt

class `Yokogawa7651.`**`Mode`**

Enum containing valid output modes for the Yokogawa 7651

`current` **= <Mode.current: 5>**

`voltage` **= <Mode.voltage: 1>**

`Yokogawa7651.`**`trigger`**`()`

Triggering function for the Yokogawa 7651.

After changing any parameters of the instrument (for example, output voltage), the device needs to be triggered before it will update.

`Yokogawa7651.`**`channel`**

Gets the specific power supply channel object. Since the Yokogawa7651 is only equiped with a single channel, a list with a single element will be returned.

This (single) channel is accessed as a list in the following manner:

```
>>> import instruments as ik
>>> yoko = ik.yokogawa.Yokogawa7651.open_gpibusb('/dev/ttyUSB0', 10)
>>> yoko.channel[0].voltage = 1 # Sets output voltage to 1V
```

Return type *Channel*

Yokogawa7651.**current**
Sets the current. This device has an max setting of 100mA.

Querying the current is not supported by this instrument.

Units As specified (if a `Quantity`) or assumed to be of units Amps.

Type `Quantity` with units Amp

Yokogawa7651.**voltage**
Sets the voltage. This device has a voltage range of 0V to +30V.

Querying the voltage is not supported by this instrument.

Units As specified (if a `Quantity`) or assumed to be of units Volts.

Type `Quantity` with units Volt

2.20 Configuration File Support

The `instruments` package provides support for loading instruments from a configuration file, so that instrument parameters can be abstracted from the software that connects to those instruments. Configuration files recognized by `instruments` are YAML files that specify for each instrument a class responsible for loading that instrument, along with a URI specifying how that instrument is connected.

Configuration files are loaded by the use of the *load_instruments* function, documented below.

2.20.1 Functions

instruments.**load_instruments**(*conf_file_name*, *conf_path='/'*)
Given the path to a YAML-formatted configuration file and a path within that file, loads the instruments described in that configuration file. The subsection of the configuration file is expected to look like a map from names to YAML nodes giving the class and instrument URI for each instrument. For example:

```
ddg:
    class: !!python/name:instruments.srs.SRSDG645
    uri: gpib+usb://COM7/15
```

Loading instruments from this configuration will result in a dictionary of the form `{'ddg': instruments.srs.SRSDG645.open_from_uri('gpib+usb://COM7/15')}`.

By specifying a path within the configuration file, one can load only a part of the given file. For instance, consider the configuration:

```
instruments:
    ddg:
        class: !!python/name:instruments.srs.SRSDG645
        uri: gpib+usb://COM7/15
prefs:
    ...
```

Then, specifying `"/instruments"` as the configuration path will cause this function to load the instruments named in that block, and ignore all other keys in the YAML file.

Parameters

- **`conf_file_name`** (*`str`*) – Name of the configuration file to load instruments from.
- **`conf_path`** (*`str`*) – `"/"` separated path to the section in the configuration file to load.

Return type `dict`

> **Warning:** The configuration file must be trusted, as the class name references allow for executing arbitrary code. Do not load instruments from configuration files sent over network connections.
>
> Note that keys in sections excluded by the `conf_path` argument are still processed, such that any side effects that may occur due to such processing will occur independently of the value of `conf_path`.

CHAPTER 3

InstrumentKit Development Guide

3.1 Design Philosophy

Here, we describe the design philosophy behind InstrumentKit at a high-level. Specific implications of this philosophy for coding style and practices are detailed in *Coding Style*.

3.1.1 Pythonic

InstrumentKit aims to make instruments and devices look and feel native to the Python development culture. Users should not have to worry if a given instrument names channels starting with 1 or 0, because Python itself is zero-based.

```
>>> scope.data_source = scope.channel[0]
```

Accessing parts of an instrument should be supported in a way that supports standard Python idioms, most notably iteration.

```
>>> for channel in scope.channel:
...     channel.coupling = scope.Coupling.ground
```

Values that can be queried and set should be exposed as properties. Instrument modes that should be entered and exited on a temporary basis should be exposed as context managers. In short, anyone familiar with Python should be able to read InstrumentKit-based programs with little to no confusion.

3.1.2 Abstract

Users should not have to worry overmuch about the particular instruments that are being used, but about the functionality that instrument exposes. To a large degree, this is enabled by using common base classes, such as `instruments.generic_scpi.SCPIOscilloscope`. While every instrument does offer its own unique functionality, by consolidating common functionality in base classes, users can employ some subset without worrying too much about the particulars.

This also extends to communications methods. By consolidating communication logic in the `instruments.abstract_instruments.comm.AbstractCommunicator` class, users can connect instruments however is convienent for them, and can change communications methods without affecting their software very much.

3.1.3 Robust

Communications with instruments should be handled in such a way that errors are reported in a natural and Python-ic way, such that incorrect or unsafe operations are avoided, and such that all communications are correct.

An important consequence of this is that all quantities communicated to or from the instrument should be *unitful*. In this way, users can specify the dimensionality of values to be sent to the device without regards for what the instrument expects; the unit conversions will be handled by InstrumentKit in a way that ensures that the expectations of the instrument are properly met, irrespective of what the user knows.

3.2 Coding Style

3.2.1 Data Types

Numeric Data

When appropriate, use `quantities.Quantity` objects to track units. If this is not possible or appropriate, use a bare `float` for scalars and `np.ndarray` for array-valued data.

Boolean and Enumerated Data

If a property or method argument can take exactly two values, of which one can be interpreted in the affirmative, use Python `bool` data types to represent this. Be permissive in what you accept as `True` and `False`, in order to be consistent with Python conventions for truthy and falsey values. This can be accomplished using the `bool` function to convert to Booleans, and is done implicitly by the `if` statement.

If a property has more than two permissible values, or the two allowable values are not naturally interpreted as a Boolean (e.g.: positive/negative, AC/DC coupling, etc.), then consider using an `Enum` or `IntEnum` as provided by `enum`. The latter is useful in for wrapping integer values that are meaningful to the device.

For example, if an instrument can operate in AC or DC mode, use an enumeration like the following:

```
class SomeInstrument(Instrument):

        # Define as an inner class.
        class Mode(Enum):
                """
                When appropriate, document the enumeration itself...
                """
                #: ...and each of the enumeration values.
                ac = "AC"
                #: The "#:" notation means that this line documents
                #: the following member, SomeInstrument.Mode.dc.
                dc = "DC"

        # For SCPI-like instruments, enum_property
        # works well to expose the enumeration.
        # This will generate commands like "MODE AC"
        # and "MODE DC".
        mode = enum_property(
            name=":MODE",
            enum=SomeInstrument.Mode,
            doc="""
            And here is the docstring for this property
            """
```

```
)

# To set the mode is now straightforward.
ins = SomeInstrument.open_somehow()
ins.mode = ins.Mode.ac
```

Note that the enumeration is an inner class, as described below in *Associated Types*.

3.2.2 Object Oriented Design

Associated Types

Many instrument classes have associated types, such as channels and axes, so that these properties of the instrument can be manipulated independently of the underlying instrument:

```
>>> channels = [ins1.channel[0], ins2.channel[3]]
```

Here, the user of `channels` need not know or care that the two channels are from different instruments, as is useful for large installations. This lets users quickly redefine their setups with minimal code changes.

To enable this, the associated types should be made inner classes that are exposed using `ProxyList`. For example:

```
class SomeInstrument(Instrument):
        # If there's a more appropriate base class, please use it
        # in preference to object!
        class Channel(object):
                # We use a three-argument initializer,
                # to remember which instrument this channel belongs to,
                # as well as its index or label on that instrument.
                # This will be useful in sending commands, and in exposing
                # via ProxyList.
                def __init__(self, parent, idx):
                        self._parent = parent
                        self._idx = idx
                # define some things here...

        @property
        def channel(self):
                return ProxyList(self, SomeInstrument.Channel, range(2))
```

This defines an instrument with two channels, having labels `0` and `1`. By using an inner class, the channel is clearly associated with the instrument, and appears with the instrument in documentation.

Since this convention is somewhat recent, you may find older code that uses a style more like this:

```
class _SomeInstrumentChannel(object):
        # stuff

class SomeInstrument(Instrument):
        @property
        def channel(self):
                return ProxyList(self, _SomeInstrumentChannel, range(2))
```

This can be redefined in a backwards-compatible way by bringing the channel class inside, then defining a new module-level variable for the old name:

```
class SomeInstrument(Instrument):
        class Channel(object):
```

```
                # stuff

        @property
        def channel(self):
                return ProxyList(self, _SomeInstrumentChannel, range(2))

_SomeInstrumentChannel = SomeInstrument.Channel
```

3.3 Testing Instrument Functionality

3.3.1 Overview

When developing new instrument classes, or adding functionality to existing instruments, it is important to also add automated checks for the correctness of the new functionality. Such tests serve two distinct purposes:

- Ensures that the protocol for each instrument is being followed correctly, even with changes in the underlying InstrumentKit behavior.
- Ensures that the API seen by external users is kept stable and consistent.

The former is especially important for instrument control, as the developers of InstrumentKit will not, in general, have access to each instrument that is supported— we rely on automated testing to ensure that future changes do not cause invalid or undesired operation.

For InstrumentKit, we rely heavily on nose, a mature and flexible unit-testing framework for Python. When run from the command line via `nosetests`, or when run by Travis CI, nose will automatically execute functions and methods whose names start with `test` in packages, modules and classes whose names start with `test` or `Test`, depending. (Please see the nose documentation for full details, as this is not intended to be a guide to nose so much as a guide to how we use it in IK.) Because of this, we keep all test cases in the `instruments.tests` package, under a subpackage named for the particular manufacturer, such as `instruments.tests.test_srs`. The tests for each instrument should be contained within its own file. Please see current tests as an example. If the number of tests for a given instrument is numerous, please consider making modules within a manufacturer test subpackage for each particular device.

Below, we discuss two distinct kinds of unit tests: those that check that InstrumentKit functionality such as *Property Factories* work correctly for new instruments, and those that check that existing instruments produce correct protocols.

3.3.2 Mock Instruments

TODO

3.3.3 Expected Protocols

As an example of asserting correctness of implemented protocols, let's consider a simple test case for `instruments.srs.SRSDG645``:

```
def test_srsdg645_output_level():
    """
    SRSDG645: Checks getting/setting unitful ouput level.
    """
    with expected_protocol(ik.srs.SRSDG645,
            [
                "LAMP? 1",
```

```
                "LAMP 1,4.0",
            ], [
                "3.2"
            ],
            sep="\n"
    ) as ddg:
        unit_eq(ddg.output['AB'].level_amplitude, pq.Quantity(3.2, "V"))
        ddg.output['AB'].level_amplitude = 4.0
```

Here, we see that the test has a name beginning with `test_`, has a simple docstring that will be printed in reports on failing tests, and then has a call to `expected_protocol()`. The latter consists of specifying an instrument class, here given as `ik.srs.DG645`, then a list of expected outputs and playback to check property accessors.

Note that `expected_protocol()` acts as a context manager, such that it will, at the end of the indented block, assert the correct operation of the contents of that block. In this example, the second argument to `expected_protocol()` specifies that the instrument class should have sent out two strings, `"LAMP? 1"` and `LAMP 1,4.0`, during the block, and should act correctly when given an answer of `"3.2"` back from the instrument. The third parameter, `sep` specifies what will be appended to the end of each lines in the previous parameters. This lets you specify the termination character that will be used in the communication without having to write it out each and every time.

Protocol Assertion Functions

3.4 Utility Functions and Classes

3.4.1 Unit Handling

`instruments.util_fns.`**`assume_units`**(*value*, *units*)

If units are not provided for `value` (that is, if it is a raw `float`), then returns a `Quantity` with magnitude given by `value` and units given by `units`.

Parameters

- **`value`** – A value that may or may not be unitful.
- **`units`** – Units to be assumed for `value` if it does not already have units.

Returns A unitful quantity that has either the units of `value` or `units`, depending on if `value` is unitful.

Return type `Quantity`

`instruments.util_fns.`**`split_unit_str`**(*s*, *default_units=Dimensionless('dimensionless', 1.0 * dimensionless)*, *lookup=None*)

Given a string of the form "12 C" or "14.7 GHz", returns a tuple of the numeric part and the unit part, irrespective of how many (if any) whitespace characters appear between.

By design, the tuple should be such that it can be unpacked into `pq.Quantity()`:

```
>>> pq.Quantity(*split_unit_str("1 s"))
array(1) * s
```

For this reason, the second element of the tuple may be a unit or a string, depending, since the quantity constructor takes either.

Parameters

- **s** (*`str`*) – Input string that will be split up

- **default_units** – If no units are specified, this argument is given as the units.
- **lookup** (*callable*) – If specified, this function is called on the units part of the input string. If None, no lookup is performed. Lookups are never performed on the default units.

Return type tuple of a float and a str or pq.Quantity

instruments.util_fns.**convert_temperature**(*temperature*, *base*)

Convert the temperature to the specified base. This is needed because the package quantities does not differentiate between degC and degK.

Parameters

- **temperature** (quantities.Quantity) – A quantity with units of Kelvin, Celsius, or Fahrenheit
- **base** (unitquantity.UnitTemperature) – A temperature unit to convert to

Returns The converted temperature

Return type quantities.Quantity

3.4.2 Enumerating Instrument Functionality

To expose parts of an instrument or device in a Python-ic way, the ProxyList class can be used to emulate a list type by calling the initializer for some inner class. This is used to expose everything from channels to axes.

3.4.3 Property Factories

To help expose instrument properties in a consistent and predictable manner, InstrumentKit offers several functions that return instances of property that are backed by the *sendcmd()* and *query()* protocol. These factories assume a command protocol that at least resembles the SCPI style:

```
-> FOO:BAR?
<- 42
-> FOO:BAR 6
-> FOO:BAR?
<- 6
```

It is recommended to use the property factories whenever possible to help reduce the amount of copy-paste throughout the code base. The factories allow for a centralized location for input/output error checking, units handling, and type conversions. In addition, improvements to the property factories benefit all classes that use it.

Lets say, for example, that you were writing a class for a power supply. This class might require these two properties: output and voltage. The first will be used to enable/disable the output on the power supply, while the second will be the desired output voltage when the output is enabled. The first lends itself well to a *bool_property*. The output voltage property corresponds with a physical quantity (voltage, of course) and so it is best to use either *unitful_property* or *bounded_unitful_property*, depending if you wish to bound user input to some set limits. *bounded_unitful_property* can take either hard-coded set limits, or it can query the instrument during runtime to determine what those bounds are, and constrain user input to within them.

Examples

These properties, when implemented in your class, might look like this:

```
output = bool_property(
    "OUT",
    inst_true="1",
    inst_false="0",
    doc="""
    Gets/sets the output status of the power supply

    :type: `bool`
    """
)

voltage, voltage_min, voltage_max = bounded_unitful_property(
    voltage = unitful_property(
    "VOLT",
    pq.volt,
    valid_range=(0*quantities.volt, 10*quantities.volt)
    doc="""
    Gets/sets the output voltage.

    :units: As specified, or assumed to be :math:`\\text{V}` otherwise.
    :type: `float` or `~quantities.Quantity`
    """
)
```

The most difficult to use parameters for the property factories are `input_decoration` and `output_decoration`. These are callable objects that will be applied to the data immediately after receiving it from the instrument (input) or before it is inserted into the string that will be sent out to the instrument (output).

Using *enum_property* as the simple example, a frequent use case for `input_decoration` will be to convert a `str` containing a numeric digit into an actual `int` so that it can be looked up in `enum.IntEnum`. Here is an example of this:

```
class Mode(IntEnum):

    """
    Enum containing valid output modes of the ABC123 instrument
    """
    foo = 0
    bar = 1
    bloop = 2

mode = enum_property(
    "MODE",
    enum=Mode,
    input_decoration=int,
    set_fmt="{}={}",
    doc="""
    Gets/sets the output mode of the ABC123 instrument

    :rtype: `ABC123.Mode`
    """
)
```

So in this example, when querying the `mode` property, the string `MODE?` will first be sent to the instrument, at which point it will return one of `"0"`, `"1"`, or `"2"`. However, before this value can be used to get the correct enum value, it needs to be converted into an `int`. This is what `input_decoration` is used for. Since `int` is callable and can convert a `str` to an `int`, this accomplishes exactly what we're looking for.

Pretty much anything callable can be passed into these parameters. Here is an example using a lambda function with a *unitful_property* taken from the *TC200* class:

```
temperature = unitful_property(
    "tact",
    units=pq.degC,
    readonly=True,
    input_decoration=lambda x: x.replace(
        " C", "").replace(" F", "").replace(" K", ""),
    doc="""
    Gets the actual temperature of the sensor

    :units: As specified (if a `~quantities.quantity.Quantity`) or assumed
        to be of units degrees C.
    :type: `~quantities.quantity.Quantity` or `int`
    :return: the temperature (in degrees C)
    :rtype: `~quantities.quantity.Quantity`
    """
)
```

An alternative to lambda functions is passing in static methods (`staticmethod`).

Bool Property

`instruments.util_fns.`**`bool_property`**(*name*, *inst_true*, *inst_false*, *doc=None*, *readonly=False*, *writeonly=False*, *set_fmt='{} {}'*)

Called inside of SCPI classes to instantiate boolean properties of the device cleanly. For example:

```
>>> my_property = bool_property("BEST:PROPERTY", "ON", "OFF")
```

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **inst_true** (*str*) – String returned and accepted by the instrument for `True` values.
- **inst_false** (*str*) – String returned and accepted by the instrument for `False` values.
- **doc** (*str*) – Docstring to be associated with the new property.
- **readonly** (*bool*) – If `True`, the returned property does not have a setter.
- **writeonly** (*bool*) – If `True`, the returned property does not have a getter. Both readonly and writeonly cannot both be `True`.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}" which places a space between the SCPI command the associated parameter. By switching to "{}={}" an equals sign would instead be used as the separator.

Enum Property

`instruments.util_fns.`**`enum_property`**(*name*, *enum*, *doc=None*, *input_decoration=None*, *output_decoration=None*, *readonly=False*, *writeonly=False*, *set_fmt='{} {}'*)

Called inside of SCPI classes to instantiate Enum properties of the device cleanly. The decorations can be functions which modify the incoming and outgoing values for dumb instruments that do stuff like include superfluous quotes that you might not want in your enum. Example: my_property = bool_property("BEST:PROPERTY", enum_class)

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **enum** (type) – Class derived from Enum representing valid values.
- **input_decoration** (*callable*) – Function called on responses from the instrument before passing to user code.
- **output_decoration** (*callable*) – Function called on commands to the instrument.
- **doc** (*str*) – Docstring to be associated with the new property.
- **readonly** (*bool*) – If True, the returned property does not have a setter.
- **writeonly** (*bool*) – If True, the returned property does not have a getter. Both readonly and writeonly cannot both be True.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}" which places a space between the SCPI command the associated parameter. By switching to "{}={}" an equals sign would instead be used as the separator.

Unitless Property

instruments.util_fns.**unitless_property**(*name*, *format_code='{:e}'*, *doc=None*, *readonly=False*, *writeonly=False*, *set_fmt='{} {}'*)

Called inside of SCPI classes to instantiate properties with unitless numeric values.

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **format_code** (*str*) – Argument to str.format used in sending values to the instrument.
- **doc** (*str*) – Docstring to be associated with the new property.
- **readonly** (*bool*) – If True, the returned property does not have a setter.
- **writeonly** (*bool*) – If True, the returned property does not have a getter. Both readonly and writeonly cannot both be True.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}" which places a space between the SCPI command the associated parameter. By switching to "{}={}" an equals sign would instead be used as the separator.

Int Property

instruments.util_fns.**int_property**(*name*, *format_code='{:d}'*, *doc=None*, *readonly=False*, *writeonly=False*, *valid_set=None*, *set_fmt='{} {}'*)

Called inside of SCPI classes to instantiate properties with unitless numeric values.

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **format_code** (*str*) – Argument to str.format used in sending values to the instrument.
- **doc** (*str*) – Docstring to be associated with the new property.

- **readonly** (*bool*) – If True, the returned property does not have a setter.
- **writeonly** (*bool*) – If True, the returned property does not have a getter. Both readonly and writeonly cannot both be True.
- **valid_set** – Set of valid values for the property, or None if all int values are valid.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}" which places a space between the SCPI command the associated parameter. By switching to "{}={}" an equals sign would instead be used as the separator.

Unitful Property

instruments.util_fns.**unitful_property**(*name, units, format_code='{:e}', doc=None, input_decoration=None, output_decoration=None, readonly=False, writeonly=False, set_fmt='{} {}', valid_range=(None, None)*)

Called inside of SCPI classes to instantiate properties with unitful numeric values. This function assumes that the instrument only accepts and returns magnitudes without unit annotations, such that all unit information is provided by the units argument. This is not suitable for instruments where the units can change dynamically due to front-panel interaction or due to remote commands.

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **units** – Units to assume in sending and receiving magnitudes to and from the instrument.
- **format_code** (*str*) – Argument to str.format used in sending the magnitude of values to the instrument.
- **doc** (*str*) – Docstring to be associated with the new property.
- **input_decoration** (*callable*) – Function called on responses from the instrument before passing to user code.
- **output_decoration** (*callable*) – Function called on commands to the instrument.
- **readonly** (*bool*) – If True, the returned property does not have a setter.
- **writeonly** (*bool*) – If True, the returned property does not have a getter. Both readonly and writeonly cannot both be True.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}" which places a space between the SCPI command the associated parameter. By switching to "{}={}" an equals sign would instead be used as the separator.
- **valid_range** (tuple or list of int or float) – Tuple containing min & max values when setting the property. Index 0 is minimum value, index 1 is maximum value. Setting None in either disables bounds checking for that end of the range. The default of (None, None) has no min or max constraints. The valid set is inclusive of the values provided.

Bounded Unitful Property

`instruments.util_fns.`**`bounded_unitful_property`**(*name*, *units*, *min_fmt_str='{}:MIN?'*, *max_fmt_str='{}:MAX?'*, *valid_range=('query'*, *'query')*, ***kwargs*)

Called inside of SCPI classes to instantiate properties with unitful numeric values which have upper and lower bounds. This function in turn calls `unitful_property` where all kwargs for this function are passed on to. See `unitful_property` documentation for information about additional parameters that will be passed on.

Compared to `unitful_property`, this function will return 3 properties: the one created by `unitful_property`, one for the minimum value, and one for the maximum value.

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **units** – Units to assume in sending and receiving magnitudes to and from the instrument.
- **min_fmt_str** (*str*) – Specify the string format to use when sending a minimum value query. The default is `"{}:MIN?"` which will place the property name in before the colon. Eg: `"MOCK:MIN?"`
- **max_fmt_str** (*str*) – Specify the string format to use when sending a maximum value query. The default is `"{}:MAX?"` which will place the property name in before the colon. Eg: `"MOCK:MAX?"`
- **valid_range** (`list` or `tuple` of `int`, `float`, `None`, or the string `"query"`.) – Tuple containing min & max values when setting the property. Index 0 is minimum value, index 1 is maximum value. Setting `None` in either disables bounds checking for that end of the range. The default of `("query", "query")` will query the instrument for min and max parameter values. The valid set is inclusive of the values provided.
- **kwargs** – All other keyword arguments are passed onto `unitful_property`

Returns Returns a `tuple` of 3 properties: first is as returned by `unitful_property`, second is a property representing the minimum value, and third is a property representing the maximum value

String Property

`instruments.util_fns.`**`string_property`**(*name*, *bookmark_symbol='"'*, *doc=None*, *readonly=False*, *writeonly=False*, *set_fmt='{} {}{}{}'*)

Called inside of SCPI classes to instantiate properties with a string value.

Parameters

- **name** (*str*) – Name of the SCPI command corresponding to this property.
- **doc** (*str*) – Docstring to be associated with the new property.
- **readonly** (*bool*) – If `True`, the returned property does not have a setter.
- **writeonly** (*bool*) – If `True`, the returned property does not have a getter. Both readonly and writeonly cannot both be `True`.
- **set_fmt** (*str*) – Specify the string format to use when sending a non-query to the instrument. The default is "{} {}{}{}" which places a space between the SCPI command the associated parameter, and places the bookmark symbols on either side of the parameter.
- **bookmark_symbol** (*str*) – The symbol that will flank both sides of the parameter to be sent to the instrument. By default this is `"`.

3.5 Introduction

This guide details how InstrumentKit is laid out from a developer's point of view, how to add instruments, communication methods and unit tests.

3.6 Getting Started

To get started with development for InstrumentKit, a few additional supporting packages must be installed. The core development packages can be found in the supporting requirements file named `dev-requirements.txt`. These will allow you to run the tests and check that all your code changes follow our linting rules (through `pylint`).

3.6.1 Required Development Dependencies

Using `pip`, these requirements can be obtained automatically by using the provided `dev-requirements.txt`:

```
$ pip install -r dev-requirements.txt
```

- mock
- nose
- pylint

3.6.2 Optional Development Dependencies

In addition to the required dev dependencies, there are optional ones. The package tox allows you to quickly run the tests against all supported versions of Python, assuming you have them installed. It is suggested that you install `tox` and regularly run your tests by calling the simple command:

```
$ tox
```

More details on running tests can be found in testing.

3.7 Contributing Code

We love getting new instruments and new functionality! When sending in pull requests, however, it helps us out a lot in maintaining InstrumentKit as a usable library if you can do a couple things for us with your submission:

- Make sure code follows PEP 8 as best as possible. This helps keep the code readable and maintainable.
- Document properties and methods, including units where appropriate.
- Contributed classes should feature complete code coverage to prevent future changes from breaking functionality. This is especially important if the lead developers do not have access to the physical hardware.
- Please use *Property Factories* when appropriate, to consolidate parsing logic into a small number of easily-tested functions. This will also reduce the number of tests required to be written.

We can help with any and all of these, so please ask, and thank you for helping make InstrumentKit even better.

CHAPTER 4

Acknowledgements

Here I've done my best to keep a list of all those who have made a contribution to this project. All names listed below are the Github account names associated with their commits.

First off, I'd like to give special thanks to cgranade for his help with pretty much every step along the way. I would be hard pressed to find something that he had nothing to do with.

- ihincks for the fantastic property factories (used throughout all classes) and for the Tektronix DPO70000 series class.
- dijkstrw for contributing several classes (HP6632b, HP3456a, Keithley 580) as well as plenty of general IK testing.
- CatherineH for the Qubitekk CC1, Thorlabs LCC25, SC10, and TC200 classes
- silverchris for the TekTDS5xx class
- wil-langford for the HP6652a class
- whitewhim2718 for the Newport ESP 301

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z