
InstrumentKit Library Documentation

Release 0.6.0

Steven Casagrande

Dec 19, 2020

Contents

1	Introduction	3
2	InstrumentKit API Reference	7
3	InstrumentKit Development Guide	159
4	Acknowledgements	173
5	Indices and tables	175
	Index	177

Contents:

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`.

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 convenience 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 convenience, 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+="tekdp4104", 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.

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 `usbttmc` 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
usbtc://USB::0x0699::0x0401::C0000001::0::INSTR
test://
```

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*, *model='pl'*)

Opens an instrument, connecting via a Prologix GPIBETHERNET adapter.

Parameters

- **host** (*str*) – Name or IP address of the instrument.
- **port** (*int*) – TCP port on which the instrument is listening.
- **gpib_address** (*int*) – Address on the connected GPIB bus assigned to the instrument.
- **model** (*str*) – The brand of adapter to be connected to. Currently supported is “gi” for Galvant Industries, and “pl” for Prologix LLC.

Warning: This function has been setup for use with the Prologix GPIBETHERNET adapter but has not been tested as confirmed working.

classmethod `open_gpibusb` (*port*, *gpib_address*, *timeout=3*, *write_timeout=3*, *model='gi'*)

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.

- **model** (*str*) – The brand of adapter to be connected to. Currently supported is “gi” for Galvant Industries, and “pl” for Prologix LLC.

Return type *Instrument*

Returns Object representing the connected instrument.

See also:

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

classmethod `open_serial` (*port=None, baud=9600, vid=None, pid=None, serial_number=None, 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.

This method can be called by either supplying a port as a string, or by specifying vendor and product IDs, and an optional serial number (used when more than one device with the same IDs is attached). If both the port and IDs are supplied, the port will default to the supplied port string, else it will search the available com ports for a port matching the defined IDs and serial number.

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.
- **vid** (*int*) – the USB port vendor id.
- **pid** (*int*) – the USB port product id.
- **serial_number** (*str*) – The USB port serial_number.
- **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 instrument 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`, `encoding='utf-8'`)

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 = ['serial', 'tcpip', 'gpib+usb', 'gpib+serial', 'visa', 'file', 'usbtmc',

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 Channel (*parent, name*)

Abstract base class for physical channels on a function generator.

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

Function generators that only have a single channel do not need to define their own concrete implementation of this class. Ones with multiple channels need their own definition of this class, where this class contains the concrete implementations of the below abstract methods. Instruments with 1 channel have their concrete implementations at the parent instrument level.

query (*cmd, size=-1*)

sendcmd (*cmd*)

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.

frequency

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

Type `Quantity`

function

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

Type `Enum`

offset

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

Type `Quantity`

phase

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

Type `Quantity`

class Function

Enum containg valid output function modes for many function generators

arbitrary = 'ARB'

noise = 'NOIS'

ramp = 'RAMP'

sinusoid = 'SIN'

square = 'SQU'

triangle = 'TRI'

class VoltageMode

Enum containing valid voltage modes for many function generators

dBm = 'DBM'

peak_to_peak = 'VPP'

rms = 'VRMS'

amplitude

Gets/sets the output amplitude of the first channel of the function generator

Type `Quantity`

channel

Gets a channel object for the function generator. This should use `ProxyList` to achieve this.

The number of channels accessible depends on the value of `FunctionGenerator._channel_count`

Return type `FunctionGenerator.Channel`

frequency

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

Type `Quantity`

function

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

Type `Enum`

offset

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

Type `Quantity`

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

Gets a specific channel object for the `SignalGenerator`.

Return type A class inherited from `SGChannel`

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 `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 = -160`

`block_data_not_allowed = -168`

`character_data_error = -140`

`character_data_not_allowed = -148`

```
character_data_too_long = -144
command_error = -100
command_header_error = -110
data_type_error = -104
exponent_too_large = -123
expression_error = -170
expression_not_allowed = -178
get_not_allowed = -105
header_separator_error = -111
header_suffix_out_of_range = -114
invalid_block_data = -161
invalid_character = -101
invalid_character_data = -141
invalid_character_in_number = -121
invalid_expression = -171
invalid_inside_macro_definition = -183
invalid_outside_macro_definition = -181
invalid_separator = -103
invalid_string_data = -151
invalid_suffix = -131
macro_error = -180
macro_parameter_error = -184
missing_parameter = -109
no_error = 0
numeric_data_error = -120
numeric_data_not_allowed = -128
operation_complete = -800
parameter_not_allowed = -108
power_on = -500
    Raised when the instrument detects that it has been turned from off to on.
program_mnemonic_too_long = -112
request_control_event = -700
string_data_error = -150
string_data_not_allowed = -158
suffix_error = -130
suffix_not_allowed = -138
```

```

suffix_too_long = -134
syntax_error = -102
too_many_digits = -124
undefined_header = -113
unexpected_number_of_parameters = -115
user_request_event = -600

```

check_error_queue()

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

clear()

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

reset()

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

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.

wait_to_continue()

Instruct the instrument to wait until it has completed all received commands before continuing.

display_brightness

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

Type *float*

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*

line_frequency

Gets/sets the power line frequency setting for the instrument.

Returns The power line frequency

Units Hertz

Type *Quantity*

name

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

Return type *str*

op_complete

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

Return type *bool*

power_on_status

Gets/sets the power on status for the instrument.

Type *bool*

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.

self_test_ok

Gets the results of the instrument's self test. This lets you check if the self test was successful 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 = 'AUTO'`

`default = 'DEF'`

`maximum = 'MAX'`

`minimum = 'MIN'`

class `Mode`

Enum of valid measurement modes for (most) SCPI compliant multimeters

`capacitance = 'CAP'`

`continuity = 'CONT'`

`current_ac = 'CURR:AC'`

`current_dc = 'CURR:DC'`

`diode = 'DIOD'`

`fourpt_resistance = 'FRES'`

`frequency = 'FREQ'`

`period = 'PER'`

`resistance = 'RES'`

`temperature = 'TEMP'`

`voltage_ac = 'VOLT:AC'`

`voltage_dc = 'VOLT:DC'`

class `Resolution`

Valid measurement resolution parameters outside of directly the resolution.

`default = 'DEF'`

`maximum = 'MAX'`

```

    minimum = 'MIN'
class SampleCount
    Valid sample count parameters outside of directly the value.
    default = 'DEF'
    maximum = 'MAX'
    minimum = 'MIN'
class 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 = 'IMM'
    timer = 'TIM'
class TriggerCount
    Valid trigger count parameters outside of directly the value.
    default = 'DEF'
    infinity = 'INF'
    maximum = 'MAX'
    minimum = 'MIN'
class 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 = 'BUS'
    external = 'EXT'
    immediate = 'IMM'
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.
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 * u.millivolt
```

Units As appropriate for the current mode setting.

Type *Quantity*, or *InputRange*

mode

Gets/sets the current measurement mode for the multimeter.

Example usage:

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

Type *Mode*

relative

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

Type *bool*

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*

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` property has been changed, the number of readings taken total will be a multiplication of sample count and trigger count (see property `SCPIMultimeter.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*

sample_source

Gets/sets the multimeter sample source. This determines whether the trigger delay or the sample timer is used to determine 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*

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`

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` property has been changed, the number of readings taken total will be a multiplication of sample count and trigger count (see property `SCPIMultimeter.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`

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`

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_scp.SCPIFunctionGenerator` (*filelike*)

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

Example usage:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.generic_scp.SCPIFunctionGenerator.open_tcpip("192.168.1.1")
>>> inst.frequency = 1 * u.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

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

Type `Quantity`

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

class `Function`

Enum containing valid functions for the Agilent/Keysight 33220a

`dc = 'DC'`

`noise = 'NOIS'`

`pulse = 'PULS'`

`ramp = 'RAMP'`

`sinusoid = 'SIN'`

`square = 'SQU'`

`user = 'USER'`

class `LoadResistance`

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

```

    high_impedance = 'INF'
    maximum = 'MAX'
    minimum = 'MIN'

```

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

```

    inverted = 'INV'
    normal = 'NORM'

```

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`

function
Gets/sets the output function of the function generator
Type `Agilent33220a.Function`

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`

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`

output_polarity
Gets/sets the polarity of the waveform relative to the offset voltage.
Type `OutputPolarity`

output_sync
Gets/sets the enabled status of the front panel sync connector.
Type `bool`

phase
Gets/sets the output phase of the function generator. This is an abstract property.
Type `Quantity`

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 compatibility 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 instruments.units as u
>>> 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 transferred as ASCII, and thus it is not recommended to transfer a large number of data points using this method.

Return type `tuple` [~ pint.Quantity, ...]` or if numpy is installed, `Quantity` with `numpy.array` data

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 transferred from the instrument in 64-bit double floating point precision format.

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

Return type `tuple` [~ pint.Quantity, ...]` or if numpy is installed, `Quantity` with `numpy.array` data

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 transferred to output buffer. If set to -1, all points in memory will be transferred.

Return type `tuple` [~ pint.Quantity, ...]` or if numpy is installed, `Quantity` with `numpy.array` data

read_data_nvmem ()

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

Return type `tuple` [~ pint.Quantity, ...]` or if numpy is installed, `Quantity` with `numpy.array` data

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 Fluke

2.4.1 Fluke3000 Industrial System

class `instruments.fluke.Fluke3000` (*filelike*)

The *Fluke3000* is an ecosystem of devices produced by Fluke that may be connected simultaneously to a Fluke PC3000 wireless adapter which exposes a serial port to the computer to send and receive commands.

The *Fluke3000* ecosystem supports the following instruments:

- Fluke 3000 FC Series Wireless Multimeter
- Fluke v3000 FC Wireless AC Voltage Module
- Fluke v3001 FC Wireless DC Voltage Module
- Fluke t3000 FC Wireless Temperature Module

Fluke3000 is a USB instrument that communicates through a serial port via the PC3000 dongle. The commands used to communicate to the dongle do not follow the SCPI standard.

When the device is reset, it searches for available wireless modules and binds them to the PC3000 dongle. At each initialization, this class checks what device has been bound and saves their module number.

This class has been tested with the 3000 FC Wireless Multimeter and the t3000 FC Wireless Temperature Module. They have been operated separately and simultaneously. It does not support the Wireless AC/DC Voltage Modules as the author did not have them on hand.

It is important to note that the mode of the multimeter cannot be set remotely. It must be set on the device prior to the measurement. If the measurement read back from the multimeter is not expressed in the expected units, this module will raise an error.

Example usage:

```
>>> import instruments as ik
>>> mult = ik.fluke.Fluke3000.open_serial("/dev/ttyUSB0", 115200)
>>> mult.measure(mult.Mode.voltage_dc) # Measures the DC voltage
array(12.345) * V
```

It is crucial not to kill this program in the process of making a measurement, as for the Fluke 3000 FC Wireless Multimeter, one has to open continuous readout, make a read and close it. If the process is killed, the read out may not be closed and the serial cache will be constantly filled with measurements that will interfere with any status query. If the multimeter is stuck in continuous trigger after a bad kill, simply do:

```
>>> mult.reset()
>>> mult.flush()
>>> mult.connect()
```

Follow the same procedure if you want to add/remove an instrument to/from the readout chain as the code will not look for new instruments if some have already been connected to the PC3000 dongle.

class Mode

Enum containing the supported mode codes.

capacitance = '0F'
Capacitance

current_ac = '03'
AC Current

current_dc = '04'
DC Current

frequency = '05'
Frequency

resistance = '0B'
Resistance

temperature = '07'
Temperature

voltage_ac = '01'
AC Voltage

voltage_dc = '02'
DC Voltage

class Module

Enum containing the supported modules serial numbers.

m3000 = 46333030304643
Multimeter

t3000 = 54333030304643
Temperature module

connect ()

Connect to available modules and returns a dictionary of the modules found and their port ID.

flush ()

Flushes the serial input cache.

This device outputs a terminator after each output line. The serial input cache is flushed by repeatedly reading until a terminator is not found.

measure (mode)

Instruct the Fluke3000 to perform a one time measurement.

Parameters *mode* (*Fluke3000.Mode*) – Desired measurement mode.

Returns A measurement from the multimeter.

Return type `Quantity`

query_lines (*cmd*, *nlines=1*)

Function used to send a query to the instrument while allowing for the multiline output of the PC3000.

Parameters

- **cmd** (*'str'*) – Command that will be sent to the instrument
- **nlines** (*'int'*) – Number of termination characters to reach

Returns The multiline result from the query

Return type Array of `str`

read_lines (*nlines=1*)

Function that keeps reading until reaches a termination character a set amount of times. This is implemented to handle the mutiline output of the PC3000.

Parameters **nlines** (*'int'*) – Number of termination characters to reach

Returns Array of lines read out

Return type Array of `str`

reset ()

Resets the device and unbinds all modules.

scan ()

Search for available modules and reformat. Returns a dictionary of the modules found and their port ID.

input_range

Gets/sets the current input range setting of the multimeter.

The *Fluke3000* FC is an autoranging only multimeter.

Return type `str`

mode

Gets/sets the measurement mode for the multimeter.

The measurement mode of the multimeter must be set on the device manually and cannot be set remotely. If a multimeter is bound to the PC3000, returns its measurement mode by making a measurement and checking the units bytes in response.

Return type `Fluke3000.Mode`

relative

Gets/sets the status of relative measuring mode for the multimeter.

The *Fluke3000* FC does not support relative measurements.

Return type `bool`

trigger_mode

Gets/sets the trigger mode for the multimeter.

The only supported mode is to trigger the device once when a measurement is queried. This device does support continuous triggering but it would quickly flood the serial input cache as readouts do not overwrite each other and are accumulated.

Return type `str`

2.5 Gentec-EO

2.5.1 Blu Power Meter

class `instruments.gentec_eo.Blu` (*filelike*)

Communicate with Gentec-eo BLU power / energy meter interfaces.

These instruments communicate via USB or via bluetooth. The bluetooth sender / receiver that is provided with the instrument is simply emulating a COM port. This routine cannot pair the device with bluetooth, but once it is paired, it can communicate with the port. Alternatively, you can plug the device into the computer using a USB cable.

Warning: If commands are issued too fast, the device will not answer. Experimentally, a 1 ms delay should be enough to get the device into answering mode. Keep this in mind when issuing many commands at once. No wait time included in this class.

Note: The instrument also has a possibility to read a continuous data stream. This is currently not implemented here since it would have to be threaded out.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.current_value
3.004 W
```

class Scale

Available scales for Blu devices.

The following list maps available scales of the Blu devices to the respective indexes. All scales are either in watts or joules, depending if power or energy mode is activated. Furthermore, the maximum value that can be measured determines the name of the scale to be set. Prefixes are given in the `enum` class while the unit is omitted since it depends on the mode the head is in.

```
max1 = '24'
max10 = '26'
max100 = '28'
max100Mega = '40'
max100kilo = '34'
max100micro = '16'
max100milli = '22'
max100nano = '10'
max100pico = '04'
max10Mega = '38'
max10kilo = '32'
max10micro = '14'
```



```
max10milli = '20'  
max10nano = '08'  
max10pico = '02'  
max1Mega = '36'  
max1kilo = '30'  
max1micro = '12'  
max1milli = '18'  
max1nano = '06'  
max1pico = '00'  
max3 = '25'  
max30 = '27'  
max300 = '29'  
max300Mega = '41'  
max300kilo = '35'  
max300micro = '17'  
max300milli = '23'  
max300nano = '11'  
max300pico = '05'  
max30Mega = '39'  
max30kilo = '33'  
max30micro = '15'  
max30milli = '21'  
max30nano = '09'  
max30pico = '03'  
max3Mega = '37'  
max3kilo = '31'  
max3micro = '13'  
max3milli = '19'  
max3nano = '07'  
max3pico = '01'
```

`confirm_connection()`

Confirm a connection to the device.

Turns of bluetooth searching by confirming a connection.

Example:

```
>>> import instruments as ik  
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')  
>>> inst.confirm_connection()
```

disconnect ()

Disconnect the device.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.disconnect()
```

scale_down ()

Set scale to next lower level.

Sets the power meter to the next lower scale. If already at the lowest possible scale, no change will be made.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.scale_down()
```

scale_up ()

Set scale to next higher level.

Sets the power meter to the next higher scale. If already at the highest possible scale, no change will be made.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.scale_up()
```

anticipation

Get / Set anticipation.

This command is used to enable or disable the anticipation processing when the device is reading from a wattmeter. The anticipation is a software-based acceleration algorithm that provides faster readings using the detector's calibration.

Returns Is anticipation enabled or not.

Return type `bool`

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.anticipation
True
>>> inst.anticipation = False
```

auto_scale

Get / Set auto scale on the device.

Returns Status of auto scale enabled feature.

Return type `bool`

Raises `ValueError` – The command was not acknowledged by the device.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.auto_scale
True
>>> inst.auto_scale = False
```

available_scales

Get available scales from connected device.

Returns Scales currently available on device.

Return type *Blu.Scale*

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.available_scales
[<Scale.max100milli: '22'>, <Scale.max300milli: '23'>,
<Scale.max1: '24'>, <Scale.max3: '25'>, <Scale.max10: '26'>,
<Scale.max30: '27'>, <Scale.max100: '28'>]
```

battery_state

Get the charge state of the battery.

Returns Charge state of battery

Return type *u.percent*

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.battery_state
array(100.) * %
```

current_value

Get the currently measured value (unitful).

Returns Currently measured value

Return type *u.Quantity*

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.current_value
3.004 W
```

head_type

Get the head type information.

Returns Type of instrument head.

Return type *str*

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.head_type
'NIG : 104552, Wattmeter, V1.95'
```

measure_mode

Get the current measurement mode.

Potential return values are 'power', which indicates power mode in W and 'sse', indicating single shot energy mode in J.

Returns 'power' if in power mode, 'sse' if in single shot energy mode.

Return type `str`

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.measure_mode
'power'
```

new_value_ready

Get status if a new value is ready.

This command is used to check whether a new value is available from the device. Though optional, its use is recommended when used with single pulse operation.

Returns Is a new value ready?

Return type `bool`

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.new_value_ready
False
```

scale

Get / Set measurement scale.

The measurement scales are chosen from the the *Scale* enum class. Scales are either in watts or joules, depending on what state the power meter is currently in.

Note: Setting a scale manually will automatically turn of auto scale.

Returns Scale that is currently set.

Return type `Blu.Scale`

Raises **ValueError** – The command was not acknowledged by the device. A scale that is not available might have been selected. Use *available_scales* to display scales that are possible on your device.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.scale = inst.Scale.max3
>>> inst.scale
<Scale.max3: '25'>
```

single_shot_energy_mode

Get / Set single shot energy mode.

Returns Is single shot energy mode turned on?**Return type** bool**Example:**

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.single_shot_energy_mode
False
>>> inst.single_shot_energy_mode = True
```

trigger_level

Get / Set trigger level when in energy mode.

The trigger level must be between 0.001 and 0.998.

Returns Trigger level (absolute) with respect to the currently set scale**Return type** float**Raises** **ValueError** – Trigger level out of range.**Example:**

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.trigger_level = 0.153
>>> inst.trigger_level
0.153
```

usb_state

Get status if USB cable is connected.

Returns Is a USB cable connected?**Return type** bool**Example:**

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.usb_state
True
```

user_multiplier

Get / Set user multiplier.

Returns User multiplier

Return type u.Quantity

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.user_multiplier = 10
>>> inst.user_multiplier
10.0
```

user_offset

Get / Set user offset.

The user offset can be set unitful in watts or joules and set to the device.

Returns User offset

Return type u.Quantity

Raises **ValueError** – Unit not supported or value for offset is out of range.

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.user_offset = 10
>>> inst.user_offset
array(10.) * W
```

version

Get device information.

Returns Version and device type

Return type str

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.version
'Blu firmware Version 1.95'
```

wavelength

Get / Set the wavelength.

The wavelength can be set unitful. Specifying zero as a wavelength or providing an out-of-bound value as a parameter restores the default settings, typically 1064nm. If no units are provided, nm are assumed.

Returns Wavelength in nm

Return type u.Quantity

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.wavelength = u.Quantity(527, u.nm)
>>> inst.wavelength
array(527) * nm
```

zero_offset

Get / Set zero offset.

Gets the status if zero offset is enabled. When set to `True`, the device will read the current level immediately for around three seconds and then set the baseline to the averaged value. If activated and set to `True` again, a new value for the baseline will be established.

Returns Is zero offset enabled?

Return type `bool`

Example:

```
>>> import instruments as ik
>>> inst = ik.gentec_eo.Blu.open_serial('/dev/ttyACM0')
>>> inst.zero_offset
True
>>> inst.zero_offset = False
```

2.6 Glassman

2.6.1 GlassmanFR Single Output Power Supply

class `instruments.glassman.GlassmanFR` (*filelike*)

The GlassmanFR is a single output power supply.

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

This class should work for any of the Glassman FR Series power supplies and is also likely to work for the EJ, ET, EY and FJ Series which seem to share their communication protocols. The code has only been tested by the author with an Glassman FR50R6 power supply.

Before this power supply can be remotely operated, remote communication must be enabled and the HV must be on. Please refer to the manual.

Example usage:

```
>>> import instruments as ik
>>> psu = ik.glassman.GlassmanFR.open_serial('/dev/ttyUSB0', 9600)
>>> psu.voltage = 100 # Sets output voltage to 100V.
>>> psu.voltage
array(100.0) * V
>>> psu.output = True # Turns on the power supply
>>> psu.voltage_sense < 200 * u.volt
True
```

This code uses default values of `voltage_max`, `current_max` and `polarity` that are only valid of the FR50R6 in its positive setting. If your power supply differs, reset those values by calling:

```
>>> import instruments.units as u
>>> psu.voltage_max = 40.0 * u.kilovolt
>>> psu.current_max = 7.5 * u.milliamp
>>> psu.polarity = -1
```

class ErrorCode

Enum containing the possible error codes returned by the instrument.

checksum_error = '2'

The checksum calculated by the instrument does not correspond to the one received

extra_bytes = '3'

The command was longer than expected

illegal_control = '4'

The digital control byte set was not one of HV On, HV Off or Power supply Reset

illegal_while_fault = '5'

A send command was sent without a reset byte while the power supply is faulted

processing_error = '6'

Command valid, error while executing it

undefined_command = '1'

Undefined command received (not S, Q, V or C)

class Mode

Enum containing the possible modes of operations of the instrument

current = '1'

Constant current mode

voltage = '0'

Constant voltage mode

class ResponseCode

Enum containing the possible response codes returned by the instrument.

C = 'A'

A configure command expects an acknowledge response (A)

Q = 'R'

A query command expects a response packet (R)

S = 'A'

A set command expects an acknowledge response (A)

V = 'B'

A version query expects a different response packet (B)

get_status ()

Gets and parses the response packet.

Returns a `dict` with the following keys: {voltage, current, mode, fault, output}

Return type `dict`

query (cmd, size=-1)

Overrides the default `query` by padding the front of each command sent to the instrument with an SOH character and the back of it with a checksum.

This implementation also automatically check that the checksum returned by the instrument is consistent with the message. If the message returned is an error, it parses it and raises.

Parameters

- **cmd** (*str*) – The query message to send to the instrument
- **size** (*int*) – The number of bytes to read back from the instrument response.

Returns The instrument response to the query

Return type *str*

reset ()

Reset device to default status (HV Off, V=0.0, A=0.0)

sendcmd (*cmd*)

Overrides the default `setcmd` by padding the front of each command sent to the instrument with an SOH character and the back of it with a checksum.

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

set_status (*voltage=None, current=None, output=None, reset=False*)

Sets the requested variables on the instrument.

This instrument can only set all of its variables simultaneously, if some of them are omitted in this function, they will simply be kept as what they were set to previously.

channel

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

Return type ‘tuple’ of length 1 containing a reference back to the parent GlassmanFR object.

current

Gets/sets the output current setting.

Units As specified, or assumed to be A otherwise.

Type *float* or *Quantity*

current_sense

Gets/sets the output current as measured by the sense wires.

Units As specified, or assumed to be A otherwise.

Type *Quantity*

device_timeout

Gets/sets the timeout instrument side.

This is a toggle setting. ON will set the timeout to 1.5 seconds while OFF will disable it.

Type *bool*

fault

Gets the output status.

Returns True if the instrument has a fault.

Type *bool*

mode

Gets/sets the mode for the specified channel.

The constant-voltage/constant-current modes of the power supply are selected automatically depending on the load (resistance) connected to the power supply. If the load greater than the set V/I is connected, a voltage V is applied and the current flowing is lower than I. If the load is smaller than V/I, the set current I acts as a current limiter and the voltage is lower than V.

Type `GlassmanFR.Mode`

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`

version

The software revision level of the power supply's data interface via the V command

Return type `str`

voltage

Gets/sets the output voltage setting.

Units As specified, or assumed to be V otherwise.

Type `float` or `Quantity`

voltage_sense

Gets the output voltage as measured by the sense wires.

Units As specified, or assumed to be V otherwise.

Type `Quantity`

2.7 Holzworth

2.7.1 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 necessary 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 necessary 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`

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`

name

Gets identification string of the HS9000

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

Return type `str`

ready

Gets the ready status of the HS9000.

Returns If the instrument is ready for operation

Return type `bool`

2.8 Hewlett-Packard

2.8.1 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 = 9`

`dbm = 4`

`null = 3`

`off = 0`

```
pass_fail = 1
percent = 8
scale = 7
statistic = 2
thermistor_c = 6
thermistor_f = 5

class Mode
    Enum containing the supported mode codes

    acv = 'S0F2'
        AC voltage

    acvdcv = 'S0F3'
        RMS of DC + AC voltage

    dcv = 'S0F1'
        DC voltage

    oc_resistance_2wire = 'S1F4'
        offset compensated 2 wire resistance

    oc_resistance_4wire = 'S1F5'
        offset compensated 4 wire resistance

    ratio_acv_dcv = 'S1F2'
        ratio AC / DC voltage

    ratio_acvdcv_dcv = 'S1F3'
        ratio (AC + DC) / DC voltage

    ratio_dcv_dcv = 'S1F1'
        ratio DC / DC voltage

    resistance_2wire = 'S0F4'
        2 wire resistance

    resistance_4wire = 'S0F5'
        4 wire resistance

class Register
    Enum with the register names for all HP3456a internal registers.

    count = 'C'

    delay = 'D'

    lower = 'L'

    mean = 'M'

    nplc = 'I'

    number_of_digits = 'G'

    number_of_readings = 'N'

    r = 'R'

    upper = 'U'

    variance = 'V'
```

```
y = 'Y'
```

```
z = 'Z'
```

class TriggerMode

Enum with valid trigger modes.

```
external = 2
```

```
hold = 4
```

```
internal = 1
```

```
single = 3
```

class ValidRange

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

```
nplc = (0.1, 1.0, 10.0, 100.0)
```

```
resistance = (100.0, 1000.0, 10000.0, 100000.0, 1000000.0, 10000000.0, 100000000.0)
```

```
voltage = (0.1, 1.0, 10.0, 100.0, 1000.0)
```

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.

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*

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*

trigger()

Signal a single manual trigger event to the *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 impedance measurements since no input switching is done.

count

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

Return type *int*

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*

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.

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*

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*

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*

mean

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

Return type float

mode

Set the measurement mode.

Type *HP3456a.Mode*

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

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

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

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

relative

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

Type bool

trigger_mode

Set the trigger mode.

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

Type *HP3456a.TriggerMode*

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

variance

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

Return type float

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

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

2.8.2 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 than 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 necessary 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 necessary 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 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 = 0
```

```
voltage = 0
```

```
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.

```
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.

```
channel_count
```

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

Type `int`

```
current
```

Gets/sets the current for all four channels.

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

Type `tuple` [~ pint.Quantity, ...]` with units Amp

```
current_sense
```

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

Units A (amps)

Return type `tuple` [~ pint.Quantity, ...]`

```
voltage
```

Gets/sets the voltage for all four channels.

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

Type `tuple` [~ pint.Quantity, ...]` with units Volt

```
voltage_sense
```

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

Units V (volts)

Return type `tuple` [~ pint.Quantity, ...]`

2.8.3 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 = 60000

normal = 15000

class DFISource

Enum containing valid DFI sources for the hp6632b

event_status_bit = 'ESB'

off = 'OFF'

operation = 'OPER'

questionable = 'QUES'

request_service_bit = 'RQS'

class DigitalFunction

Enum containing valid digital function modes for the hp6632b

data = 'DIG'

remote_inhibit = 'RIDF'

class ErrorCodes

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

block_data_error = -160

block_data_not_allowed = -168

cal_not_enabled = 403

cal_password_incorrect = 402

cal_switch_prevents_cal = 401

character_data_error = -140

character_data_not_allowed = -148
character_data_too_long = -144
command_error = -100
command_header_error = -110
command_only_applic_rs232 = 602
computed_prog_cal_constants_incorrect = 405
computed_readback_cal_const_incorrect = 404
curr_or_volt_fetch_incompat_with_last_acq = 603
cv_or_cc_status_incorrect = 407
data_out_of_range = -222
data_type_error = -104
digital_io_selftest = 80
execution_error = -200
exponent_too_large = -123
expression_error = -170
expression_not_allowed = -178
front_panel_uart_buffer_overflow = 223
front_panel_uart_framing = 221
front_panel_uart_overflow = 220
front_panel_uart_parity = 222
front_panel_uart_timeout = 224
get_not_allowed = -105
header_separator_error = -111
header_suffix_out_of_range = -114
illegal_macro_label = -273
illegal_parameter_value = -224
incorrect_seq_cal_commands = 406
ingrd_rcv_buffer_overflow = 213
invalid_block_data = -161
invalid_character = -101
invalid_character_data = -141
invalid_character_in_number = -121
invalid_expression = -171
invalid_inside_macro_definition = -183
invalid_outside_macro_definition = -181
invalid_separator = -103

```
invalid_string_data = -151
invalid_suffix = -131
macro_error_180 = -180
macro_error_270 = -270
macro_execution_error = -272
macro_parameter_error = -184
macro_recursion_error = -276
macro_redefinition_not_allowed = -277
measurement_overrange = 604
missing_parameter = -109
no_error = 0
numeric_data_error = -120
numeric_data_not_allowed = -128
operation_complete = -800
out_of_memory = -225
output_mode_must_be_normal = 408
ovdac_selftest = 15
parameter_not_allowed = -108
power_on = -500
    Raised when the instrument detects that it has been turned from off to on.
program_mnemonic_too_long = -112
query_deadlocked = -430
query_error = -400
query_interrupted = -410
query_terminated = -420
query_terminated_after_indefinite_response = -440
ram_cal_checksum_failed = 3
ram_config_checksum_failed = 2
ram_rd0_checksum_failed = 1
ram_rst_checksum_failed = 5
ram_selftest = 10
ram_state_checksum_failed = 4
request_control_event = -700
rs232_recv_framing_error = 216
rs232_recv_overrun_error = 218
rs232_recv_parity_error = 217
```

```

string_data_error = -150
string_data_not_allowed = -158
suffix_error = -130
suffix_not_allowed = -138
suffix_too_long = -134
syntax_error = -102
system_error = -310
too_many_digits = -124
too_many_errors = -350
too_many_sweep_points = 601
too_much_data = -223
undefined_header = -113
unexpected_number_of_parameters = -115
user_request_event = -600
vdac_idac_selftest1 = 11
vdac_idac_selftest2 = 12
vdac_idac_selftest3 = 13
vdac_idac_selftest4 = 14

```

class RemoteInhibit
Enum containing valid remote inhibit modes for the hp6632b.

```

latching = 'LATC'
live = 'LIVE'
off = 'OFF'

```

class SenseWindow
Enum containing valid sense window modes for the hp6632b.

```

hanning = 'HANN'
rectangular = 'RECT'

```

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

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.

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.

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`

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`

digital_data

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

Type `int`

digital_function

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

Type `DigitalFunction`

display_brightness

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

Type `float`

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`

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`

line_frequency

Gets/sets the power line frequency setting for the instrument.

Returns The power line frequency

Units Hertz

Type `Quantity`

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`

output_dfi_source

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

Type `DFISource`

output_protection_delay

Get/set the time between programming of an output change that produces a constant current condition and the recording of that condition 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`

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`

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`

sense_sweep_points

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

Type `int`

sense_window

Get/set the measurement window function.

Type `SenseWindow`

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`

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`

2.8.4 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
```

(continues on next page)

(continued from previous page)

```

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) alphanumeric characters 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 sent 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`

2.8.5 HPe3631a Power Supply

class `instruments.hp.HPe3631a` (*filelike*)

The HPe3631a is a three channels voltage/current supply. - Channel 1 is a positive +6V/5A channel (P6V) - Channel 2 is a positive +25V/1A channel (P25V) - Channel 3 is a negative -25V/1A channel (N25V)

This module is designed for the power supply to be set to a specific channel and remain set afterwards as this device does not offer commands to set or read multiple channels without calling the channel set command each time (0.5s). It is possible to call a specific channel through `psu.channel[idx]`, which will automatically reset the channel id, when necessary.

This module is likely to work as is for the Agilent E3631 and Keysight E3631 which seem to be rebranded but identical devices.

Example usage:

```
>>> import instruments as ik
>>> psu = ik.hp.HPe3631a.open_gpibusb("/dev/ttyUSB0", 10)
>>> psu.channelid = 2           # Sets channel to P25V
>>> psu.voltage = 12.5         # Sets voltage to 12.5V
>>> psu.voltage                # Reads back set voltage
array(12.5) * V
>>> psu.voltage_sense          # Reads back sensed voltage
array(12.501) * V
```

class Channel (*parent, valid_set*)

Class representing a power output channel on the HPe3631a.

Warning: This class should NOT be manually created by the user. It is designed to be initialized by the *HPe3631a* class.

channel

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

Return type *HPe3631a.Channel*

See also:

HPe3631a for example using this property.

channelid

Gets/Sets the active channel ID.

Type *HPe3631a.ChannelType*

current

Gets/sets the output current of the source.

Units As specified, or assumed to be A otherwise.

Type *float* or *Quantity*

current_max

current_min

current_sense

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

Units As specified, or assumed to be A otherwise.

Type *Quantity*

mode

Gets/sets the mode for the specified channel.

The constant-voltage/constant-current modes of the power supply are selected automatically depending on the load (resistance) connected to the power supply. If the load greater than the set V/I is connected, a

voltage V is applied and the current flowing is lower than I . If the load is smaller than V/I , the set current I acts as a current limiter and the voltage is lower than V .

output

Gets/sets the outputting status of the specified channel.

This is a toggle setting. ON will turn on the channel output while OFF will turn it off.

Type `bool`

voltage

Gets/sets the output voltage of the source.

Units As specified, or assumed to be V otherwise.

Type `float` or `Quantity`

voltage_max

Gets the maximum voltage for the current channel.

Units V .

Type `Quantity`

voltage_min

Gets the minimum voltage for the current channel.

Units V .

Type `Quantity`

voltage_range

Gets the voltage range for the current channel.

The MAX function SCPI command is designed in such a way on this device that it always returns the largest absolute value. There is no need to query MIN, as it is always 0., but one has to order the values as MAX can be negative.

Units V .

Type array of `Quantity`

voltage_sense

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

Units As specified, or assumed to be V otherwise.

Type `Quantity`

2.9 Keithley

2.9.1 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 instruments.units as u
>>> 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 = 4

current_dc = 3

resistance = 2

voltage_ac = 1

voltage_dc = 0

class TriggerMode

Enum containing valid trigger modes for the Keithley 195

ext_continuous = 6

ext_one_shot = 7

get_continuous = 2

get_one_shot = 3

talk_continuous = 0

talk_one_shot = 1

x_continuous = 4

x_one_shot = 5

class ValidRange

Enum containing valid range settings for the Keithley 195

current_ac = (2e-05, 0.0002, 0.002, 0.02, 0.2, 2, 2)

current_dc = (2e-05, 0.0002, 0.002, 0.02, 0.2, 2)

resistance = (20, 200, 2000, 20000.0, 200000.0, 2000000.0, 20000000.0)

voltage_ac = (0.02, 0.2, 2, 20, 200, 700)

voltage_dc = (0.02, 0.2, 2, 20, 200, 1000)

auto_range()

Turn on auto range for the Keithley 195.

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

get_status_word()

Retrieve 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`

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 separately 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 guarantee.

Example usage:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> 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 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, delay, multiplex, 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*

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).

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*

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 equipped 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*

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*

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*

2.9.2 Keithley485 Picoammeter

class `instruments.keithley.Keithley485` (*filelike*)

The Keithley Model 485 is a 4 1/2 digit resolution autoranging picoammeter with a +- 20000 count LCD. It is designed for low current measurement requirements from 0.1pA to 2mA.

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

Example usage:

```
>>> import instruments as ik
>>> inst = ik.keithley.Keithley485.open_gpibusb("/dev/ttyUSB0", 22)
>>> inst.measure() # Measures the current
array(-1.278e-10) * A
```


class SRQDataMask

Enum containing valid SRQ data masks for the Keithley 485

busy = 16

Device busy

busy_read_done = 24

Device busy or read overflow

busy_read_done_ovf = 25

Device busy, read done or read overflow

busy_read_ovf = 17

Device busy or read overflow

read_done = 8

Read done

read_done_ovf = 9

Read done or read overflow

read_ovf = 1

Read overflow

srq_disabled = 0

Service request (SRQ) disabled

class SRQErrorMask

Enum containing valid SRQ error masks for the Keithley 485

idcc = 2

Illegal Device-Dependent Command (IDDC)

idcco = 1

Illegal Device-Dependent Command Option (IDDCO)

idcco_idcc = 3

IDDCO or IDDC

not_remote = 4

Device not in remote

not_remote_idcc = 6

Device not in remote or IDDC

not_remote_idcco = 5

Device not in remote or IDDCO

not_remote_idcco_idcc = 7

Device not in remote, IDDCO or IDDC

srq_disabled = 0

Service request (SRQ) disabled

class Status

Enum containing valid status keys in the measurement string

normal = b'N'

Measurement normal

overflow = b'O'

Measurement overflow

relative = b'Z'
Measurement relative

zerocheck = b'C'
Measurement zero-check

class TriggerMode

Enum containing valid trigger modes for the Keithley 485

continuous_onget = 2
Continuously measures current, returns on GET

continuous_ontalk = 0
Continuously measures current, returns on talk

continuous_onx = 4
Continuously measures current, returns on X

oneshot_onget = 3
Measures current once and returns on GET

oneshot_ontalk = 1
Measures current once and returns on talk

oneshot_onx = 5
Measures current once and returns on X

auto_range ()

Turn on auto range for the Keithley 485.

This is the same as calling the `Keithley485.set_current_range` method and setting the parameter to "AUTO".

get_status ()

Gets and parses the status word.

Returns a `dict` with the following keys: {zerocheck, log, range, relative, eoi, relative, trigger, datamask, errormask, terminator}

Return type `dict`

measure ()

Perform a current measurement with the Keithley 485.

Return type `Quantity`

eoi_mode

Gets/sets the 'eoi' mode (K) of the Keithley 485.

The model 485 will normally send an end of interrupt (EOI) during the last byte of its data string or status word. The EOI response of the instrument may be included or omitted. Warning: the default setting (KO) includes it.

See the Keithley 485 manual for more information.

Type `bool`

input_range

Gets/sets the range (R) of the Keithley 485 input terminals. The valid ranges are one of {auto|2e-9|2e-8|2e-7|2e-6|2e-5|2e-4|2e-3}

Type `Quantity` or `str`

log

Gets/sets the ‘log’ mode (D) of the Keithley 485.

Once log is enabled (D1 sent), the device will return the logarithm of the current readings.

See the Keithley 485 manual for more information.

Type `bool`

relative

Gets/sets the relative measurement mode (Z) of the Keithley 485.

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 currents 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`

trigger_mode

Gets/sets the trigger mode (T) of the Keithley 485.

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 current. One-shot performs a single current measurement when requested to do so.

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.

It is recommended to leave it in the default mode (T0, continuous on talk), and simply ignore the output when other commands are called.

Type `Keithley485.TriggerMode`

zero_check

Gets/sets the ‘zero check’ mode (C) of the Keithley 485.

Once zero check is enabled (C1 sent), the display can be zeroed with the REL feature or the front panel pot.

See the Keithley 485 manual for more information.

Type `bool`

2.9.3 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 = 1
    pulsed = 0
class Polarity
    Enum containing valid polarity modes for the Keithley 580
    negative = 1
    positive = 0
class TriggerMode
    Enum containing valid trigger modes for the Keithley 580
    get_continuous = 2
    get_one_shot = 3
    talk_continuous = 0
    talk_one_shot = 1
    trigger_continuous = 4
    trigger_one_shot = 5
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".
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
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 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
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,trigger, sqrndata,sqronerror,linefreq,terminator}

    Parameters statusword – Byte string to be unpacked and parsed

    Type str

    Return type dict
```

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*

sendcmd (*cmd*)

Sends a command without waiting for a response.

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

set_calibration_value (*value*)

Sets the calibration value. This is not currently implemented.

Parameters **value** – Calibration value to write

store_calibration_constants ()

Instructs the instrument to store the calibration constants. This is not currently implemented.

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).

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*

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*

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*

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`

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`

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`

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`

2.9.4 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

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*

class Mode

Enum containing valid measurement modes for the Keithley 2182

temperature = 'TEMP'

voltage_dc = 'VOLT'

class TriggerMode

Enum containing valid trigger modes for the Keithley 2182

bus = 'BUS'

external = 'EXT'

immediate = 'IMM'

manual = 'MAN'

timer = 'TIM'

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 transferred 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 *tuple` [`~pint.Quantity, ...]* or if numpy is installed, *Quantity* with *numpy.array* data

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

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*

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 * u.millivolt
```

Units As appropriate for the current mode setting.

Type *Quantity*, or *InputRange*

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*

units

Gets the current measurement units of the instrument.

Return type *Unit*

2.9.5 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 instruments.units as u
>>> import instruments as ik
>>> ccs = ik.keithley.Keithley6220.open_gpibusb("/dev/ttyUSB0", 10)
>>> ccs.current = 10 * u.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.

2.9.6 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 instruments.units as u
>>> dmm = ik.keithley.Keithley6514.open_gpibusb('/dev/ttyUSB0', 12)
```

class `ArmSource`

Enum containing valid trigger arming sources for the Keithley 6514

`bus = 'BUS'`

`immediate = 'IMM'`

`manual = 'MAN'`

`nstest = 'NST'`

`pstest = 'PST'`

```
    stest = 'STES'
    timer = 'TIM'
    tlink = 'TLIN'

class Mode
    Enum containing valid measurement modes for the Keithley 6514
    charge = 'CHAR'
    current = 'CURR:DC'
    resistance = 'RES'
    voltage = 'VOLT:DC'

class TriggerMode
    Enum containing valid trigger modes for the Keithley 6514
    immediate = 'IMM'
    tlink = 'TLINK'

class ValidRange
    Enum containing valid measurement ranges for the Keithley 6514
    charge = (2e-08, 2e-07, 2e-06, 2e-05)
    current = (2e-11, 2e-10, 2e-09, 2e-08, 2e-07, 2e-06, 2e-05, 0.0002, 0.002, 0.02)
    resistance = (2000.0, 20000.0, 200000.0, 2000000.0, 20000000.0, 200000000.0, 2000000000.0)
    voltage = (2, 20, 200)
```

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

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)

read_measurements ()

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

arm_source

Gets/sets the arm source of the Keithley 6514.

auto_range

Gets/sets the auto range setting

Type `bool`

input_range

Gets/sets the upper limit of the current range.

Type `Quantity`

mode

Gets/sets the measurement mode of the Keithley 6514.

trigger_mode

Gets/sets the trigger mode of the Keithley 6514.

unit

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

Type `Unit`

zero_check

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

zero_correct

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

2.10 Lakeshore

2.10.1 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 instruments.units as u
>>> 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`

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*

2.10.2 Lakeshore370 AC Resistance Bridge

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

The Lakeshore 370 is a multichannel AC resistance bridge for use in low temperature dilution refrigerator 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*

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*

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

class `Filter`

Enum containing valid filter modes for the Lakeshore 475

```

    lowpass = 3
    narrow = 2
    wide = 1
class Mode
    Enum containing valid measurement modes for the Lakeshore 475
    dc = 1
    peak = 3
    rms = 2
class PeakDisplay
    Enum containing valid peak displays for the Lakeshore 475
    both = 3
    negative = 2
    positive = 1
class PeakMode
    Enum containing valid peak modes for the Lakeshore 475
    periodic = 1
    pulse = 2
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.

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

    Type bool

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

field
    Read field from connected probe.

    Type Quantity

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

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

field_units

Gets/sets the units of the Gaussmeter.

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

Type Unit

i_value

Gets/sets the I value for the field control ramp.

Type float

p_value

Gets/sets the P value for the field control ramp.

Type float

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

temp_units

Gets/sets the temperature units of the Gaussmeter.

Acceptable units are celcius and kelvin.

Type Unit

2.11 Minghe

2.11.1 MHS5200 Function Generator

class instruments.minghe.MHS5200 (*filelike*)

The MHS5200 is a low-cost, 2 channel function generator.

There is no user manual, but Al Williams has reverse-engineered the communications protocol: <https://github.com/wd5gnr/mhs5200a/blob/master/MHS5200AProtocol.pdf>

class Channel (*mhs, idx*)

Class representing a channel on the MHS52000.

duty_cycle

Gets/Sets the duty cycle of this channel.

Units A fraction

Type float

enable

Gets/Sets the enable state of this channel.

Type bool

frequency

Gets/Sets the frequency of this channel.

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

function

Gets/Sets the wave type of this channel.

Type `MHS5200.Function`

offset

Gets/Sets the offset of this channel.

The fraction of the duty cycle to offset the function by.

Type `float`

phase

Gets/Sets the phase of this channel.

Units As specified (if a `Quantity`) or assumed to be of degrees. :type: `Quantity`

class Function

Enum containing valid wave modes for

`sawtooth_down = 4`

`sawtooth_up = 3`

`sine = 0`

`square = 1`

`triangular = 2`

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:

```
>>> import instruments as ik
>>> mhs = ik.minghe.MHS5200.open_serial(vid=1027, pid=24577,
```

```
baud=19200, timeout=1) >>> print(mhs.channel[0].frequency)
```

Return type `list`[`MHS5200.Channel]`

serial_number

Get the serial number, as an int

Return type `int`

2.12 Newport

2.12.1 Agilis Piezo Motor Controller

class `instruments.newport.AGUC2` (*filelike*)

Handles the communication with the AGUC2 controller using the serial connection.

Example usage:

```
>>> import instruments as ik
>>> agl = ik.newport.AGUC2.open_serial(port='COM5', baud=921600)
```

This loads a controller into the instance `agl`. The two axis are called 'X' (axis 1) and 'Y' (axis 2). Controller commands and settings can be executed as following, as examples:

Reset the controller:

```
>>> agl.reset_controller()
```

Print the firmware version:

```
>>> print(agl.firmware_version)
```

Individual axes can be controlled and queried as following:

Relative move by 1000 steps:

```
>>> agl.axis["X"].move_relative(1000)
```

Activate jogging in mode 3:

```
>>> agl.axis["X"].jog(3)
```

Jogging will continue until the axis is stopped

```
>>> agl.axis["X"].stop()
```

Query the step amplitude, then set the positive one to +10 and the negative one to -20

```
>>> print(agl.axis["X"].step_amplitude)
>>> agl.axis["X"].step_amplitude = 10
>>> agl.axis["X"].step_amplitude = -20
```

class Axes

Enumeration of valid delay channels for the AG-UC2 controller.

X = 1

Y = 2

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

This runs the query command. However, the query command often times out for this device. The response of all queries are always strings. If timeout occurs, the response will be: "Query timed out."

ag_sendcmd (*cmd*)

Sends the command, then sleeps

reset_controller ()

Resets the controller. All temporary settings are reset to the default value. Controller is put into local model.

axis

Gets a specific axis object.

The desired axis is accessed by passing an EnumValue from `Channels`. For example, to access the X axis (axis 1):


```
>>> import instruments as ik
>>> agl = ik.newport.AGUC2.open_serial(port='COM5', baud=921600)
>>> agl.axis["X"].move_relative(1000)
```

See example in *AGUC2* for a more details

Return type `_Axis`

enable_remote_mode

Gets / sets the status of remote mode.

error_previous_command

Retrieves the error of the previous command and translates it into a string. The string is returned

firmware_version

Returns the firmware version of the controller

limit_status

PARTLY UNTESTED: SEE COMMENT ABOVE

Returns the limit switch status of the controller. Possible returns are: - PH0: No limit switch is active - PH1: Limit switch of axis #1 (X) is active,

limit switch of axis #2 (Y) is not active

- **PH2: Limit switch of axis #2 (Y) is active,** limit switch of axis #1 (X) is not active
- **PH3: Limit switches of axis #1 (X) and axis #2 (Y) are active**

If device has no limit switch, this routine always returns PH0

sleep_time

The device often times out. Therefore a sleep time can be set. The routine will wait for this amount (in seconds) every time after a command or a query are sent. Setting the sleep time: Give time in seconds If queried: Returns the sleep time in seconds as a float

class `instruments.newport._Axis` (*cont, ax*)

Class representing one axis attached to a Controller. This will likely work with the AG-UC8 controller as well.

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

am_i_still (*max_retries=5*)

Function to test if an axis stands still. It queries the status of the given axis and returns True (if axis is still) or False if it is moving. The reason this routine is implemented is because the status messages can time out. If a timeout occurs, this routine will retry the query until `max_retries` is reached. If query is still not successful, an IOError will be raised.

Parameters `max_retries` (*int*) – Maximum number of retries

Returns True if the axis is still, False if the axis is moving

Return type `bool`

stop ()

Stops the axis. This is useful to interrupt a jogging motion.

zero_position ()

Resets the step counter to zero. See *number_of_steps* for details.

axis_status

Returns the status of the current axis.

jog

Start jog motion / get jog mode Defined jog steps are defined with *step_amplitude* function (default 16). If a jog mode is supplied, the jog motion is started. Otherwise the current jog mode is queried. Valid jog modes are:

-4 — Negative direction, 666 steps/s at defined step amplitude. -3 — Negative direction, 1700 steps/s at max. step amplitude. -2 — Negative direction, 100 step/s at max. step amplitude. -1 — Negative direction, 5 steps/s at defined step amplitude.

0 — No move, go to READY state. 1 — Positive direction, 5 steps/s at defined step amplitude. 2 — Positive direction, 100 steps/s at max. step amplitude. 3 — Positive direction, 1700 steps/s at max. step amplitude. 4 — Positive direction, 666 steps/s at defined step amplitude.

Returns Jog motion set

Return type `int`

move_relative

Moves the axis by nn steps / Queries the status of the axis. Steps must be given a number that can be converted to a signed integer between -2,147,483,648 and 2,147,483,647. If queried, command returns the current target position. At least this is the expected behaviour, never worked with the rotation stage.

move_to_limit

UNTESTED: SEE COMMENT ON TOP

The command functions properly only with devices that feature a limit switch like models AG-LS25, AG-M050L and AG-M100L.

Starts a jog motion at a defined speed to the limit and stops automatically when the limit is activated. See *jog* command for details on available modes.

Returns the distance of the current position to the limit in 1/1000th of the total travel.

number_of_steps

Returns the number of accumulated steps in forward direction minus the number of steps in backward direction since powering the controller or since the last ZP (zero position) command, whatever was last.

Note: The step size of the Agilis devices are not 100% repeatable and vary between forward and backward direction. Furthermore, the step size can be modified using the SU command. Consequently, the TP command provides only limited information about the actual position of the device. In particular, an Agilis device can be at very different positions even though a TP command may return the same result.

Returns Number of steps

Return type `int`

step_amplitude

Sets / Gets the *step_amplitude*.

Sets the step amplitude (step size) in positive and / or negative direction. If the parameter is positive, it will set the step amplitude in the forward direction. If the parameter is negative, it will set the step amplitude in the backward direction. You can also provide a tuple or list of two values (one positive, one negative), which will set both values. Valid values are between -50 and 50, except for 0.

Returns Tuple of first negative, then positive step amplitude response.

Return type `(int, int)`

step_delay

Sets/gets the step delay of stepping mode. The delay applies for both positive and negative directions. The delay is programmed as multiple of 10 μ s. For example, a delay of 40 is equivalent to 40 x 10 μ s = 400 μ s. The maximum value of the parameter is equal to a delay of 2 seconds between pulses. By default, after reset, the value is 0. Setter: value must be integer between 0 and 200000 included

Returns Step delay

Return type `int`

2.12.2 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 (*program_id*)

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 (*errcheck=True*)

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 `pint.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/s²)
- ‘acceleration’ = acceleration (U/s²)
- ‘velocity’ = velocity (U/s)
- ‘deceleration’ = set deceleration (U/s²)
- ‘error_threshold’ = set error threshold (U)
- ‘estop_deceleration’ = estop deceleration (U/s²)
- ‘jerk’ = jerk rate (U/s³)
- ‘proportional_gain’ = PID proportional gain (optional)
- ‘derivative_gain’ = PID derivative gain (optional)
- ‘integral_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' = between 0 to 2e9
- 'microstep_factor' = axis microstep factor
- 'reduce_motor_torque_time' = time (ms) between 0 and 60000,
- 'reduce_motor_torque_percentage' = percentage between 0 and 100

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 deceleration. Max deacceleration 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 = `<Unit('microinch')>`

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 `Unit` 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 = 1

Search for combined Home and Index signals.

home_signal_only = 2

Search only for the Home signal.

neg_index_signals = 6

Search for the negative limit and Index signals.

neg_limit_signal = 4

Search for the negative limit signal.

pos_index_signals = 5

Search for the positive limit and Index signals.

pos_limit_signal = 3

Search for the positive limit signal.

zero_position_count = 0

Search along specified axes for the +0 position.

2.12.3 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 particular axis.

Type `int`

errcode

Gets the error code reported by the device.

Type `int`

`messageDict = {'0': 'NO ERROR DETECTED', '1': 'PCI COMMUNICATION TIME-OUT', '10': '...`

`start_time = datetime.datetime(2020, 12, 19, 5, 0, 50, 120938)`

timestamp

Returns the timestamp reported by the device as the time at which this error occurred.

Type `datetime`

2.13 Ondax

2.13.1 LM Ondax SureLock Laser Module

class `instruments.ondax.LM` (*filelike*)

The LM is the Ondax SureLock VHG-stabilized laser diode.

The user manual can be found on the [Ondax website](#).

class `Status`

Enum containing the valid states of the laser

`calibrate = 4`

`inner_modulation = 2`

`normal = 1`

`power_scan = 3`

`shutdown_current = 5`

`shutdown_overheat = 6`

`waiting = 8`

`waiting_stable_temperature = 7`

reset ()

Reset the laser controller.

save ()

Save current settings in flash memory.

current

Gets/sets the laser diode current, in mA.

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

Type `Quantity`

enabled

Gets/sets the laser emission enabled status.

Type `bool`

firmware

Gets the laser system firmware version.

Type `str`

maximum_current

Get/Set the maximum laser diode current in mA. If the current is set over the limit, the laser will shut down.

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

Type `Quantity`

power

Get/Set the laser's optical power in mW.

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

Return type `Quantity`

serial_number

Gets the laser controller serial number

Type `str`

status

Read laser controller run status.

Type `LM.Status`

temperature

Gets/sets laser diode temperature.

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

Type `Quantity`

2.14 Oxford

2.14.1 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`

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`

2.15 PhaseMatrix

2.15.1 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 instruments.units as u
>>> inst = ik.phasematrix.PhaseMatrixFSW0020.open_serial("/dev/ttyUSB0",
↳ baud=115200)
>>> inst.frequency = 1 * u.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 $5\mu\text{s}$.

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

Gets/sets the output phase of the signal generator channel

Type `Quantity`

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`

2.16 Picowatt

2.16.1 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 = 1`

`ground = 0`

`reference = 2`

class `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`

display

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

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

Type `int`

excitation

Gets/sets the excitation sensor number.

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

Type `int`

input_source

Gets/sets the input source.

Type `PicowattAVS47.InputSource`

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`

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`

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.

2.17 Qubitekk

2.17.1 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`

clear_counts ()

Clears the current total counts on the counters.

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`

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`

delay

Gets/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 `Quantity`

Returns the delay value

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`

firmware

Gets the firmware version

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

gate

Gets/sets the gate enable status

Type `bool`

subtract

Gets/sets the subtract enable status

Type `bool`

trigger_mode

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

Type `CC1.TriggerMode`

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`

2.17.2 MC1 Motor Controller

class `instruments.qubitekk.MC1` (*filelike*)

The MC1 is a controller for the qubitekk motor controller. Used with a linear actuator to perform a HOM dip.

class `MotorType`

Enum for the motor types for the MC1

radio = `'Radio'`

relay = `'Relay'`

center ()

Commands the motor to go to the center of its travel range

is_centering()

Query whether the motor is in its centering phase

Returns False if not centering, True if centering

Return type `bool`

move(*new_position*)

Move to a specified location. Position is unitless and is defined as the number of motor steps. It varies between motors.

Parameters `new_position` (`Quantity`) – the location

reset()

Sends the stage to the limit of one of its travel ranges

controller

Get the motor controller type.

direction

Get the internal direction variable, which is a function of how far the motor needs to go.

Type `Quantity`

Units milliseconds

firmware

Gets the firmware version

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

increment

Gets/sets the stepping increment value of the motor controller

Units As specified, or assumed to be of units milliseconds

Type `Quantity`

inertia

Gets/Sets the amount of force required to overcome static inertia. Must be between 0 and 100 milliseconds.

Type `Quantity`

Units milliseconds

internal_position

Get the internal motor state position, which is equivalent to the total number of milliseconds that voltage has been applied to the motor in the positive direction minus the number of milliseconds that voltage has been applied to the motor in the negative direction.

Type `Quantity`

Units milliseconds

lower_limit

Gets/sets the stepping lower limit value of the motor controller

Units As specified, or assumed to be of units milliseconds

Type `Quantity`

metric_position

Get the estimated motor position, in millimeters.

Type `Quantity`

Units millimeters

move_timeout

Get the motor's timeout value, which indicates the number of milliseconds before the motor can start moving again.

Type `Quantity`

Units milliseconds

setting

Gets/sets the output port of the optical switch. 0 means input 1 is directed to output 1, and input 2 is directed to output 2. 1 means that

input 1 is directed to output 2 and input 2 is directed to output 1.

Type `int`

step_size

Gets/Sets the number of milliseconds per step. Must be between 1 and 100 milliseconds.

Type `Quantity`

Units milliseconds

upper_limit

Gets/sets the stepping upper limit value of the motor controller

Units As specified, or assumed to be of units milliseconds

Type `Quantity`

2.18 Rigol

2.18.1 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 = 'AVER'`

`normal = 'NORM'`

`peak_detect = 'PEAK'`

class 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.

class Coupling

Enum containing valid coupling modes for the Rigol DS1000

ac = 'AC'

dc = 'DC'

ground = 'GND'

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 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*)

Gets the waveform of the specified data source channel. This is an abstract property.

Parameters **bin_format** (*bool*) – If the waveform should be transferred in binary (*True*) or ASCII (*False*) formats.

Returns The waveform with both x and y components.

Return type *numpy.ndarray*

name

Gets the name of the channel. This is an abstract property.

Type *str*

force_trigger()

Forces a trigger event to occur on the attached oscilloscope.

release_panel()

Releases any lockout of the local control panel.

run()

Starts running the oscilloscope trigger.

stop()

Stops running the oscilloscope trigger.

acquire_averages

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

Type `int`

acquire_type

channel

Gets an iterator or list for easy Pythonic access to the various channel objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

math

Gets an iterator or list for easy Pythonic access to the various math data sources objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

panel_locked

ref

Gets an iterator or list for easy Pythonic access to the various ref data sources objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

2.19 Stanford Research Systems

2.19.1 SRS345 Function Generator

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

The SRS DS345 is a 30MHz function generator.

Example usage:

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

class `Function`

Enum containing valid output function modes for the SRS 345

`arbitrary = 5`

`noise = 4`

`ramp = 3`

`sinusoid = 0`

`square = 1`

triangle = 2

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 `Function`

offset
Gets/sets the offset voltage for the output waveform.
Units As specified, or assumed to be V otherwise.
Type `float` or `Quantity`

phase
Gets/sets the phase for the output waveform.
Units As specified, or assumed to be degrees (°) otherwise.
Type `float` or `Quantity`

2.19.2 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 instruments.units as u
>>> srs = ik.srs.SRS830.open_gpibusb('/dev/ttyUSB0', 1)
>>> srs.frequency = 1000 * u.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 = 1

one_shot = 0

class `Coupling`
Enum for the SRS830 channel coupling settings.

ac = 0

dc = 1

class `FreqSource`
Enum for the SRS830 frequency source settings.

external = 0

internal = 1

class `Mode`
Enum containing valid modes for the SRS 830

```
aux1 = 'aux1'  
aux2 = 'aux2'  
aux3 = 'aux3'  
aux4 = 'aux4'  
ch1 = 'ch1'  
ch2 = 'ch2'  
none = 'none'  
r = 'r'  
ref = 'ref'  
theta = 'theta'  
x = 'x'  
xnoise = 'xnoise'  
y = 'y'  
ynoise = 'ynoise'
```

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 {XIYIR}.

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.

clear_data_buffer ()

Clears the data buffer of the 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: {XIYIR|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: {XIYIR|THETA|AUX1|AUX2|AUX3|AUX4|REF|CH1|CH2}

Return type *list*

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 $n \in \{-4\dots +9\}$ 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.

pause()

Has the instrument pause data capture.

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 `tuple`[`float, ...]` or if `numpy` is installed, `numpy.array`

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}

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}.

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.

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.

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 `tuple`[`tuple`[`float,...], tuple`[`float,...]]` or if `numpy` is installed, `numpy.array`[`numpy.array, numpy.array]`

amplitude

Gets/set the amplitude of the internal reference signal.

Set value should be $0.004 \leq \text{newval} \leq 5.000$

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.

amplitude_max

amplitude_min

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](#)

coupling

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

Type [SRS830.Coupling](#)

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`

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.

frequency_source

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

Type [SRS830.FreqSource](#)

input_shield_ground

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

Type `bool`

num_data_points

Gets the number of data sets in the SRS830 buffer.

Type `int`

phase

Gets/set the phase of the internal reference signal.

Set value should be $-360\text{deg} \leq \text{newval} < +730\text{deg}$.

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

Type `Quantity` with units degrees.

phase_max**phase_min****sample_rate**

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

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

Type `Quantity` with units Hertz.

2.19.3 SRSC100 Cryogenic Temperature Controller

class `instruments.srs.SRSC100` (*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 If numpy is installed, tuple of 2x `Quantity`, each comprised of a numpy array (`numpy.ndarray`). Else, tuple` [`tuple` [~pint.Quantity, ...], tuple` [~pint.Quantity, ...]]`

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** (*Unit*) – 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 `Unit`

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 SensorType

Enum containing valid sensor types for the SRS CTC-100

`diode = 'Diode'`

`rox = 'ROX'`

`rtd = 'RTD'`

`thermistor = 'Thermistor'`

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 `u.dimensionless`.

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

clear_log()

Clears the SRS CTC100 log

Not sure if this works.

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

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*

sendcmd (*cmd*)

Sends a command without waiting for a response.

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

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 `SRSC100._channel_names()` function.

Type *SRSC100.Channel*

display_figures

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

Type *int*

error_check_toggle

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

Bool

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

class Channels

Enumeration of valid delay channels for the DDG.

```
A = 2
B = 3
C = 4
D = 5
E = 6
F = 7
```

```
G = 8
H = 9
T0 = 0
T1 = 1
```

class DisplayMode
Enumeration of possible modes for the physical front-panel display.

```
adv_triggering_enable = 4
burst_T0_config = 14
burst_count = 9
burst_delay = 8
burst_mode = 7
burst_period = 10
channel_delay = 11
channel_levels = 12
channel_polarity = 13
prescale_config = 6
trigger_holdoff = 5
trigger_line = 3
trigger_rate = 0
trigger_single_shot = 2
trigger_threshold = 1
```

class LevelPolarity
Polarities for output levels.

```
negative = 0
positive = 1
```

class 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.

level_offset
Amplitude offset (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 `SRS DG645.LevelPolarity`

class Outputs
Enumeration of valid outputs from the DDG.

```

AB = 1
CD = 2
EF = 3
GH = 4
T0 = 0

```

class TriggerSource

Enumeration of the different allowed trigger sources and modes.

```

external_falling = 2
external_rising = 1
internal = 0
line = 6
single_shot = 5
ss_external_falling = 4
ss_external_rising = 3

```

burst_count

Gets/sets the burst count. When burst mode is enabled, the DG645 outputs burst count delay cycles per trigger. Valid numbers for burst count are between 1 and $2^{32} - 1$

burst_delay

Gets/sets the burst delay. When burst mode is enabled the DG645 delays the first burst pulse relative to the trigger by the burst delay. The burst delay may range from 0 ps to < 2000 s with a resolution of 5 ps.

Units Assume seconds if no units given.

burst_period

Gets/sets the burst period. The burst period sets the time between delay cycles during a burst. The burst period may range from 100 ns to 2000 – 10 ns in 10 ns steps.

Units Assume seconds if no units given.

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*

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*

enable_adv_triggering

Gets/sets whether advanced triggering is enabled.

Type `bool`

enable_burst_mode

Gets/sets whether burst mode is enabled.

Type `bool`

enable_burst_t0_first

Gets/sets whether T0 output in burst mode is on first. If enabled, the T0 output is enabled for first delay cycle of the burst only. If disabled, the T0 output is enabled for all delay cycles of the burst.

Type `bool`

holdoff

Gets/sets the trigger holdoff time.

Type `Quantity` or `float`

Units As passed, or s if not specified.

output

Gets the specified output port.

Type `SRSDG645.Output`

trigger_rate

Gets/sets the rate of the internal trigger.

Type `Quantity` or `float`

Units As passed or Hz if not specified.

trigger_source

Gets/sets the source for the trigger.

Type `SRSDG645.TriggerSource`

class `instruments.srs._SRSDG645Channel` (*ddg, chan*)

Class representing a sensor attached to the SRS DG645.

Warning: This class should NOT be manually created by the user. It is designed to be initialized by the `SRSDG645` class.

delay

Gets/sets the delay of this channel. Formatted as a two-tuple of the reference and the delay time. For example, `(SRSDG645.Channels.A, u.Quantity(10, "ps"))` indicates a delay of 10 picoseconds from delay channel A.

Units Assume seconds if no units given.

idx

Gets the channel identifier number as used for communication

Returns The communication identification number for the specified channel

Return type `int`

2.20 Tektronix

2.20.1 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 `Polarity`

Enum containing valid polarity modes for the AWG2000

inverted = 'INVERTED'

normal = 'NORMAL'

class `Shape`

Enum containing valid waveform shape modes for hte AWG2000

pulse = 'PULSE'

ramp = 'RAMP'

sine = 'SINUSOID'

```
square = 'SQUARE'
```

```
triangle = 'TRIANGLE'
```

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.

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*

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*

2.20.2 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 = 'AC'
```

```
dc = 'DC'
```

```
ground = 'GND'
```


force_trigger()

Forces a trigger event to occur on the attached oscilloscope. Note that this is distinct from the standard SCPI *TRG functionality.

aquisition_continuous

Gets/sets whether the aquisition is continuous (“run/stop mode”) or whether aquisition halts after the next sequence (“single mode”).

Type `bool`

aquisition_length

Gets/sets the aquisition length of the oscilloscope

Type `int`

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`

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`

data_source

Gets/sets the the data source for waveform transfer.

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`

math

Gets a data source object corresponding to the MATH channel.

Return type `_TekDPO4104DataSource`

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`

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 transferred in a binary format. Otherwise, data is transferred in ASCII.

Return type `tuple`[`tuple`[`~pint.Quantity, ...], tuple`[`~pint.Quantity, ...]]` or if numpy is installed, `tuple` of two `Quantity` with `numpy.array` data

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`

2.20.3 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 = 'AVE'

```

envelope = 'ENV'
hi_res = 'HIR'
peak_detect = 'PEAK'
sample = 'SAM'
waveform_db = 'WFMDB'

class AcquisitionState
    Enum containing valid acquisition states for the Tektronix 70000 series oscilloscopes.

    off = 'OFF'
    on = 'ON'
    run = 'RUN'
    stop = 'STOP'

class BinaryFormat
    Enum containing valid binary formats for the Tektronix 70000 series oscilloscopes (int, unsigned-int,
    floating-point).

    float = 'FP'
    int = 'RI'
    uint = 'RP'

class ByteOrder
    Enum containing valid byte order (big-/little-endian) for the Tektronix 70000 series oscilloscopes.

    big_endian = 'MSB'
    little_endian = 'LSB'

class 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 = 'AC'
    dc = 'DC'
    dc_reject = 'DCREJ'
    ground = 'GND'

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`

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

bandwidth

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
```

deskew

label

Just a human readable label for the channel.

label_xpos

The x position, in divisions, to place the label.

label_ypos

The y position, in divisions, to place the label.

offset

The vertical offset in units of volts. Voltage is given by $\text{offset} + \text{scale} * (5 * \text{raw} / 2^{15} - \text{position})$.

position

The vertical position, in divisions from the center graticule, ranging from -8 to 8. Voltage is given by $\text{offset} + \text{scale} * (5 * \text{raw} / 2^{15} - \text{position})$.

scale

Vertical channel scale in units volts/division. Voltage is given by $\text{offset} + \text{scale} * (5 * \text{raw} / 2^{15} - \text{position})$.

termination

class 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*)

Gets the waveform of the specified data source channel. This is an abstract property.

Parameters **bin_format** (*bool*) – If the waveform should be transferred in binary (`True`) or ASCII (`False`) formats.

Returns The waveform with both x and y components.

Return type `numpy.ndarray`

name

Gets the name of the channel. This is an abstract property.

Type `str`

class HorizontalMode

Enum containing valid horizontal scan modes for the Tektronix 70000 series oscilloscopes.

```
auto = 'AUTO'
constant = 'CONST'
manual = 'MAN'
```

class 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 = 'CENT'
shifted = 'SHIF'
```

class Mag

Enum containing valid amplitude units for a math channel on the TekDPO70000 series oscilloscope.

```
db = 'DB'
dbm = 'DBM'
linear = 'LINEA'
```

class Phase

Enum containing valid phase units for a math channel on the TekDPO70000 series oscilloscope.

```
degrees = 'DEG'
group_delay = 'GROUPD'
radians = 'RAD'
```

class SpectralWindow

Enum containing valid spectral windows for a math channel on the TekDPO70000 series oscilloscope.

```
blackman_harris = 'BLACKMANH'
flattop2 = 'FLATTOP2'
gaussian = 'GAUSS'
hamming = 'HAMM'
hanning = 'HANN'
kaiser_besse = 'KAISERB'
rectangular = 'RECTANG'
tek_exponential = 'TEKEXP'
```

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*

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

autoscale

Enables or disables the auto-scaling of new math waveforms.

define

A text string specifying the math to do, ex. CH1+CH2

filter_mode

filter_risetime

label

Just a human readable label for the channel.

label_xpos

The x position, in divisions, to place the label.

label_ypos

The y position, in divisions, to place the label.

num_avg

The number of acquisitions over which exponential averaging is performed.

position

The vertical position, in divisions from the center graticule.

scale

The scale in volts per division. The range is from $100e-36$ to $100e+36$.

spectral_center

The desired frequency of the spectral analyzer output data span in Hz.

spectral_gatepos

The gate position. Units are represented in seconds, with respect to trigger position.

spectral_gatewidth

The time across the 10-division screen in seconds.

spectral_lock

spectral_mag

Whether the spectral magnitude is linear, db, or dbm.

spectral_phase

Whether the spectral phase is degrees, radians, or group delay.

spectral_reflevel

The value that represents the topmost display screen graticule. The units depend on **spectral_mag**.

spectral_reflevel_offset

spectral_resolution_bandwidth

The desired resolution bandwidth value. Units are represented in Hertz.

spectral_span
Specifies the frequency span of the output data vector from the spectral analyzer.

spectral_suppress
The magnitude level that data with magnitude values below this value are displayed as zero phase.

spectral_unwrap
Enables or disables phase wrapping.

spectral_window

threshold
The math threshold in volts

unit_string
Just a label for the units... doesn't actually change anything.

class SamplingMode
Enum containing valid sampling modes for the Tektronix 70000 series oscilloscopes.

equivalent_time_allowed = 'ET'

interpolation_allowed = 'IT'

real_time = 'RT'

class StopAfter
Enum containing valid stop condition modes for the Tektronix 70000 series oscilloscopes.

run_stop = 'RUNST'

sequence = 'SEQ'

class TriggerState
Enum containing valid trigger states for the Tektronix 70000 series oscilloscopes.

armed = 'ARMED'

auto = 'AUTO'

dpo = 'DPO'

partial = 'PARTIAL'

ready = 'READY'

class WaveformEncoding
Enum containing valid waveform encoding modes for the Tektronix 70000 series oscilloscopes.

ascii = 'ASCII'

binary = 'BINARY'

force_trigger()
Forces a trigger event to happen for the oscilloscope.

run()
Enables the trigger for the oscilloscope.

select_fastest_encoding()
Sets the encoding for data returned by this instrument to be the fastest encoding method consistent with the current data source.

stop()
Disables the trigger for the oscilloscope.

HOR_DIVS = 10

VERT_DIVS = 10

acquire_enhanced_enob

Valid values are AUTO and OFF.

acquire_enhanced_state

acquire_interp_8bit

Valid values are AUTO, ON and OFF.

acquire_magnivu

acquire_mode

acquire_mode_actual

acquire_num_acquisitions

The number of waveform acquisitions that have occurred since starting acquisition with the AC-Quire:STATE RUN command

acquire_num_avgs

The number of waveform acquisitions to average.

acquire_num_envelop

The number of waveform acquisitions to be enveloped

acquire_num_frames

The number of frames acquired when in FastFrame Single Sequence and acquisitions are running.

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.

acquire_sampling_mode

acquire_state

This command starts or stops acquisitions.

acquire_stop_after

This command sets or queries whether the instrument continually acquires acquisitions or acquires a single sequence.

channel

Gets an iterator or list for easy Pythonic access to the various channel objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

data_framestart

data_framestop

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`

data_start

The first data point that will be transferred, which ranges from 1 to the record length.

data_stop

The last data point that will be transferred.

data_sync_sources

horiz_acq_duration

The duration of the acquisition.

horiz_acq_length

The record length.

horiz_delay_mode**horiz_delay_pos**

The percentage of the waveform that is displayed left of the center graticule.

horiz_delay_time

The base trigger delay time setting.

horiz_interp_ratio

The ratio of interpolated points to measured points.

horiz_main_pos

The percentage of the waveform that is displayed left of the center graticule.

horiz_mode**horiz_pos**

The position of the trigger point on the screen, left is 0%, right is 100%.

horiz_record_length

The record length in samples. See *horiz_mode*; manual mode lets you change the record length, while the length is readonly for auto and constant mode.

horiz_record_length_lim

The record length limit in samples.

horiz_roll

Valid arguments are AUTO, OFF, and ON.

horiz_sample_rate

The sample rate in samples per second.

horiz_scale

The horizontal scale in seconds per division. The horizontal scale is readonly when *horiz_mode* is manual.

horiz_unit**math**

Gets an iterator or list for easy Pythonic access to the various math data sources objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

outgoing_binary_format

Controls the data type of samples when transferring waveforms from the instrument to the host using binary encoding.

outgoing_byte_order

Controls whether binary data is returned in little or big endian.

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.

outgoing_waveform_encoding

Controls the encoding used for outgoing waveforms (instrument → host).

ref

Gets an iterator or list for easy Pythonic access to the various ref data sources objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

trigger_state

2.20.4 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 = 'AC'`

`dc = 'DC'`

`ground = 'GND'`

force_trigger()

Forces a trigger event to occur on the attached oscilloscope.

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`

data_source

Gets/sets the the data source for waveform transfer.

data_width

Gets/sets the byte-width of the data points being returned by the instrument. Valid widths are 1 or 2.

Type `int`

math

Gets a data source object corresponding to the MATH channel.

Return type `_TekTDS224DataSource`

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`

2.20.5 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 = 'FUL'`

`OneHundred = 'HUN'`

`Twenty = 'TWE'`

`TwoHundred = 'TWO'`

`class Coupling`

Available coupling options for input sources and trigger

`ac = 'AC'`

`dc = 'DC'`

`ground = 'GND'`

`class Edge`

Available Options for trigger slope

`Falling = 'FALL'`

`Rising = 'RIS'`

`class Impedance`

Available options for input source impedance

`Fifty = 'FIF'`

`OneMeg = 'MEG'`

`class Source`

Available Data sources

`CH1 = 'CH1'`

`CH2 = 'CH2'`

`CH3 = 'CH3'`

`CH4 = 'CH4'`

```
Math1 = 'MATH1'  
Math2 = 'MATH2'  
Math3 = 'MATH3'  
Ref1 = 'REF1'  
Ref2 = 'REF2'  
Ref3 = 'REF3'  
Ref4 = 'REF4'
```

class Trigger

Available Trigger sources (AUX not Available on TDS520A/TDS540A)

```
AUX = 'AUX'  
CH1 = 'CH1'  
CH2 = 'CH2'  
CH3 = 'CH3'  
CH4 = 'CH4'  
LINE = 'LINE'
```

force_trigger()

Forces a trigger event to occur on the attached oscilloscope.

get_hardcopy()

Gets a screenshot of the display

Return type `string`

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`

clock

Get/Set oscilloscope clock

Type `datetime.datetime`

data_source

Gets/sets the the data source for waveform transfer.

Type `TekTDS5xx.Source` or `_TekTDS5xxDataSource`

Return type `'_TekTDS5xxDataSource'`

data_width

Gets/Sets the data width for waveform transfers

Type `int`

display_clock

Get/Set the visibility of clock on the display

Type `bool`

horizontal_scale

Get/Set Horizontal Scale

Type `float`

math

Gets a data source object corresponding to the MATH channel.

Return type `_TekTDS5xxDataSource`

measurement

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

Return type `_TDS5xxMeasurement`

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`

sources

Returns list of all active sources

Return type `list`

trigger_coupling

Get/Set trigger coupling

Type `TekTDS5xx.Coupling`

trigger_level

Get/Set trigger level

Type `float`

trigger_slope

Get/Set trigger slope

Type `TekTDS5xx.Edge`

trigger_source

Get/Set trigger source

Type `TekTDS5xx.Trigger`

2.21 Teledyne-LeCroy

2.21.1 MAUI Oscilloscope Controller

class `instruments.teledyne.MAUI` (*filelike*)

Medium to high-end Teledyne-Lecroy Oscilloscopes are shipped with the MAUI user interface. This class can

be used to communicate with these instruments.

By default, the IEEE 488.2 commands are used. However, commands based on MAUI's app definition can be submitted too using the appropriate send / query commands.

Your scope must be set up to communicate via LXI (VXI11) to be used with pyvisa. Make sure that either the pyvisa-py or the NI-VISA backend is installed. Please see the pyvisa documentation for more information.

This class inherits from: Oscilloscope

Example usage (more examples below):

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> # start the trigger in automatic mode
>>> inst.run()
>>> print(inst.trigger_state) # print the trigger state
<TriggerState.auto: 'AUTO'>
>>> # set timebase to 20 ns per division
>>> inst.time_div = u.Quantity(20, u.ns)
>>> # call the first oscilloscope channel
>>> channel = inst.channel[0]
>>> channel.trace = True # turn the trace on
>>> channel.coupling = channel.Coupling.dc50 # coupling to 50 Ohm
>>> channel.scale = u.Quantity(1, u.V) # vertical scale to 1V/division
>>> # transfer a waveform into xdat and ydat:
>>> xdat, ydat = channel.read_waveform()
```

class Channel (*parent, idx*)

Class representing a channel on a MAUI oscilloscope.

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

class Coupling

Enum containing valid coupling modes for the oscilloscope channel. 1 MOhm and 50 Ohm are included.

ac1M = 'A1M'

dc1M = 'D1M'

dc50 = 'D50'

ground = 'GND'

query (*cmd, size=-1*)

Executes the given query. Wraps commands sent from property factories in this class with identifiers for the specified channel.

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*

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

coupling

Gets/sets the coupling for the specified channel.

Example usage:

```
>>> import instruments as ik
>>> address = "TCPIP0::192.168.0.10::INSTR"
>>> inst = inst = ik.teledyne.MAUI.open_visa(address)
>>> channel = inst.channel[0]
>>> channel.coupling = channel.Coupling.dc50
```

offset

Sets/gets the vertical offset of the specified input channel.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> channel = inst.channel[0] # set up channel
>>> channel.offset = u.Quantity(-1, u.V)
```

scale

Sets/gets the vertical scale of the channel.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> channel = inst.channel[0] # set up channel
>>> channel.scale = u.Quantity(20, u.mV)
```

class DataSource (*parent, name*)

Class representing a data source (channel, math, ref) on a MAUI oscilloscope.

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

read_waveform (*bin_format=False, single=True*)

Reads the waveform and returns an array of floats with the data.

Parameters

- **bin_format** (*bool*) – Not implemented, always False
- **single** (*bool*) – Run a single trigger? Default True. In case a waveform from a channel is required, this option is recommended to be set to True. This means that the acquisition system is first stopped, a single trigger is issued, then the waveform is transferred, and the system is set back into the state it was in before. If sampling math with multiple samples, set this to false, otherwise the sweeps are cleared by the oscilloscope prior when a single trigger command is issued.

Returns Data (time, signal) where time is in seconds and signal in V

Return type tuple`[\`tuple`[\`~pint.Quantity, ...], tuple`[\`~pint.Quantity, ...]] or if numpy is installed, tuple`[\`numpy.array, numpy.array]

Raises `NotImplementedError` – Bin format was chosen, but it is not implemented.

Example usage:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> channel = inst.channel[0] # set up channel
>>> xdat, ydat = channel.read_waveform() # read waveform
```

name

Gets the name of the channel. This is an abstract property.

Type `str`

trace

Gets/Sets if a given trace is turned on or off.

Example usage:

```
>>> import instruments as ik
>>> address = "TCPIP0::192.168.0.10::INSTR"
>>> inst = inst = ik.teledyne.MAUI.open_visa(address)
>>> channel = inst.channel[0]
>>> channel.trace = False
```

class Math (*parent, idx*)

Class representing a function on a MAUI oscilloscope.

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

class Operators (*parent*)

Sets the operator for a given channel. Most operators need a source `src`. If the source is given as an integer, it is assume that the a signal channel is requested. If you want to select another math channel for example, you will need to specify the source as a tuple: Example: `src=('f', 0)` would represent the first function channel (called F1 in the MAUI manual). A channel could be selected by calling `src=('c', 1)`, which would request the second channel (oscilloscope channel 2). Please consult the oscilloscope manual / the math setup itself for further possibilities.

Note: Your oscilloscope might not have all functions

that are described here. Also: Not all possibilities are currently implemented. However, extension of this functionality should be simple when following the given structure

absolute (*src*)

Absolute of wave form.

Parameters `src` (*int, tuple*) – Source, see info above

average (*src, average_type='summed', sweeps=1000*)

Average of wave form.

Parameters

- `src` (*int, tuple*) – Source, see info above
- `average_type` (*str*) – summed or continuous
- `sweeps` (*int*) – In summed mode, how many sweeps to collect. In continuous mode the weight of each sweep is equal to $1/1 \text{ `sweeps}$

derivative (*src, vscale=1000000.0, voffset=0, autoscale=True*)

Derivative of waveform using subtraction of adjacent samples. If `vscale` and `voffset` are unitless, `V/s` are assumed.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **vscale** (*float*) – vertical units to display (V/s)
- **voffset** (*float*) – vertical offset (V/s)
- **autoscale** (*bool*) – auto scaling of vscale, voffset?

difference (*src1, src2, vscale_variable=False*)

Difference between two sources, src1-src2.

Parameters

- **src1** (*int, tuple*) – Source 1, see info above
- **src2** (*int, tuple*) – Source 2, see info above
- **vscale_variable** (*bool*) – Horizontal and vertical scale for addition and subtraction must be identical. Allow for variable vertical scale in result?

envelope (*src, sweeps=1000, limit_sweeps=True*)

Highest and lowest Y values at each X in N sweeps.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **sweeps** (*int*) – Number of sweeps
- **limit_sweeps** (*bool*) – Limit the number of sweeps?

eres (*src, bits=0.5*)

Smoothing function defined by extra bits of resolution.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **bits** (*float*) – Number of bits. Possible values are (0.5, 1.0, 1.5, 2.0, 2.5, 3.0). If not in list, default to 0.5.

fft (*src, type='powerspectrum', window='vonhann', suppress_dc=True*)

Fast Fourier Transform of signal.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **type** (*str*) – Type of power spectrum. Possible options are: ['real', 'imaginary', 'magnitude', 'phase', 'powerspectrum', 'powerdensity']. Default: 'powerspectrum'
- **window** (*str*) – Window. Possible options are: ['blackmanharris', 'flattop', 'hamming', 'rectangular', 'vonhann']. Default: 'vonhann'
- **suppress_dc** (*bool*) – Suppress DC?

floor (*src, sweeps=1000, limit_sweeps=True*)

Lowest vertical value at each X value in N sweeps.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **sweeps** (*int*) – Number of sweeps
- **limit_sweeps** (*bool*) – Limit the number of sweeps?

integral (*src, multiplier=1, adder=0, vscale=0.001, voffset=0*)

Integral of waveform.

Parameters

- **src** (*int, tuple*) – Source, see info above
- **multiplier** (*float*) – 0 to 1e15
- **adder** (*float*) – 0 to 1e15
- **vscale** (*float*) – vertical units to display (Wb)
- **voffset** (*float*) – vertical offset (Wb)

invert (*src*)

Inversion of waveform (-waveform).

Parameters `src` (*int, tuple*) – Source, see info above

product (*src1, src2*)

Product of two sources, `src1`*`src2`.

Parameters

- `src1` (*int, tuple*) – Source 1, see info above
- `src2` (*int, tuple*) – Source 2, see info above

ratio (*src1, src2*)

Ratio of two sources, `src1/src2`.

Parameters

- `src1` (*int, tuple*) – Source 1, see info above
- `src2` (*int, tuple*) – Source 2, see info above

reciprocal (*src*)

Reciprocal of waveform (1/waveform).

Parameters `src` (*int, tuple*) – Source, see info above

rescale (*src, multiplier=1, adder=0*)

Rescales the waveform (w) in the style. `multiplier * w + adder`

Parameters

- `src` (*int, tuple*) – Source, see info above
- `multiplier` (*float*) – multiplier
- `adder` (*float*) – addition in V or assuming V

roof (*src, sweeps=1000, limit_sweeps=True*)

Highest vertical value at each X value in N sweeps.

Parameters

- `src` (*int, tuple*) – Source, see info above
- `sweeps` (*int*) – Number of sweeps
- `limit_sweeps` (*bool*) – Limit the number of sweeps?

sinx (*src*)

Sin(x)/x interpolation to produce 10x output samples.

Parameters `src` (*int, tuple*) – Source, see info above

square (*src*)

Square of the input waveform.

Parameters `src` (*int, tuple*) – Source, see info above

square_root (*src*)

Square root of the input waveform.

Parameters `src` (*int, tuple*) – Source, see info above

sum (*src1, src2*)

Product of two sources, `src1`+`src2`.

Parameters

- `src1` (*int, tuple*) – Source 1, see info above
- `src2` (*int, tuple*) – Source 2, see info above

trend (*src, vscale=1, center=0, autoscale=True*)

Trend of the values of a paramter

Parameters

- `vscale` (*float*) – vertical units to display (V)
- `center` (*float*) – center (V)

current_setting

Gets the current setting and returns it as the full command, as sent to the scope when setting an operator.

clear_sweeps ()

Clear the sweeps in a measurement.

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

Executes the given query. Wraps commands sent from property factories in this class with identifiers for the specified channel.

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*

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

operator

Get an operator object to set use to do math.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> channel = inst.channel[0] # set up channel
>>> # set up the first math function
>>> function = inst.math[0]
>>> function.trace = True # turn the trace on
>>> # set function to average the first oscilloscope channel
>>> function.operator.average(0)
```

class Measurement (*parent, idx*)

Class representing a measurement on a MAUI oscilloscope.

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

class State

Enum class for Measurement Parameters. Required to turn it on or off.

both = 'CUST,BOTH'

histogram_icon = 'CUST,HISTICON'

off = 'CUST,OFF'

statistics = 'CUST,STAT'

delete ()

Deletes the given measurement parameter.

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

Executes the given query. Wraps commands sent from property factories in this class with identifiers for the specified channel.

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*

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

set_parameter (*param, src*)

Sets a given parameter that should be measured on this given channel.

Parameters

- **param** (*inst.MeasurementParameters*) – The parameter to set from the given enum list.
- **src** (*int, tuple*) – Source, either as an integer if a channel is requested (e.g., src=0 for Channel 1) or as a tuple in the form, e.g., ('F', 1). Here 'F' refers to a mathematical function and 1 would take the second mathematical function F2.

Raises **AttributeError** – The chosen parameter is invalid.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> msr1 = inst.measurement[0] # set up first measurement
>>> # setup to measure the 10 - 90% rise time on first channel
>>> msr1.set_parameter(inst.MeasurementParameters.rise_time_10_90, 0)
```

measurement_state

Sets / Gets the measurement state. Valid values are 'statistics', 'histogram_icon', 'both', 'off'.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> msr1 = inst.measurement[0] # set up first measurement
>>> msr1.measurement_state = msr1.State.both # set to 'both'
```

statistics

Gets the statistics for the selected parameter. The scope must be in My_Measure mode.

Return tuple (average, low, high, sigma, sweeps)

Return type (float, float, float, float, float)

class MeasurementParameters

Enum containing valid measurement parameters that only require one or more sources. Only single source parameters are currently implemented.

amplitude = 'AMPL'

```
area = 'AREA'  
base = 'BASE'  
delay = 'DLY'  
duty_cycle = 'DUTY'  
fall_time_80_20 = 'FALL82'  
fall_time_90_10 = 'FALL'  
frequency = 'FREQ'  
maximum = 'MAX'  
mean = 'MEAN'  
minimum = 'MIN'  
none = 'NULL'  
overshoot_neg = 'OVSN'  
overshoot_pos = 'OVSP'  
peak_to_peak = 'PKPK'  
period = 'PER'  
phase = 'PHASE'  
rise_time_10_90 = 'RISE'  
rise_time_20_80 = 'RISE28'  
rms = 'RMS'  
stdev = 'SDEV'  
top = 'TOP'  
width_50_neg = 'WIDN'  
width_50_pos = 'WID'
```

class TriggerSource

Enum containing valid trigger sources.

This is an enum for the default values.

Note: This class is initialized like this for four channels, which is the default setting. If you change the number of channels, `TriggerSource` will be recreated using the routine `_create_trigger_source_enum`. This will make further channels available to you or remove channels that are not present in your setup.

```
c0 = 'C1'  
c1 = 'C2'  
c2 = 'C3'  
c3 = 'C4'  
etm10 = 'ETM10'  
ext = 'EX'
```

```
ext10 = 'EX10'
```

```
ext5 = 'EX5'
```

```
line = 'LINE'
```

```
class TriggerState
```

Enum containing valid trigger state for the oscilloscope.

```
auto = 'AUTO'
```

```
normal = 'NORM'
```

```
single = 'SINGLE'
```

```
stop = 'STOP'
```

```
class TriggerType
```

Enum containing valid trigger state.

Availability depends on oscilloscope options. Please consult your manual. Only simple types are currently included.

Warning: Some of the trigger types are untested and might need further parameters in order to be appropriately set.

```
dropout = 'DROPOUT'
```

```
edge = 'EDGE'
```

```
glitch = 'GLIT'
```

```
interval = 'INTV'
```

```
pattern = 'PA'
```

```
qualified = 'TEQ'
```

```
runt = 'RUNT'
```

```
slew_rate = 'SLEW'
```

```
tv = 'TV'
```

```
width = 'WIDTH'
```

```
clear_sweeps()
```

Clears the sweeps in a measurement.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.clear_sweeps()
```

```
force_trigger()
```

Forces a trigger event to occur on the attached oscilloscope.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.force_trigger()
```

run()

Enables the trigger for the oscilloscope and sets it to auto.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.run()
```

stop()

Disables the trigger for the oscilloscope.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.stop()
```

channel

Gets an iterator or list for easy Pythonic access to the various channel objects on the oscilloscope instrument.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> channel = inst.channel[0] # get first channel
```

math

Gets an iterator or list for easy Pythonic access to the various math data sources objects on the oscilloscope instrument.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> math = inst.math[0] # get first math function
```

measurement

Gets an iterator or list for easy Pythonic access to the various measurement data sources objects on the oscilloscope instrument.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> msr = inst.measurement[0] # get first measurement parameter
```

number_channels

Sets/Gets the number of channels available on the specific oscilloscope. Defaults to 4.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.number_channel = 2 # for a oscilloscope with 2 channels
>>> inst.number_channel
2
```

number_functions

Sets/Gets the number of functions available on the specific oscilloscope. Defaults to 2.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.number_functions = 4 # for a oscilloscope with 4 math functions
>>> inst.number_functions
4
```

number_measurements

Sets/Gets the number of measurements available on the specific oscilloscope. Defaults to 6.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.number_measurements = 4 # for a oscilloscope with 4_
↳measurements
>>> inst.number_measurements
4
```

ref

Gets an iterator or list for easy Pythonic access to the various ref data sources objects on the oscilloscope instrument. Typically generated by the `ProxyList` helper.

self_test

Runs an oscilloscope's internal self test and returns the result. The self-test includes testing the hardware of all channels, the timebase and the trigger circuits. Hardware failures are identified by a unique binary code in the returned <status> number. A status of 0 indicates that no failures occurred.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.self_test()
```

show_id

Gets the scope information and returns it. The response comprises manufacturer, oscilloscope model, serial number, and firmware revision level.

Example:


```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.show_id()
```

show_options

Gets and returns oscilloscope options: installed software or hardware that is additional to the standard instrument configuration. The response consists of a series of response fields listing all the installed options.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.show_options()
```

time_div

Sets/Gets the time per division, modifies the timebase setting. Unitful.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.time_div = u.Quantity(200, u.ns)
```

trigger_delay

Sets/Gets the trigger offset with respect to time zero (i.e., a horizontal shift). Unitful.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.trigger_delay = u.Quantity(60, u.ns)
```

trigger_source

Sets / Gets the trigger source.

Note: The *TriggerSource* class is dynamically generated

when the number of channels is switched. The above shown class is only the default! Channels are added and removed, as required.

Warning: If a trigger type is currently set on the oscilloscope that is not implemented in this class, setting the source will fail. The oscilloscope is set up such that the the trigger type and source are set at the same time. However, for convenience, these two properties are split apart here.

Returns Trigger source.

Return type Member of *TriggerSource* class.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.trigger_source = inst.TriggerSource.ext # external trigger
```

trigger_state

Sets / Gets the trigger state. Valid values are defined in *TriggerState* enum class.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.trigger_state = inst.TriggerState.normal
```

trigger_type

Sets / Gets the trigger type.

Warning: If a trigger source is currently set on the oscilloscope that is not implemented in this class, setting the source will fail. The oscilloscope is set up such that the the trigger type and source are set at the same time. However, for convenience, these two properties are split apart here.

Returns Trigger type.

Return type Member of *TriggerType* enum class.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.teledyne.MAUI.open_visa("TCPIP0::192.168.0.10::INSTR")
>>> inst.trigger_type = inst.TriggerType.edge # trigger on edge
```

2.22 ThorLabs

2.22.1 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 = 'CURR'`

`energy = 'ENER'`

`energy_density = 'EDEN'`

`frequency = 'FREQ'`

`power = 'POW'`

`power_density = 'PDEN'`

```

resistance = 'RES'
temperature = 'TEMP'
voltage = 'VOLT'

```

class `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 `SensorFlags`

Enum containing valid sensor flags for the PM100USB

has_temperature_sensor = 256

is_energy_sensor = 2

is_power_sensor = 1

response_settable = 16

tau_settable = 64

wavelength_settable = 32

read (*size=-1, encoding='utf-8'*)

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`

averaging_count

Integer specifying how many samples to collect and average over for each measurement, with each sample taking approximately 3 ms.

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`

measurement_configuration

Returns the current measurement configuration.

Return type `PM100USB.MeasurementConfiguration`

sensor

Returns information about the currently connected sensor.

Type `PM100USB.Sensor`

2.22.2 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`

Raises `TypeError` – If controller is not supported

identify()

Causes a light on the APT instrument to blink, so that it can be identified.

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`

destination

Gets the destination for the APT controller

Type `int`

model_number

Gets the model number for the APT controller

Type `str`

n_channels

Gets/sets the number of channels attached to the APT controller

Type `int`

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`

serial_number

Gets the serial number for the APT controller

Type `str`

class `instruments.thorlabs.APTPiezoInertiaActuator` (*filelike*)

Represent a Thorlabs APT piezo inertia actuator.

Currently only the KIM piezo inertia actuator is implemented. Some routines will work with the TIM actuator as well. Routines that are specific for the KIM101 controller will raise a `TypeError` if not implemented for this controller. Unfortunately, handling all these controller specific functions is fairly messy, but necessary.

Example for a KIM101 controller:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/ttyUSB0",
↳baud=115200)
>>> # set first channel to enabled
>>> ch = kim.channel[0]
>>> ch.enabled_single = True
>>> # define and set drive parameters
>>> max_volts = u.Quantity(110, u.V)
>>> step_rate = u.Quantity(1000, 1/u.s)
>>> acceleration = u.Quantity(10000, 1/u.s**2)
>>> ch.drive_op_parameters = [max_volts, step_rate, acceleration]
>>> # absolute move to 1000 steps
>>> ch.move_abs(1000)
```

class `PiezoChannel` (*apt, idx_chan*)

Class representing a single piezo channel within a piezo stage on the Thorlabs APT controller.

move_abs (*pos*)

Moves the axis to a position specified as the number of steps away from the zero position.

To set the moving parameters, use the setter for `drive_op_parameters`.

Parameters `pos` (*int*) – Position to move to, in steps.

Example:

```
>>> import instruments as ik
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # move to 314 steps
>>> ch.move_abs(314)
```

move_jog (*direction='fwd'*)

Jogs the axis in forward or backward direction by the number of steps that are stored in the controller.

To set the moving parameters, use the setter for `jog_parameters`.

Parameters `direction` (*str*) – Direction of jog. ‘fwd’ for forward, ‘rev’ for backward. ‘fwd’ if invalid argument given

Example:

```
>>> import instruments as ik
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # set jog parameters
>>> params = ch.jog_parameters
>>> params[0] = 2 # move by number of steps
>>> params[1] = 100 # step size forward
>>> params[2] = 200 # step size reverse
>>> ch.jog_parameters = params # set parameters
>>> # jog forward (default)
>>> ch.move_jog()
>>> # jog reverse
>>> ch.move_jog('rev')
```

move_jog_stop()

Stops the current motor movement.

Stop a jog command. The regular motor move stop command does not work for jogging. This command somehow does...

Note: This information is quite empirical. It would only be really needed if jogging parameters are set to continuous. The safer method is to set the step range.

drive_op_parameters

Get / Set various drive parameters for move motion.

Defines the speed and acceleration of moves initiated in the following ways: - by clicking in the position display - via the top panel controls when ‘Go To Position’ mode is selected (in the Set_TIM_JogParameters (09) or Set_KCubeMMIParams (15) sub-messages). - via software using the MoveVelocity, MoveAbsoluteStepsEx or MoveRelativeStepsEx methods.

Setter The setter must be given as a list of 3 entries. The three entries are: - Maximum Voltage: The maximum piezo drive voltage, in the range 85V to 125V. Unitful, if no unit given, V are assumed. - Step Rate: The piezo motor moves by ramping up the drive voltage to the value set in the MaxVoltage parameter and then dropping quickly to zero, then repeating. One cycle is termed a step. This parameter specifies the velocity to move when a command is initiated. The step rate is specified in steps/sec, in the range 1 to 2,000. Unitful, if no unit given, 1 / sec assumed. - Step Acceleration: This parameter specifies the acceleration up to the step rate, in the range 1 to 100,000 cycles/sec/sec. Unitful, if no unit given, 1/sec**2 assumed.

Returns List with the drive parameters, unitful.

Raises

- **TypeError** – The setter was not a list or tuple.
- **ValueError** – The setter was not given a tuple with three values.
- **ValueError** – One of the parameters was out of range.

Example:

```

>>> import instruments as ik
>>> import instruments.units as u
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # change the step rate to 2000 /s
>>> drive_params = ch.drive_op_parameters
>>> drive_params[1] = 2000
>>> ch.drive_op_parameters = drive_params

```

enabled_single

Get / Set single axis enabled.

Note: Enabling multi channels for KIM101 is defined in

the controller class.

Returns Axis status enabled.

Return type bool

Raises `TypeError` – Invalid controller for this command.

Example for a KIM101 controller:

```

>>> import instruments as ik
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # enable channel 0
>>> ch.enabled_single = True

```

jog_parameters

Get / Set the jog parameters.

Define the speed and acceleration of moves initiated in the following ways: - By clicking the jog buttons on the GUI panel - By moving the joystick on the unit when 'Jog Mode' is selected. - via software using the MoveJog method.

It differs from the normal motor jog message in that there are two jog step sizes, one for forward and one for reverse. The reason for this is that due to the inherent nature of the PIA actuators going further in one direction as compared with another this will allow the user to potentially make adjustments to get fore and aft movement the same or similar.

Setter The setter must be given as a list of 5 entries. The three entries are: - Jog Mode (1 for continuous, i.e., until stop command is issued, or 2 jog by the number of steps defined) - Jog Step Size Forward: Range 1 - 2000 - Jog Step Size Backward: Range 1 - 2000 The piezo motor moves by ramping up the drive voltage to the value set in the MaxVoltage parameter and then dropping quickly to zero, then repeating. One cycle is termed a step. This parameter specifies the velocity to move when a command is initiated. The step rate is specified in steps/sec, in the range 1 to 2,000. Unitful, if

no unit given, 1 / sec assumed. - Jog Step Acceleration: This parameter specifies the acceleration up to the step rate, in the range 1 to 100,000 cycles/sec/sec. Unitful, if no unit given, 1/sec**2 assumed.

Returns List with the jog parameters.

Raises

- **TypeError** – The setter was not a list or tuple.
- **ValueError** – The setter was not given a tuple with three values.
- **ValueError** – One of the parameters was out of range.
- **TypeError** – Invalid controller for this command.

Example for a KIM101 controller:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # set jog parameters
>>> mode = 2 # only move by set step size
>>> step = 100 # step size
>>> rate = u.Quantity(1000, 1/u.s) # step rate
>>> # if no quantity given, SI units assumed
>>> accl = 10000
>>> ch.jog_parameters = [mode, step, step, rate, accl]
>>> ch.jog_parameters
[2, 100, 100, array(1000) * 1/s, array(10000) * 1/s**2]
```

position_count

Get/Set the position count of a given channel.

Setter pos Position (steps) of axis.

Returns Position (steps) of axis.

Return type `int`

Example:

```
>>> import instruments as ik
>>> # call the controller
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/
↳ttyUSB0", baud=115200)
>>> # grab channel 0
>>> ch = kim.channel[0]
>>> # set position count to zero
>>> ch.position_count = 0
>>> ch.position_count
0
```

enabled_multi

Enable / Query mulitple channel mode.

For KIM101 controller, where multiple axes can be selected simultaneously (i. e., for a mirror mount).

Setter mode Channel pair to be activated. 0: All channels deactivated 1: First channel pair activated (channel 0 & 1) 2: Second channel pair activated (channel 2 & 3)

Returns The selected mode: 0 - multi-channel selection disabled 1 - Channel 0 & 1 enabled 2 - Channel 2 & 3 enabled

Return type `int`

Raises

- **ValueError** – No valid channel pair selected
- **TypeError** – Invalid controller for this command.

Example:

```
>>> import instruments as ik
>>> kim = ik.thorlabs.APTPiezoInertiaActuator.open_serial("/dev/ttyUSB0",
↳ baud=115200)
>>> # activate the first two channels
>>> kim.enabled_multi = 1
>>> # read back
>>> kim.enabled_multi
1
```

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

Type `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 `APTPiezoInertiaActuator` on the Thorlabs APT controller.

Warning: This is not currently implemented

class `instruments.thorlabs.APTMotorController` (*filelike*)
Class representing a Thorlabs APT motor controller.

Note: A motor model must be selected in order to use unitful distances.

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
```

```
>>> # load the controller, a KDC101 cube
>>> kdc = ik.thorlabs.APTMotorController.open_serial("/dev/ttyUSB0", u
↳baud=115200)
>>> # assign a channel to `ch`
>>> ch = kdc.channel[0]
>>> # select the stage that is connected to the controller
>>> ch.motor_model = 'PRM1-Z8' # a rotation stage
```

```
>>> # home the stage
>>> ch.go_home()
>>> # move to 52 degrees absolute position
>>> ch.move(u.Quantity(52, u.deg))
>>> # move 10 degrees back from current position
>>> ch.move(u.Quantity(-10, u.deg), absolute=False)
```

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

Example:

```
>>> import instruments as ik
>>> import instruments.units as u
```

```
>>> # load the controller, a KDC101 cube
>>> kdc = ik.thorlabs.APTMotorController.open_serial("/dev/ttyUSB0",
↳baud=115200)
>>> # assign a channel to `ch`
>>> ch = kdc.channel[0]
>>> # select the stage that is connected to the controller
>>> ch.motor_model = 'PRM1-Z8' # a rotation stage
```

```
>>> # move to 32 degrees absolute position
>>> ch.move(u.Quantity(32, u.deg))
```

```
>>> # move 10 degrees forward from current position
>>> ch.move(u.Quantity(10, u.deg), absolute=False)
```

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).

motion_timeout

Gets/sets the motor channel motion timeout.

Units Seconds

Type *Quantity*

motor_model

Gets or sets the model name of the attached motor. Note that the scale factors for this motor channel are based on the model of the attached motor and the specifications of the driver of which this is a channel, such that setting a new motor model will update the scale factors accordingly.

Type *str* or *None*

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 = (<*Quantity*(1, 'dimensionless')>, <*Quantity*(1, 'dimensionless')>, <

Sets the scale between the encoder counts and physical units for the position, velocity and acceleration parameters of this channel. By default, set to dimensionless, indicating that the proper scale is not known.

In keeping with the APT protocol documentation, the scale factor is multiplied by the physical quantity to get the encoder count, such that scale factors should have units similar to microsteps/mm, in the example of a linear motor.

Encoder counts are represented by the quantities package unit “ct”, which is considered dimensionally equivalent to dimensionless. Finally, note that the “/s” and “/s**2” are not included in scale factors, so as to produce quantities of dimension “ct/s” and “ct/s**2” from dimensionful input.

For more details, see the APT protocol documentation.

status_bits

Gets the status bits for the specified motor channel.

Type `dict`

2.22.3 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 = 2`

`external = 5`

`manual = 1`

`repeat = 4`

`single = 3`

default ()

Restores instrument to factory settings.

Returns 1 if successful, zero otherwise.

Return type `int`

restore ()

Loads the settings from memory.

Returns 1 if successful, zero otherwise.

Return type `int`

save ()

Stores the parameters in static memory

Returns 1 if successful, zero otherwise.

Return type `int`

save_mode ()

Stores output trigger mode and baud rate settings in memory.

Returns 1 if successful, zero otherwise.

Return type `int`

baud_rate

Gets/sets the instrument baud rate.

Valid baud rates are 9600 and 115200.

Type `int`

closed

Gets the shutter closed status.

`True` represents the shutter is closed, and `False` for the shutter is open.

Return type `bool`

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`

interlock

Gets the interlock tripped status.

Returns `True` if the interlock is tripped, and `False` otherwise.

Return type `bool`

mode

Gets/sets the output mode of the SC10

Return type `SC10.Mode`

name

Gets the name and version number of the device.

Returns Name and version number of the device

Return type `str`

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`

out_trigger

Gets/sets the out trigger source.

0 trigger out follows shutter output, 1 trigger out follows controller output

Type `int`

repeat

Gets/sets the repeat count for repeat mode. Valid range is [1,99] inclusive.

Type `int`

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`

trigger

Gets/sets the trigger source.

0 for internal trigger, 1 for external trigger

Type `int`

2.22.4 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 = 0

voltage1 = 1

voltage2 = 2

default ()

Restores instrument to factory settings.

Returns 1 if successful, 0 otherwise

Return type `int`

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`

save ()

Stores the parameters in static memory

Returns 1 if successful, zero otherwise.

Return type `int`

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`

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`

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`

enable

Gets/sets the output enable status.

If output enable is on (`True`), there is a voltage on the output.

Return type `bool`

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`

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`

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`

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`

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`

mode

Gets/sets the output mode of the LCC25

Return type `LCC25.Mode`

name

Gets the name and version number of the device

Return type `str`

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`

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`

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`

2.22.5 TC200 Temperature Controller

class `instruments.thorlabs.TC200` (*filelike*)

The TC200 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 = 1`

`normal = 0`

class `Sensor`

Enum containing valid temperature sensor types for the TC200.

`ntc10k = 'ntc10k'`

`ptc100 = 'ptc100'`

`ptc1000 = 'ptc1000'`

`th10k = 'th10k'`

name ()

Gets the name and version number of the device

Returns the name string of the device

Return type `str`

beta

Gets/sets the beta value of the thermistor curve.

Value within [2000, 6000]

Returns the gain (in nnn)

Type `int`

d

Gets/sets the d-gain. Valid numbers are [0, 250]

Returns the d-gain (in nnn)

Type `int`

degrees

Gets/sets the units of the temperature measurement.

Returns The temperature units (degC/F/K) the TC200 is measuring in

Type `Unit`

enable

Gets/sets the heater enable status.

If output enable is on (`True`), there is a voltage on the output.

Type `bool`

i

Gets/sets the i-gain. Valid numbers are [1,250]

Returns the i-gain (in nnn)

Return type `int`

max_power

Gets/sets the maximum power

Returns The maximum power

Units Watts (linear units)

Type `Quantity`

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`

mode

Gets/sets the output mode of the TC200

Type `TC200.Mode`

P

Gets/sets the p-gain. Valid numbers are [1,250].

Returns the p-gain (in nnn)

Return type `int`

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`

sensor

Gets/sets the current thermistor type. Used for converting resistances to temperatures.

Returns The thermistor type

Type `TC200.Sensor`

status

Gets the the status code of the TC200

Return type `int`

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`

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`

2.23 Toptica

2.23.1 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 = 3

in_progress = 1

success = 2

un_initialized = 0

class 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`

is_connected

Check whether a laser is connected.

Returns Whether the controller successfully connected to a laser

Type `bool`

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`

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

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.

reboot ()

Reboots the system (note that the serial connect might have to be re-opened after this)

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`

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

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`

enable

is the laser lasing? :return:

firmware

Gets the firmware version of the charm controller

Returns The firmware version of the charm controller

Type `tuple`

fpga_status

Gets the FPGA health status

Returns `False` if there has been a failure for the FPGA, `True` otherwise

Type `bool`

interlock

Gets the interlock switch open state

Returns `True` if interlock switch is open, `False` otherwise

Type `bool`

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 instruments.units as u
>>> tm = ik.toptica.TopMode.open_serial('/dev/ttyUSB0', 115200)
>>> print(tm.laser[0].wavelength)
```

Return type `Laser`

locked

Gets the key switch lock status

Returns `True` if key switch is locked, `False` otherwise

Type `bool`

serial_number

Gets the serial number of the charm controller

Returns The serial number of the charm controller

Type `str`

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`

2.24 Yokogawa

2.24.1 Yokogawa6370 Optical Spectrum Analyzer

class `instruments.yokogawa.Yokogawa6370` (**args, **kwargs*)

The Yokogawa 6370 is a optical spectrum analyzer.

Example usage:

```
>>> import instruments as ik
>>> import instruments.units as u
>>> inst = ik.yokogawa.Yokogawa6370.open_visa('TCPIP0:192.168.0.35')
>>> inst.start_wl = 1030e-9 * u.m
```

class `Channel` (*parent, idx*)

Class representing the channels on the Yokogawa 6370.

This class inherits from `OSChannel`.

Warning: This class should NOT be manually created by the user. It is designed to be initialized by the `Yokogawa6370` class.

data (*bin_format=True*)

Gets the y-axis of the specified data source channel. This is an abstract property.

Parameters `bin_format` (*bool*) – If the waveform should be transferred in binary (`True`) or ASCII (`False`) formats.

Returns The y-component of the waveform.

Return type `numpy.ndarray`

wavelength (*bin_format=True*)

Gets the x-axis of the specified data source channel. This is an abstract property.

Parameters `bin_format` (*bool*) – If the waveform should be transferred in binary (`True`) or ASCII (`False`) formats.

Returns The wavelength component of the waveform.

Return type `numpy.ndarray`

class `SweepModes`

Enum containing valid output modes for the Yokogawa 6370

`AUTO = 3`

`REPEAT = 2`

```

SINGLE = 1
class Traces
    Enum containing valid Traces for the Yokogawa 6370
A = 'TRA'
B = 'TRB'
C = 'TRC'
D = 'TRD'
E = 'TRE'
F = 'TRF'
G = 'TRG'
data()
    Function to query the active Trace data of the OSA.
start_sweep()
    Triggering function for the Yokogawa 6370.
    After changing the sweep mode, the device needs to be triggered before it will update.
wavelength()
    Query the wavelength axis of the active trace.
active_trace
    The active trace of the OSA of enum Yokogawa6370.Traces. Determines the result of Yokogawa6370.data() and Yokogawa6370.wavelength().
bandwidth
    The bandwidth in m.
center_wl
    A floating point property that controls the center wavelength m.
channel
    Gets the specific channel object. This channel is accessed as a list in the following manner:

```

```

>>> import instruments as ik
>>> osa = ik.yokogawa.Yokogawa6370.open_gpibusb('/dev/ttyUSB0')
>>> dat = osa.channel["A"].data # Gets the data of channel 0

```

```

Return type list` [ ~Yokogawa6370.Channel]
points
    An integer property that controls the number of points in a trace.
span
    A floating point property that controls the wavelength span in m.
start_wl
    The start wavelength in m.
start_wl_max
start_wl_min
stop_wl
    The stop wavelength in m.

```

stop_wl_max

stop_wl_min

sweep_mode

A property to control the Sweep Mode as one of `Yokogawa6370.SweepMode`. Effective only after a `self.start_sweep()`.

2.24.2 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 instruments.units as u
>>> inst = ik.yokogawa.Yokogawa7651.open_gpibusb("/dev/ttyUSB0", 1)
>>> inst.voltage = 10 * u.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 Mode

Enum containing valid output modes for the Yokogawa 7651

current = 5

voltage = 1

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.

channel

Gets the specific power supply channel object. Since the Yokogawa7651 is only equipped 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*

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

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

2.25 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.

2.25.1 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')}`.

Each instrument configuration section can also specify one or more attributes to set. These attributes are specified using a `attrs` section as well as the required `class` and `uri` sections. For instance, the following dictionary creates a ThorLabs APT motor controller instrument with a single motor model configured:

```
rot_stage:
  class: !!python/name:instruments.thorabsapt.APTMotorController
  uri: serial:///dev/ttyUSB0?baud=115200
  attrs:
    channel[0].motor_model: PRM1-Z8
```

Unitful attributes can be specified by using the `!Q` tag to quickly create instances of `u.Quantity`. In the example above, for instance, we can set a motion timeout as a unitful quantity:

```
attrs:
  motion_timeout: !Q 1 minute
```

When using the `!Q` tag, any text before a space is taken to be the magnitude of the quantity, and text following is taken to be the unit specification.

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. Alternatively, a file-like object may be provided.
- **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`.

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] # doctest: +SKIP
```

Accessing parts of an instrument should be supported in a way that supports standard Python idioms, most notably iteration.

```
>>> for channel in scope.channel: # doctest: +SKIP
...     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 convenient 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 `pint.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".
```

(continues on next page)

(continued from previous page)

```

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_somewhat()
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:
        # 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:
    # stuff

class SomeInstrument(Instrument):

```

(continues on next page)

(continued from previous page)

```
@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:
        # 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 [pytest](#), a mature and flexible unit-testing framework for Python. When run from the command line via `pytest`, or when run by Travis CI, `pytest` 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 [pytest](#) documentation for full details, as this is not intended to be a guide to `pytest` 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 output 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, u.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 line 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=<Unit('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 `u.Quantity()`:

```
>>> u.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 `u.Quantity`

`instruments.util_fns.convert_temperature(temperature, base)`

Obsolete with the transition to Pint from Quantities.

Parameters

- **temperature** (`pint.Quantity`) – A quantity with units of Kelvin, Celsius, or Fahrenheit
- **base** (`pint.Quantity`) – A temperature unit to convert to

Returns The converted temperature

Return type `pint.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",
        u.volt,
        valid_range=(0*u.volt, 10*u.volt)
        doc="""
        Gets/sets the output voltage.

        :units: As specified, or assumed to be :math:`\text{V}` otherwise.
        :type: `float` or `~pint.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="{}={}",
)
```

(continues on next page)

(continued from previous page)

```

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=u.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 `~pint.Quantity`) or assumed
to be of units degrees C.
:type: `~pint.Quantity` or `int`
:return: the temperature (in degrees C)
:rtype: `~pint.Quantity`
"""
)

```

An alternative to lambda functions is passing in static methods (`staticmethod`).

Bool Property

```

instruments.util_fns.bool_property(command,          set_cmd=None,          inst_true='ON',
                                  inst_false='OFF',  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",
...     inst_true="ON",
...     inst_false="OFF"
... ) # doctest: +SKIP

```

This will result in "BEST:PROPERTY ON" or "BEST:PROPERTY OFF" being sent when setting, and "BEST:PROPERTY?" being sent when getting.

Parameters

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.

- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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` (*command*, *enum*, *set_cmd=None*, *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

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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.

- **get_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when reading/querying from the instrument. If used, the name parameter is still used to set the command for pure-write commands to the instrument.

Unitless Property

```
instruments.util_fns.unitless_property(command, set_cmd=None, format_code='{:e}',  
                                     doc=None, readonly=False, writeonly=False,  
                                     set_fmt='{} {}')
```

Called inside of SCPI classes to instantiate properties with unitless numeric values.

Parameters

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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(command, set_cmd=None, 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

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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 (command, units, set_cmd=None, 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

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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 (command, 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

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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 (command, set_cmd=None, bookmark_symbol='',  
                                     doc=None, readonly=False, writeonly=False,  
                                     set_fmt='{ } {{{}}}')
```

Called inside of SCPI classes to instantiate properties with a string value.

Parameters

- **command** (*str*) – Name of the SCPI command corresponding to this property. If parameter `set_cmd` is not specified, then this parameter is also used for both getting and setting.
- **set_cmd** (*str*) – If not `None`, this parameter sets the command string to be used when sending commands with no return values to the instrument. This allows for non-symmetric properties that have different strings for getting vs setting a 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 “ ”.

3.4.4 Named Structures

The `NamedStruct` class can be used to represent C-style structures for serializing and deserializing data.

class `instruments.named_struct.NamedStruct` (***kwargs*)

Represents a C-style struct with one or more named fields, useful for packing and unpacking serialized data documented in terms of C examples. For instance, consider a struct of the form:

```
typedef struct {
    unsigned long a = 0x1234;
    char[12] dummy;
    unsigned char b = 0xab;
} Foo;
```

This struct can be represented as the following `NamedStruct`:

```
class Foo(NamedStruct):
    a = Field('L')
    dummy = Padding(12)
    b = Field('B')

foo = Foo(a=0x1234, b=0xab)
```

class `instruments.named_struct.Field` (*fmt*, *strip_null=False*)

A named field within a C-style structure.

Parameters **fmt** (*str*) – Format for the field, corresponding to the documentation of the `struct` standard library package.

class `instruments.named_struct.Padding` (*n_bytes=1*)

Represents a field whose value is insignificant, and will not be kept in serialization and deserialization.

Parameters **n_bytes** (*int*) – Number of padding bytes occupied by this field.

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
```

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.

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.
- dijksrwh 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

- _Axis (class in *instruments.newport*), 77
 _SRSDG645Channel (class in *instruments.srs*), 106
 _TekDPO4104Channel (class in *instruments.tektronix*), 110
 _TekDPO4104DataSource (class in *instruments.tektronix*), 109
- ### A
- A (*instruments.srs.SRSDG645.Channels attribute*), 103
 A (*instruments.yokogawa.Yokogawa6370.Traces attribute*), 155
 AB (*instruments.srs.SRSDG645.Outputs attribute*), 104
 abort() (*instruments.agilent.Agilent34410a method*), 24
 abort_motion() (*instruments.newport.NewportESP301Axis method*), 80
 abort_output_trigger() (*instruments.hp.HP6632b method*), 51
 absolute() (*instruments.teledyne.MAUI.Math.Operators method*), 124
 ac (*instruments.rigol.RigolDS1000Series.Channel.Coupling attribute*), 95
 ac (*instruments.srs.SRS830.Coupling attribute*), 97
 ac (*instruments.tektronix.TekDPO4104.Coupling attribute*), 108
 ac (*instruments.tektronix.TekDPO70000.Channel.Coupling attribute*), 111
 ac (*instruments.tektronix.TekTDS224.Coupling attribute*), 118
 ac (*instruments.tektronix.TekTDS5xx.Coupling attribute*), 119
 ac1M (*instruments.teledyne.MAUI.Channel.Coupling attribute*), 122
 acceleration (*instruments.newport.NewportESP301Axis attribute*), 82
 acceleration_feed_forward (*instruments.newport.NewportESP301Axis attribute*), 82
 acknowledge (*instruments.qubitek.CC1 attribute*), 91
 acquire_averages (*instruments.rigol.RigolDS1000Series attribute*), 96
 acquire_enhanced_enob (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_enhanced_state (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_interp_8bit (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_magnivu (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_mode (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_mode_actual (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_num_acquisitions (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_num_avgs (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_num_envelop (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_num_frames (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_num_samples (*instruments.tektronix.TekDPO70000 attribute*), 116
 acquire_sampling_mode (*instruments.tektronix.TekDPO70000 attribute*), 116

- 116
- acquire_state (*instruments.tektronix.TekDPO70000 attribute*), 116
- acquire_stop_after (*instruments.tektronix.TekDPO70000 attribute*), 116
- acquire_type (*instruments.rigol.RigolDS1000Series attribute*), 96
- active_trace (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- actual (*instruments.picowatt.PicowattAVS47.InputSource attribute*), 90
- acv (*instruments.hp.HP3456a.Mode attribute*), 41
- acvdcv (*instruments.hp.HP3456a.Mode attribute*), 41
- address (*instruments.Instrument attribute*), 11
- adv_triggering_enable (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- ag_query() (*instruments.newport.AGUC2 method*), 76
- ag_sendcmd() (*instruments.newport.AGUC2 method*), 76
- Agilent33220a (*class in instruments.agilent*), 22
- Agilent33220a.Function (*class in instruments.agilent*), 22
- Agilent33220a.LoadResistance (*class in instruments.agilent*), 22
- Agilent33220a.OutputPolarity (*class in instruments.agilent*), 23
- Agilent34410a (*class in instruments.agilent*), 24
- AGUC2 (*class in instruments.newport*), 75
- AGUC2.Axes (*class in instruments.newport*), 76
- am_i_still() (*instruments.newport._Axis method*), 77
- am_modulation (*instruments.phasematrix.PhaseMatrixFSW0020 attribute*), 89
- amplitude (*instruments.abstract_instruments.FunctionGenerator attribute*), 113
- amplitude (*instruments.abstract_instruments.FunctionGenerator.Channel attribute*), 115
- amplitude (*instruments.srs.SRS830 attribute*), 100
- amplitude (*instruments.tektronix.TekAWG2000.Channel attribute*), 107
- amplitude (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 128
- amplitude_max (*instruments.srs.SRS830 attribute*), 100
- amplitude_min (*instruments.srs.SRS830 attribute*), 100
- anticipation (*instruments.gentec_eo.Blu attribute*), 30
- APTMotorController (*class in instruments.thorlabs*), 142
- APTMotorController.MotorChannel (*class in instruments.thorlabs*), 142
- APTPiezoInertiaActuator (*class in instruments.thorlabs*), 137
- APTPiezoInertiaActuator.PiezoChannel (*class in instruments.thorlabs*), 137
- APTPiezoStage (*class in instruments.thorlabs*), 141
- APTPiezoStage.PiezoChannel (*class in instruments.thorlabs*), 141
- APTStrainGaugeReader (*class in instruments.thorlabs*), 141
- APTStrainGaugeReader.StrainGaugeChannel (*class in instruments.thorlabs*), 141
- aquisition_continuous (*instruments.tektronix.TekDPO4104 attribute*), 109
- aquisition_length (*instruments.tektronix.TekDPO4104 attribute*), 109
- aquisition_running (*instruments.tektronix.TekDPO4104 attribute*), 109
- arbitrary (*instruments.abstract_instruments.FunctionGenerator.Function attribute*), 13
- arbitrary (*instruments.srs.SRS345.Function attribute*), 96
- area (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 128
- arm_source (*instruments.keithley.Keithley6514 attribute*), 70
- armed (*instruments.tektronix.TekDPO70000.TriggerState attribute*), 115
- ascii (*instruments.tektronix.TekDPO70000.WaveformEncoding attribute*), 115
- assume_units() (*in module instruments.util_fns*), 163
- auto (*instruments.tektronix.TekDPO70000.HorizontalMode attribute*), 113
- auto (*instruments.tektronix.TekDPO70000.TriggerState attribute*), 130
- auto (*instruments.teledyne.MAUI.TriggerState attribute*), 130
- auto (*instruments.thorlabs.SC10.Mode attribute*), 144
- AUTO (*instruments.yokogawa.Yokogawa6370.SweepModes attribute*), 154
- auto_config() (*instruments.keithley.Keithley6514 method*), 70
- auto_offset() (*instruments.srs.SRS830 method*), 98
- auto_phase() (*instruments.srs.SRS830 method*), 98
- auto_range (*instruments.keithley.Keithley6514 attribute*), 70
- auto_range() (*instruments.hp.HP3456a method*), 42
- auto_range() (*instruments.keithley.Keithley195 method*), 58

- auto_range() (*instruments.keithley.Keithley485 method*), 62
 auto_range() (*instruments.keithley.Keithley580 method*), 64
 auto_scale (*instruments.gentec_eo.Blu attribute*), 30
 automatic (*instruments.generic_scp.SCPIMultimeter.InputRange attribute*), 113
 autoscale (*instruments.tektronix.TekDPO70000.Math attribute*), 114
 autozero (*instruments.hp.HP3456a attribute*), 43
 AUX (*instruments.tektronix.TekTDS5xx.Trigger attribute*), 120
 aux1 (*instruments.srs.SRS830.Mode attribute*), 97
 aux2 (*instruments.srs.SRS830.Mode attribute*), 98
 aux3 (*instruments.srs.SRS830.Mode attribute*), 98
 aux4 (*instruments.srs.SRS830.Mode attribute*), 98
 available_scales (*instruments.gentec_eo.Blu attribute*), 31
 average (*instruments.rigol.RigolDS1000Series.AcquisitionType attribute*), 94
 average (*instruments.srs.SRSCTC100.Channel attribute*), 101
 average (*instruments.tektronix.TekDPO70000.AcquisitionMode attribute*), 110
 average() (*instruments.teledyne.MAUI.Math.Operators method*), 124
 averaging_count (*instruments.thorlabs.PM100USB attribute*), 135
 axis (*instruments.newport.AGUC2 attribute*), 76
 axis (*instruments.newport.NewportError attribute*), 86
 axis (*instruments.newport.NewportESP301 attribute*), 80
 axis_id (*instruments.newport.NewportESP301Axis attribute*), 82
 axis_status (*instruments.newport._Axis attribute*), 77
- ## B
- B (*instruments.srs.SRSDG645.Channels attribute*), 103
 B (*instruments.yokogawa.Yokogawa6370.Traces attribute*), 155
 bandwidth (*instruments.tektronix.TekDPO70000.Channel attribute*), 112
 bandwidth (*instruments.yokogawa.Yokogawa6370 attribute*), 155
 base (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
 battery_state (*instruments.gentec_eo.Blu attribute*), 31
 baud_rate (*instruments.thorlabs.SC10 attribute*), 144
 beta (*instruments.thorlabs.TC200 attribute*), 148
 big_endian (*instruments.tektronix.TekDPO70000.ByteOrder attribute*), 111
 binary (*instruments.tektronix.TekDPO70000.WaveformEncoding attribute*), 115
 binblockread() (*instruments.Instrument method*), 7
 blackman_harris (*instruments.tektronix.TekDPO70000.Math.SpectralWindow attribute*), 89
 blanking (*instruments.phasematrix.PhaseMatrixFSW0020 attribute*), 89
 block_data_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 15
 block_data_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
 block_data_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 15
 block_data_not_allowed (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
 Blu (*class in instruments.gentec_eo*), 28
 Blu.Scale (*class in instruments.gentec_eo*), 28
 block_property() (*in module instruments.util_fns*), 166
 both (*instruments.lakeshore.Lakeshore475.PeakDisplay attribute*), 73
 both (*instruments.teledyne.MAUI.Measurement.State attribute*), 127
 bounded_unitful_property() (*in module instruments.util_fns*), 170
 buffer_mode (*instruments.srs.SRS830 attribute*), 100
 burst_count (*instruments.srs.SRSDG645 attribute*), 105
 burst_count (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
 burst_delay (*instruments.srs.SRSDG645 attribute*), 105
 burst_delay (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
 burst_mode (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
 burst_period (*instruments.srs.SRSDG645 attribute*), 105
 burst_period (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
 burst_T0_config (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
 bus (*instruments.generic_scp.SCPIMultimeter.TriggerMode attribute*), 19
 bus (*instruments.keithley.Keithley2182.TriggerMode attribute*), 115

- tribute), 67
- bus (*instruments.keithley.Keithley6514.ArmSource attribute*), 69
- busy (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- busy_read_done (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- busy_read_done_ovf (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- busy_read_ovf (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- bw_limit (*instruments.rigol.RigolDS1000Series.Channel attribute*), 95
- ## C
- C (*instruments.glassman.GlassmanFR.ResponseCode attribute*), 36
- C (*instruments.srs.SRSDG645.Channels attribute*), 103
- C (*instruments.yokogawa.Yokogawa6370.Traces attribute*), 155
- c0 (*instruments.teledyne.MAUI.TriggerSource attribute*), 129
- c1 (*instruments.teledyne.MAUI.TriggerSource attribute*), 129
- c2 (*instruments.teledyne.MAUI.TriggerSource attribute*), 129
- c3 (*instruments.teledyne.MAUI.TriggerSource attribute*), 129
- cache_units (*instruments.thorlabs.PM100USB attribute*), 135
- cal_not_enabled (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
- cal_password_incorrect (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
- cal_switch_prevents_cal (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
- calibrate (*instruments.ondax.LM.Status attribute*), 87
- calibration_message (*instruments.thorlabs.PM100USB.Sensor attribute*), 135
- capacitance (*instruments.fluke.Fluke3000.Mode attribute*), 26
- capacitance (*instruments.generic_scp.SCPIMultimeter.Mode attribute*), 18
- CC1 (*class in instruments.qubitekk*), 91
- CC1.Channel (*class in instruments.qubitekk*), 91
- CD (*instruments.srs.SRSDG645.Outputs attribute*), 105
- center () (*instruments.qubitekk.MC1 method*), 92
- center_wl (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- centered (*instruments.tektronix.TekDPO70000.Math.FilterMode attribute*), 113
- ch1 (*instruments.srs.SRS830.Mode attribute*), 98
- CH1 (*instruments.tektronix.TekTDS5xx.Source attribute*), 119
- CH1 (*instruments.tektronix.TekTDS5xx.Trigger attribute*), 120
- ch2 (*instruments.srs.SRS830.Mode attribute*), 98
- CH2 (*instruments.tektronix.TekTDS5xx.Source attribute*), 119
- CH2 (*instruments.tektronix.TekTDS5xx.Trigger attribute*), 120
- CH3 (*instruments.tektronix.TekTDS5xx.Source attribute*), 119
- CH3 (*instruments.tektronix.TekTDS5xx.Trigger attribute*), 120
- CH4 (*instruments.tektronix.TekTDS5xx.Source attribute*), 119
- CH4 (*instruments.tektronix.TekTDS5xx.Trigger attribute*), 120
- change_measurement_mode () (*instruments.lakeshore.Lakeshore475 method*), 73
- change_position_control_mode () (*instruments.thorlabs.APTPiezoStage.PiezoChannel method*), 141
- channel (*instruments.abstract_instruments.FunctionGenerator attribute*), 14
- channel (*instruments.abstract_instruments.signal_generator.SignalGenerator attribute*), 14
- channel (*instruments.abstract_instruments.signal_generator.SingleChannel attribute*), 14
- channel (*instruments.glassman.GlassmanFR attribute*), 37
- channel (*instruments.holzworth.HS9000 attribute*), 40
- channel (*instruments.hp.HP6624a attribute*), 47
- channel (*instruments.hp.HP6652a attribute*), 54
- channel (*instruments.hp.HPe3631a attribute*), 56
- channel (*instruments.keithley.Keithley2182 attribute*), 68
- channel (*instruments.keithley.Keithley6220 attribute*), 69
- channel (*instruments.lakeshore.Lakeshore370 attribute*), 72
- channel (*instruments.minghe.MHS5200 attribute*), 75
- channel (*instruments.qubitekk.CCI attribute*), 91
- channel (*instruments.rigol.RigolDS1000Series attribute*), 96
- channel (*instruments.srs.SRSCTC100 attribute*), 103
- channel (*instruments.srs.SRSDG645 attribute*), 105

- channel (*instruments.tektronix.TekAWG2000 attribute*), 108
- channel (*instruments.tektronix.TekDPO4104 attribute*), 109
- channel (*instruments.tektronix.TekDPO70000 attribute*), 116
- channel (*instruments.tektronix.TekTDS224 attribute*), 118
- channel (*instruments.tektronix.TekTDS5xx attribute*), 120
- channel (*instruments.teledyne.MAUI attribute*), 131
- channel (*instruments.thorlabs.ThorLabsAPT attribute*), 136
- channel (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- channel (*instruments.yokogawa.Yokogawa7651 attribute*), 157
- channel_count (*instruments.hp.HP6624a attribute*), 47
- channel_delay (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- channel_levels (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- channel_polarity (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- channel_units() (*instruments.srs.SRSCTC100 method*), 102
- channelid (*instruments.hp.HPe3631a attribute*), 56
- character_data_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 15
- character_data_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
- character_data_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 15
- character_data_not_allowed (*instruments.hp.HP6632b.ErrorCodes attribute*), 48
- character_data_too_long (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 15
- character_data_too_long (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- charge (*instruments.keithley.Keithley6514.Mode attribute*), 70
- charge (*instruments.keithley.Keithley6514.ValidRange attribute*), 70
- charm_status (*instruments.toptica.TopMode.Laser attribute*), 150
- check_error_queue() (*instruments.generic_scp.SCPIInstrument method*), 17
- check_error_queue() (*instruments.hp.HP6632b method*), 51
- checksum_error (*instruments.glassman.GlassmanFR.ErrorCode attribute*), 36
- clear() (*instruments.generic_scp.SCPIInstrument method*), 17
- clear() (*instruments.hp.HP6624a method*), 47
- clear_counts() (*instruments.qubitek.CC1 method*), 91
- clear_data_buffer() (*instruments.srs.SRS830 method*), 98
- clear_log() (*instruments.srs.SRSCTC100 method*), 102
- clear_memory() (*instruments.agilent.Agilent34410a method*), 24
- clear_sweeps() (*instruments.teledyne.MAUI method*), 130
- clear_sweeps() (*instruments.teledyne.MAUI.Math method*), 126
- clock (*instruments.tektronix.TekTDS5xx attribute*), 120
- closed (*instruments.thorlabs.SC10 attribute*), 144
- command_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
- command_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- command_header_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
- command_header_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- command_only_applic_rs232 (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- computed_prog_cal_constants_incorrect (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- computed_readback_cal_const_incorrect (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- confirm_connection() (*instruments.gentec_eo.Blu method*), 29
- connect() (*instruments.fluke.Fluke3000 method*), 26
- constant (*instruments.tektronix.TekDPO70000.HorizontalMode attribute*), 113
- continuity (*instruments.generic_scp.SCPIMultimeter.Mode attribute*), 18

continuous_onget (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
 continuous_ontalk (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
 continuous_onx (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
 control_mode (*instruments.lakeshore.Lakeshore475 attribute*), 73
 control_slope_limit (*instruments.lakeshore.Lakeshore475 attribute*), 73
 controller (*instruments.qubitekk.MC1 attribute*), 93
 convert_temperature() (*in module instruments.util_fns*), 164
 correction() (*instruments.toptica.TopMode.Laser method*), 150
 correction_status (*instruments.toptica.TopMode.Laser attribute*), 150
 count (*instruments.hp.HP3456a attribute*), 43
 count (*instruments.hp.HP3456a.Register attribute*), 41
 count (*instruments.qubitekk.CC1.Channel attribute*), 91
 coupling (*instruments.rigol.RigolDS1000Series.Channel attribute*), 95
 coupling (*instruments.srs.SRS830 attribute*), 100
 coupling (*instruments.tektronix._TekDPO4104Channel attribute*), 110
 coupling (*instruments.tektronix.TekDPO70000.Channel attribute*), 112
 coupling (*instruments.teledyne.MAUI.Channel attribute*), 123
 curr_or_volt_fetch_incompat_with_last_acq (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
 current (*instruments.glassman.GlassmanFR attribute*), 37
 current (*instruments.glassman.GlassmanFR.Mode attribute*), 36
 current (*instruments.hp.HP6624a attribute*), 47
 current (*instruments.hp.HP6624a.Channel attribute*), 46
 current (*instruments.hp.HP6624a.Mode attribute*), 47
 current (*instruments.hp.HP6652a attribute*), 54
 current (*instruments.hp.HPe3631a attribute*), 56
 current (*instruments.keithley.Keithley6220 attribute*), 69
 current (*instruments.keithley.Keithley6514.Mode attribute*), 70
 current (*instruments.keithley.Keithley6514.ValidRange attribute*), 70
 current (*instruments.newport.NewportESP301Axis attribute*), 82
 current (*instruments.ondax.LM attribute*), 87
 current (*instruments.thorlabs.PM100USB.MeasurementConfiguration attribute*), 134
 current (*instruments.yokogawa.Yokogawa7651 attribute*), 157
 current (*instruments.yokogawa.Yokogawa7651.Channel attribute*), 156
 current (*instruments.yokogawa.Yokogawa7651.Mode attribute*), 157
 current_ac (*instruments.fluke.Fluke3000.Mode attribute*), 26
 current_ac (*instruments.generic_scp.SCPIMultimeter.Mode attribute*), 18
 current_ac (*instruments.keithley.Keithley195.Mode attribute*), 58
 current_ac (*instruments.keithley.Keithley195.ValidRange attribute*), 58
 current_control_status (*instruments.toptica.TopMode.Laser attribute*), 150
 current_dc (*instruments.fluke.Fluke3000.Mode attribute*), 26
 current_dc (*instruments.generic_scp.SCPIMultimeter.Mode attribute*), 18
 current_dc (*instruments.keithley.Keithley195.Mode attribute*), 58
 current_dc (*instruments.keithley.Keithley195.ValidRange attribute*), 58
 current_max (*instruments.hp.HPe3631a attribute*), 56
 current_max (*instruments.keithley.Keithley6220 attribute*), 69
 current_min (*instruments.hp.HPe3631a attribute*), 56
 current_min (*instruments.keithley.Keithley6220 attribute*), 69
 current_sense (*instruments.glassman.GlassmanFR attribute*), 37
 current_sense (*instruments.hp.HP6624a attribute*), 47
 current_sense (*instruments.hp.HP6624a.Channel attribute*), 46
 current_sense (*instruments.hp.HP6652a attribute*), 54
 current_sense (*instruments.hp.HPe3631a attribute*), 56
 current_sense_range (*instruments.hp.HP6632b attribute*), 51
 current_setting (*instruments.teledyne.MAUI.Math.Operators attribute*), 126
 current_status (*instruments.toptica.TopMode attribute*), 153
 current_trigger (*instruments.hp.HP6632b attribute*), 52

- current_value (*instruments.gentec_eo.Blu attribute*), 31
- cv_or_cc_status_incorrect (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- cycle (*instruments.thorlabs.TC200.Mode attribute*), 148
- ## D
- D (*instruments.srs.SRSDG645.Channels attribute*), 103
- d (*instruments.thorlabs.TC200 attribute*), 148
- D (*instruments.yokogawa.Yokogawa6370.Traces attribute*), 155
- data (*instruments.hp.HP6632b.DigitalFunction attribute*), 48
- data () (*instruments.yokogawa.Yokogawa6370 method*), 155
- data () (*instruments.yokogawa.Yokogawa6370.Channel method*), 154
- data_frame_start (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_frame_stop (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_out_of_range (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- data_point_count (*instruments.agilent.Agilent34410a attribute*), 25
- data_snap () (*instruments.srs.SRS830 method*), 98
- data_source (*instruments.tektronix.TekDPO4104 attribute*), 109
- data_source (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_source (*instruments.tektronix.TekTDS224 attribute*), 118
- data_source (*instruments.tektronix.TekTDS5xx attribute*), 120
- data_start (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_stop (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_sync_sources (*instruments.tektronix.TekDPO70000 attribute*), 116
- data_transfer (*instruments.srs.SRS830 attribute*), 100
- data_type_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
- data_type_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
- data_width (*instruments.tektronix.TekDPO4104 attribute*), 109
- data_width (*instruments.tektronix.TekTDS224 attribute*), 118
- data_width (*instruments.tektronix.TekTDS5xx attribute*), 120
- db (*instruments.hp.HP3456a.MathMode attribute*), 40
- db (*instruments.tektronix.TekDPO70000.Math.Mag attribute*), 113
- dBm (*instruments.abstract_instruments.FunctionGenerator.VoltageMode attribute*), 13
- dbm (*instruments.hp.HP3456a.MathMode attribute*), 40
- dbm (*instruments.tektronix.TekDPO70000.Math.Mag attribute*), 113
- dc (*instruments.agilent.Agilent33220a.Function attribute*), 22
- dc (*instruments.keithley.Keithley580.Drive attribute*), 63
- dc (*instruments.lakeshore.Lakeshore475.Mode attribute*), 73
- dc (*instruments.rigol.RigolDS1000Series.Channel.Coupling attribute*), 95
- dc (*instruments.srs.SRS830.Coupling attribute*), 97
- dc (*instruments.tektronix.TekDPO4104.Coupling attribute*), 108
- dc (*instruments.tektronix.TekDPO70000.Channel.Coupling attribute*), 111
- dc (*instruments.tektronix.TekTDS224.Coupling attribute*), 118
- dc (*instruments.tektronix.TekTDS5xx.Coupling attribute*), 119
- dc1M (*instruments.teledyne.MAUI.Channel.Coupling attribute*), 122
- dc50 (*instruments.teledyne.MAUI.Channel.Coupling attribute*), 122
- dc_reject (*instruments.tektronix.TekDPO70000.Channel.Coupling attribute*), 111
- dcbv (*instruments.hp.HP3456a.Mode attribute*), 41
- deceleration (*instruments.newport.NewportESP301Axis attribute*), 82
- default (*instruments.generic_scp.SCPIMultimeter.InputRange attribute*), 18
- default (*instruments.generic_scp.SCPIMultimeter.Resolution attribute*), 18
- default (*instruments.generic_scp.SCPIMultimeter.SampleCount attribute*), 19
- default (*instruments.generic_scp.SCPIMultimeter.TriggerCount attribute*), 19
- default () (*instruments.thorlabs.LCC25 method*), 146
- default () (*instruments.thorlabs.SC10 method*), 144
- define (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- define_program () (*instruments.newport.NewportESP301 method*),

- 79
- degrees (*instruments.tektronix.TekDPO70000.Math.Phase* attribute), 113
- degrees (*instruments.thorlabs.TC200* attribute), 148
- delay (*instruments.hp.HP3456a* attribute), 43
- delay (*instruments.hp.HP3456a.Register* attribute), 41
- delay (*instruments.qubitekk.CC1* attribute), 92
- delay (*instruments.srs._SRSDG645Channel* attribute), 106
- delay (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- delete() (*instruments.teledyne.MAUI.Measurement* method), 127
- derivative() (*instruments.teledyne.MAUI.Math.Operators* method), 124
- derivative_gain (*instruments.newport.NewportESP301Axis* attribute), 82
- desired_position (*instruments.newport.NewportESP301Axis* attribute), 82
- desired_velocity (*instruments.newport.NewportESP301Axis* attribute), 83
- deskew (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
- destination (*instruments.thorlabs.ThorLabsAPT* attribute), 136
- device_timeout (*instruments.glassman.GlassmanFR* attribute), 37
- difference() (*instruments.teledyne.MAUI.Math.Operators* method), 125
- digital_data (*instruments.hp.HP6632b* attribute), 52
- digital_function (*instruments.hp.HP6632b* attribute), 52
- digital_io_selftest (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
- diode (*instruments.generic_scp.SCPIMultimeter.Mode* attribute), 18
- diode (*instruments.srs.SRSCTC100.SensorType* attribute), 102
- direction (*instruments.qubitekk.MC1* attribute), 93
- disable() (*instruments.keithley.Keithley6220* method), 69
- disable() (*instruments.newport.NewportESP301Axis* method), 80
- disconnect() (*instruments.gentec_eo.Blu* method), 29
- display (*instruments.picowatt.PicowattAVS47* attribute), 90
- display (*instruments.rigol.RigolDS1000Series.Channel* attribute), 95
- display (*instruments.srs.SRSDG645* attribute), 105
- display() (*instruments.toptica.TopMode* method), 152
- display_brightness (*instruments.generic_scp.SCPIInstrument* attribute), 17
- display_brightness (*instruments.hp.HP6632b* attribute), 52
- display_clock (*instruments.tektronix.TekTDS5xx* attribute), 120
- display_contrast (*instruments.generic_scp.SCPIInstrument* attribute), 17
- display_contrast (*instruments.hp.HP6632b* attribute), 52
- display_figures (*instruments.srs.SRSCTC100* attribute), 103
- display_text() (*instruments.hp.HP6652a* method), 54
- display_textmode (*instruments.hp.HP6652a* attribute), 54
- dpo (*instruments.tektronix.TekDPO70000.TriggerState* attribute), 115
- drive (*instruments.keithley.Keithley580* attribute), 65
- drive_op_parameters (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* attribute), 138
- dropout (*instruments.teledyne.MAUI.TriggerType* attribute), 130
- dry_circuit_test (*instruments.keithley.Keithley580* attribute), 65
- duty_cycle (*instruments.agilent.Agilent33220a* attribute), 23
- duty_cycle (*instruments.minghe.MHS5200.Channel* attribute), 74
- duty_cycle (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- dwelling (*instruments.thorlabs.LCC25* attribute), 146
- dwelling_time (*instruments.qubitekk.CC1* attribute), 92
- ## E
- E (*instruments.srs.SRSDG645.Channels* attribute), 103
- E (*instruments.yokogawa.Yokogawa6370.Traces* attribute), 155
- edge (*instruments.teledyne.MAUI.TriggerType* attribute), 130
- EF (*instruments.srs.SRSDG645.Outputs* attribute), 105
- enable (*instruments.minghe.MHS5200.Channel* attribute), 74
- enable (*instruments.thorlabs.LCC25* attribute), 146
- enable (*instruments.thorlabs.SC10* attribute), 144

- enable (*instruments.thorlabs.TC200* attribute), 148
- enable (*instruments.toptica.TopMode* attribute), 153
- enable (*instruments.toptica.TopMode.Laser* attribute), 151
- enable() (*instruments.newport.NewportESP301Axis* method), 80
- enable_adv_triggering (*instruments.srs.SRSDG645* attribute), 105
- enable_burst_mode (*instruments.srs.SRSDG645* attribute), 106
- enable_burst_t0_first (*instruments.srs.SRSDG645* attribute), 106
- enable_remote_mode (*instruments.newport.AGUC2* attribute), 77
- enabled (*instruments.ondax.LM* attribute), 87
- enabled (*instruments.thorlabs.ThorLabsAPT.APTChannel* attribute), 136
- enabled_multi (*instruments.thorlabs.APTPiezoInertiaActuator* attribute), 140
- enabled_single (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* attribute), 139
- encoder_position (*instruments.newport.NewportESP301Axis* attribute), 83
- encoder_resolution (*instruments.newport.NewportESP301Axis* attribute), 83
- energy (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 134
- energy_density (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 134
- enum_property() (*in module instruments.util_fns*), 167
- envelope (*instruments.tektronix.TekDPO70000.AcquisitionMode* attribute), 110
- envelope() (*instruments.teledyne.MAUI.Math.Operators* method), 125
- eoi_mode (*instruments.keithley.Keithley485* attribute), 62
- equivalent_time_allowed (*instruments.tektronix.TekDPO70000.SamplingMode* attribute), 115
- eres() (*instruments.teledyne.MAUI.Math.Operators* method), 125
- errcheck() (*instruments.srs.SRSCTC100* method), 102
- errcode (*instruments.newport.NewportError* attribute), 86
- error_check_toggle (*instruments.srs.SRSCTC100* attribute), 103
- error_previous_command (*instruments.newport.AGUC2* attribute), 77
- error_threshold (*instruments.newport.NewportESP301Axis* attribute), 83
- estop_deceleration (*instruments.newport.NewportESP301Axis* attribute), 83
- etm10 (*instruments.teledyne.MAUI.TriggerSource* attribute), 129
- event_status_bit (*instruments.hp.HP6632b.DFISource* attribute), 48
- excitation (*instruments.picowatt.PicowattAVS47* attribute), 90
- execute() (*instruments.toptica.TopMode* method), 152
- execute_bulk_command() (*instruments.newport.NewportESP301* method), 79
- execution_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
- exponent_too_large (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- exponent_too_large (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
- expression_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- expression_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
- expression_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- expression_not_allowed (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
- ext (*instruments.teledyne.MAUI.TriggerSource* attribute), 129
- ext10 (*instruments.teledyne.MAUI.TriggerSource* attribute), 129
- ext5 (*instruments.teledyne.MAUI.TriggerSource* attribute), 130
- ext_continuous (*instruments.keithley.Keithley195.TriggerMode* attribute), 58
- ext_one_shot (*instruments.keithley.Keithley195.TriggerMode* attribute), 58
- extern (*instruments.thorlabs.LCC25* attribute), 146
- external (*instruments.generic_scp.SCPIMultimeter.TriggerMode*

- attribute), 19
- external (*instruments.hp.HP3456a.TriggerMode attribute*), 42
- external (*instruments.keithley.Keithley2182.TriggerMode attribute*), 67
- external (*instruments.srs.SRS830.FreqSource attribute*), 97
- external (*instruments.thorlabs.SC10.Mode attribute*), 144
- external_falling (*instruments.srs.SRSDG645.TriggerSource attribute*), 105
- external_rising (*instruments.srs.SRSDG645.TriggerSource attribute*), 105
- extra_bytes (*instruments.glassman.GlassmanFR.ErrorCode attribute*), 36
- ## F
- F (*instruments.srs.SRSDG645.Channels attribute*), 103
- F (*instruments.yokogawa.Yokogawa6370.Traces attribute*), 155
- failure (*instruments.toptica.TopMode.CharmStatus attribute*), 150
- fall_time_80_20 (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- fall_time_90_10 (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- Falling (*instruments.tektronix.TekTDS5xx.Edge attribute*), 119
- fast (*instruments.hp.HP6632b.ALCBandwidth attribute*), 48
- fault (*instruments.glassman.GlassmanFR attribute*), 37
- feedback_configuration (*instruments.newport.NewportESP301Axis attribute*), 83
- fetch() (*instruments.agilent.Agilent34410a method*), 24
- fetch() (*instruments.hp.HP3456a method*), 42
- fetch() (*instruments.keithley.Keithley2182 method*), 67
- fetch() (*instruments.keithley.Keithley6514 method*), 70
- fft() (*instruments.teledyne.MAUI.Math.Operators method*), 125
- Field (*class in instruments.named_struct*), 171
- field (*instruments.lakeshore.Lakeshore475 attribute*), 73
- field_control_params (*instruments.lakeshore.Lakeshore475 attribute*), 73
- field_setpoint (*instruments.lakeshore.Lakeshore475 attribute*), 74
- field_units (*instruments.lakeshore.Lakeshore475 attribute*), 74
- Fifty (*instruments.tektronix.TekTDS5xx.Impedance attribute*), 119
- filter (*instruments.hp.HP3456a attribute*), 43
- filter (*instruments.rigol.RigolDS1000Series.Channel attribute*), 95
- filter_mode (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- filter_risetime (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- firmware (*instruments.ondax.LM attribute*), 87
- firmware (*instruments.qubitek.CC1 attribute*), 92
- firmware (*instruments.qubitek.MC1 attribute*), 93
- firmware (*instruments.toptica.TopMode attribute*), 153
- firmware_version (*instruments.newport.AGUC2 attribute*), 77
- first_mode_hop_time (*instruments.toptica.TopMode.Laser attribute*), 151
- flags (*instruments.thorlabs.PM100USB.Sensor attribute*), 135
- flattop2 (*instruments.tektronix.TekDPO70000.Math.SpectralWindow attribute*), 113
- float (*instruments.tektronix.TekDPO70000.BinaryFormat attribute*), 111
- floor() (*instruments.teledyne.MAUI.Math.Operators method*), 125
- Fluke3000 (*class in instruments.fluke*), 25
- Fluke3000.Mode (*class in instruments.fluke*), 26
- Fluke3000.Module (*class in instruments.fluke*), 26
- flush() (*instruments.fluke.Fluke3000 method*), 26
- force_trigger() (*instruments.rigol.RigolDS1000Series method*), 95
- force_trigger() (*instruments.tektronix.TekDPO4104 method*), 108
- force_trigger() (*instruments.tektronix.TekDPO70000 method*), 115
- force_trigger() (*instruments.tektronix.TekTDS224 method*), 118
- force_trigger() (*instruments.tektronix.TekTDS5xx method*), 120
- force_trigger() (*instruments.teledyne.MAUI method*), 130
- fourpt_resistance (*instruments.generic_scpi.SCPIMultimeter.Mode attribute*), 130

- attribute), 18
- fpga_status (instruments.toptica.TopMode attribute), 153
- frequency (instruments.abstract_instruments.FunctionGenerator attribute), 14
- frequency (instruments.abstract_instruments.FunctionGenerator.Channel attribute), 13
- frequency (instruments.abstract_instruments.signal_generator.SignalGenerator attribute), 15
- frequency (instruments.fluke.Fluke3000.Mode attribute), 26
- frequency (instruments.generic_scp.SCPIFunctionGenerator attribute), 21
- frequency (instruments.generic_scp.SCPIMultimeter.Mode attribute), 18
- frequency (instruments.holzworth.HS9000.Channel attribute), 39
- frequency (instruments.minghe.MHS5200.Channel attribute), 74
- frequency (instruments.phasematrix.PhaseMatrixFSW0020 attribute), 89
- frequency (instruments.srs.SRS345 attribute), 97
- frequency (instruments.srs.SRS830 attribute), 100
- frequency (instruments.tektronix.TekAWG2000.Channel attribute), 107
- frequency (instruments.teledyne.MAUI.MeasurementParameters attribute), 129
- frequency (instruments.thorlabs.LCC25 attribute), 147
- frequency (instruments.thorlabs.PM100USB.MeasurementConfiguration attribute), 134
- frequency_max (instruments.holzworth.HS9000.Channel attribute), 39
- frequency_min (instruments.holzworth.HS9000.Channel attribute), 39
- frequency_source (instruments.srs.SRS830 attribute), 100
- front_panel_uart_buffer_overrun (instruments.hp.HP6632b.ErrorCodes attribute), 49
- front_panel_uart_framing (instruments.hp.HP6632b.ErrorCodes attribute), 49
- front_panel_uart_overrun (instruments.hp.HP6632b.ErrorCodes attribute), 49
- front_panel_uart_parity (instruments.hp.HP6632b.ErrorCodes attribute), 49
- front_panel_uart_timeout (instruments.hp.HP6632b.ErrorCodes attribute), 49
- FULL (instruments.tektronix.TekTDS5xx.Bandwidth attribute), 119
- full_step_resolution (instruments.newport.NewportESP301Axis attribute), 83
- FunctionGenerator (instruments.abstract_instruments.FunctionGenerator attribute), 14
- FunctionGenerator.Channel (instruments.abstract_instruments.FunctionGenerator.Channel attribute), 13
- function (instruments.agilent.Agilent33220a attribute), 23
- function (instruments.generic_scp.SCPIFunctionGenerator attribute), 22
- function (instruments.minghe.MHS5200.Channel attribute), 75
- function (instruments.srs.SRS345 attribute), 97
- FunctionGenerator (class in instruments.abstract_instruments), 12
- FunctionGenerator.Channel (class in instruments.abstract_instruments), 12
- FunctionGenerator.Function (class in instruments.abstract_instruments), 13
- FunctionGenerator.VoltageMode (class in instruments.abstract_instruments), 13
- ## G
- G (instruments.srs.SRSDG645.Channels attribute), 103
- G (instruments.yokogawa.Yokogawa6370.Traces attribute), 155
- gate (instruments.qubitek.CC1 attribute), 92
- gaussian (instruments.tektronix.TekDPO70000.Math.SpectralWindow attribute), 113
- get_continuous (instruments.keithley.Keithley195.TriggerMode attribute), 58
- get_continuous (instruments.keithley.Keithley580.TriggerMode attribute), 64
- get_hardcopy() (instruments.tektronix.TekTDS5xx method), 120
- get_log() (instruments.srs.SRSCTC100.Channel method), 101
- get_log_point() (instruments.srs.SRSCTC100.Channel method), 101
- get_message() (instruments.newport.NewportError static method), 86
- get_not_allowed (instruments.generic_scp.SCPIInstrument.ErrorCodes attribute), 16
- get_not_allowed (instruments.hp.HP6632b.ErrorCodes attribute), 49

get_one_shot (*instruments.keithley.Keithley195.TriggerMode attribute*), 58
 get_one_shot (*instruments.keithley.Keithley580.TriggerMode attribute*), 64
 get_settings() (*instruments.thorlabs.LCC25 method*), 146
 get_status() (*instruments.glassman.GlassmanFR method*), 36
 get_status() (*instruments.keithley.Keithley485 method*), 62
 get_status() (*instruments.newport.NewportESP301Axis method*), 80
 get_status_word() (*instruments.keithley.Keithley195 method*), 58
 get_status_word() (*instruments.keithley.Keithley580 method*), 64
 GH (*instruments.srs.SRSDG645.Outputs attribute*), 105
 GlassmanFR (*class in instruments.glassman*), 35
 GlassmanFR.ErrorCode (*class in instruments.glassman*), 36
 GlassmanFR.Mode (*class in instruments.glassman*), 36
 GlassmanFR.ResponseCode (*class in instruments.glassman*), 36
 glitch (*instruments.teledyne.MAUI.TriggerType attribute*), 130
 go_home() (*instruments.thorlabs.APTMotorController.MotorChannel method*), 142
 ground (*instruments.picowatt.PicowattAVS47.InputSourceHold attribute*), 90
 ground (*instruments.rigol.RigolDS1000Series.Channel.Coupling attribute*), 95
 ground (*instruments.tektronix.TekDPO4104.Coupling attribute*), 108
 ground (*instruments.tektronix.TekDPO70000.Channel.Coupling attribute*), 111
 ground (*instruments.tektronix.TekTDS224.Coupling attribute*), 118
 ground (*instruments.tektronix.TekTDS5xx.Coupling attribute*), 119
 ground (*instruments.teledyne.MAUI.Channel.Coupling attribute*), 122
 group_delay (*instruments.tektronix.TekDPO70000.Math.Phase attribute*), 113

H

H (*instruments.srs.SRSDG645.Channels attribute*), 104
 hamming (*instruments.tektronix.TekDPO70000.Math.SpectralWindow attribute*), 113
 hanning (*instruments.hp.HP6632b.SenseWindow attribute*), 51
 hanning (*instruments.tektronix.TekDPO70000.Math.SpectralWindow attribute*), 113
 hardware_limit_configuration (*instruments.newport.NewportESP301Axis attribute*), 83
 has_temperature_sensor (*instruments.thorlabs.PM100USB.SensorFlags attribute*), 135
 head_type (*instruments.gentec_eo.Blu attribute*), 31
 header_separator_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 header_separator_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
 header_suffix_out_of_range (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 header_suffix_out_of_range (*instruments.hp.HP6632b.ErrorCodes attribute*), 49
 hi_res (*instruments.tektronix.TekDPO70000.AcquisitionMode attribute*), 111
 high_impedance (*instruments.agilent.Agilent33220a.LoadResistance attribute*), 22
 histogram_icon (*instruments.teledyne.MAUI.Measurement.State attribute*), 127
 hold (*instruments.hp.HP3456a.TriggerMode attribute*), 42
 home (*instruments.newport.NewportESP301Axis attribute*), 83
 home_index_signals (*instruments.newport.NewportESP301HomeSearchMode attribute*), 86
 home_signal_only (*instruments.newport.NewportESP301HomeSearchMode attribute*), 86
 homing_velocity (*instruments.newport.NewportESP301Axis attribute*), 83
 HOR_DIVS (*instruments.tektronix.TekDPO70000 attribute*), 115
 horiz_acq_duration (*instruments.tektronix.TekDPO70000 attribute*), 116
 horiz_acq_length (*instruments.tektronix.TekDPO70000 attribute*), 117
 horiz_delay_mode (*instruments.tektronix.TekDPO70000 attribute*), 117

- ments.tektronix.TekDPO70000* attribute), 117
 hori_z_delay_pos (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_delay_time (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_interp_ratio (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_main_pos (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_mode (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_pos (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_record_length (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_record_length_lim (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_roll (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_sample_rate (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_scale (*instruments.tektronix.TekDPO70000* attribute), 117
 hori_z_unit (*instruments.tektronix.TekDPO70000* attribute), 117
 horizontal_scale (*instruments.tektronix.TekTDS5xx* attribute), 121
 HP3456a (*class in instruments.hp*), 40
 HP3456a.MathMode (*class in instruments.hp*), 40
 HP3456a.Mode (*class in instruments.hp*), 41
 HP3456a.Register (*class in instruments.hp*), 41
 HP3456a.TriggerMode (*class in instruments.hp*), 42
 HP3456a.ValidRange (*class in instruments.hp*), 42
 HP6624a (*class in instruments.hp*), 45
 HP6624a.Channel (*class in instruments.hp*), 45
 HP6624a.Mode (*class in instruments.hp*), 46
 HP6632b (*class in instruments.hp*), 47
 HP6632b.ALCBandwidth (*class in instruments.hp*), 48
 HP6632b.DFISource (*class in instruments.hp*), 48
 HP6632b.DigitalFunction (*class in instruments.hp*), 48
 HP6632b.ErrorCodes (*class in instruments.hp*), 48
 HP6632b.RemoteInhibit (*class in instruments.hp*), 51
 HP6632b.SenseWindow (*class in instruments.hp*), 51
 HP6652a (*class in instruments.hp*), 53
 HPe3631a (*class in instruments.hp*), 55
 HPe3631a.Channel (*class in instruments.hp*), 56
 HS9000 (*class in instruments.holzworth*), 38
 HS9000.Channel (*class in instruments.holzworth*), 38
I
 i (*instruments.thorlabs.TC200* attribute), 148
 i_value (*instruments.lakeshore.Lakeshore475* attribute), 74
 idcc (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
 idcco (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
 idcco_idcc (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
 identify () (*instruments.thorlabs.ThorLabsAPT* method), 136
 idx (*instruments.srs._SRSDG645Channel* attribute), 106
 illegal_control (*instruments.glassman.GlassmanFR.ErrorCode* attribute), 36
 illegal_macro_label (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
 illegal_parameter_value (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
 illegal_while_fault (*instruments.glassman.GlassmanFR.ErrorCode* attribute), 36
 immediate (*instruments.generic_scp.SCPIMultimeter.SampleSource* attribute), 19
 immediate (*instruments.generic_scp.SCPIMultimeter.TriggerMode* attribute), 19
 immediate (*instruments.keithley.Keithley2182.TriggerMode* attribute), 67
 immediate (*instruments.keithley.Keithley6514.ArmSource* attribute), 69
 immediate (*instruments.keithley.Keithley6514.TriggerMode* attribute), 70
 in_progress (*instruments.toptica.TopMode.CharmStatus* attribute), 150
 incorrect_seq_cal_commands (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
 increment (*instruments.qubitek.MC1* attribute), 93
 increment (*instruments.thorlabs.LCC25* attribute), 147
 inertia (*instruments.qubitek.MC1* attribute), 93
 infinity (*instruments.generic_scp.SCPIMultimeter.TriggerCount* attribute), 19

`ingrd_recv_buffer_overrun` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`init()` (*instruments.agilent.Agilent34410a* method), 24
`init()` (*instruments.srs.SRS830* method), 98
`init_output_continuous` (*instruments.hp.HP6632b* attribute), 52
`init_output_trigger()` (*instruments.hp.HP6632b* method), 51
`inner_modulation` (*instruments.ondax.LM.Status* attribute), 87
`input_range` (*instruments.abstract_instruments.Multimeter* attribute), 12
`input_range` (*instruments.fluke.Fluke3000* attribute), 27
`input_range` (*instruments.generic_scp.SCPIMultimeter* attribute), 19
`input_range` (*instruments.hp.HP3456a* attribute), 43
`input_range` (*instruments.keithley.Keithley195* attribute), 59
`input_range` (*instruments.keithley.Keithley2182* attribute), 68
`input_range` (*instruments.keithley.Keithley2182.Channel* attribute), 67
`input_range` (*instruments.keithley.Keithley485* attribute), 62
`input_range` (*instruments.keithley.Keithley580* attribute), 65
`input_range` (*instruments.keithley.Keithley6514* attribute), 71
`input_shield_ground` (*instruments.srs.SRS830* attribute), 100
`input_source` (*instruments.picowatt.PicowattAVS47* attribute), 90
`Instrument` (class in *instruments*), 7
`int` (*instruments.tektronix.TekDPO70000.BinaryFormat* attribute), 111
`int_property()` (in module *instruments.util_fns*), 168
`integral()` (*instruments.teledyne.MAUI.Math.Operators* method), 125
`integral_gain` (*instruments.newport.NewportESP301Axis* attribute), 84
`integral_saturation_gain` (*instruments.newport.NewportESP301Axis* attribute), 84
`intensity` (*instruments.toptica.TopMode.Laser* attribute), 151
`interlock` (*instruments.thorlabs.SC10* attribute), 145
`interlock` (*instruments.toptica.TopMode* attribute), 153
`internal` (*instruments.hp.HP3456a.TriggerMode* attribute), 42
`internal` (*instruments.srs.SRS830.FreqSource* attribute), 97
`internal` (*instruments.srs.SRSDG645.TriggerSource* attribute), 105
`internal_position` (*instruments.qubitek.MC1* attribute), 93
`interpolation_allowed` (*instruments.tektronix.TekDPO70000.SamplingMode* attribute), 115
`interval` (*instruments.teledyne.MAUI.TriggerType* attribute), 130
`invalid_block_data` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_block_data` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_character` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_character` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_character_data` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_character_data` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_character_in_number` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_character_in_number` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_expression` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_expression` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_inside_macro_definition` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_inside_macro_definition` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
`invalid_outside_macro_definition` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
`invalid_outside_macro_definition` (*instruments.hp.HP6632b.ErrorCodes* attribute), 49

- ments.hp.HP6632b.ErrorCodes* attribute), 49
 invalid_separator (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
 invalid_separator (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
 invalid_string_data (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
 invalid_string_data (*instruments.hp.HP6632b.ErrorCodes* attribute), 49
 invalid_suffix (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
 invalid_suffix (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
 invert (*instruments.rigol.RigolDS1000Series.Channel* attribute), 95
 invert () (*instruments.teledyne.MAUI.Math.Operators* method), 125
 inverted (*instruments.agilent.Agilent33220a.OutputPolarity* attribute), 23
 inverted (*instruments.tektronix.TekAWG2000.Polarity* attribute), 107
 is_centering () (*instruments.qubitekk.MC1* method), 92
 is_connected (*instruments.toptica.TopMode.Laser* attribute), 151
 is_energy_sensor (*instruments.thorlabs.PM100USB.SensorFlags* attribute), 135
 is_motion_done (*instruments.newport.NewportESP301Axis* attribute), 84
 is_power_sensor (*instruments.thorlabs.PM100USB.SensorFlags* attribute), 135
- ## J
- jerk (*instruments.newport.NewportESP301Axis* attribute), 84
 jog (*instruments.newport._Axis* attribute), 78
 jog_high_velocity (*instruments.newport.NewportESP301Axis* attribute), 84
 jog_low_velocity (*instruments.newport.NewportESP301Axis* attribute), 84
 jog_parameters (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* attribute), 139
- ## K
- kaiser_besse (*instruments.tektronix.TekDPO70000.Math.SpectralWindow* attribute), 113
 Keithley195 (*class in instruments.keithley*), 57
 Keithley195.Mode (*class in instruments.keithley*), 58
 Keithley195.TriggerMode (*class in instruments.keithley*), 58
 Keithley195.ValidRange (*class in instruments.keithley*), 58
 Keithley2182 (*class in instruments.keithley*), 66
 Keithley2182.Channel (*class in instruments.keithley*), 66
 Keithley2182.Mode (*class in instruments.keithley*), 67
 Keithley2182.TriggerMode (*class in instruments.keithley*), 67
 Keithley485 (*class in instruments.keithley*), 60
 Keithley485.SRQDataMask (*class in instruments.keithley*), 60
 Keithley485.SRQErrorMask (*class in instruments.keithley*), 61
 Keithley485.Status (*class in instruments.keithley*), 61
 Keithley485.TriggerMode (*class in instruments.keithley*), 62
 Keithley580 (*class in instruments.keithley*), 63
 Keithley580.Drive (*class in instruments.keithley*), 63
 Keithley580.Polarity (*class in instruments.keithley*), 64
 Keithley580.TriggerMode (*class in instruments.keithley*), 64
 Keithley6220 (*class in instruments.keithley*), 68
 Keithley6514 (*class in instruments.keithley*), 69
 Keithley6514.ArmSource (*class in instruments.keithley*), 69
 Keithley6514.Mode (*class in instruments.keithley*), 70
 Keithley6514.TriggerMode (*class in instruments.keithley*), 70
 Keithley6514.ValidRange (*class in instruments.keithley*), 70
- ## L
- label (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
 label (*instruments.tektronix.TekDPO70000.Math* attribute), 114
 label_xpos (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
 label_xpos (*instruments.tektronix.TekDPO70000.Math* attribute), 114

- label_ypos (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
- label_ypos (*instruments.tektronix.TekDPO70000.Math* attribute), 114
- Lakeshore340 (*class in instruments.lakeshore*), 71
- Lakeshore340.Sensor (*class in instruments.lakeshore*), 71
- Lakeshore370 (*class in instruments.lakeshore*), 72
- Lakeshore370.Channel (*class in instruments.lakeshore*), 72
- Lakeshore475 (*class in instruments.lakeshore*), 72
- Lakeshore475.Filter (*class in instruments.lakeshore*), 72
- Lakeshore475.Mode (*class in instruments.lakeshore*), 73
- Lakeshore475.PeakDisplay (*class in instruments.lakeshore*), 73
- Lakeshore475.PeakMode (*class in instruments.lakeshore*), 73
- laser (*instruments.toptica.TopMode* attribute), 153
- latching (*instruments.hp.HP6632b.RemoteInhibit* attribute), 51
- latest_mode_hop_time (*instruments.toptica.TopMode.Laser* attribute), 151
- LCC25 (*class in instruments.thorlabs*), 145
- LCC25.Mode (*class in instruments.thorlabs*), 145
- left_limit (*instruments.newport.NewportESP301Axis* attribute), 84
- level_amplitude (*instruments.srs.SRSDG645.Output* attribute), 104
- level_offset (*instruments.srs.SRSDG645.Output* attribute), 104
- limit_status (*instruments.newport.AGUC2* attribute), 77
- line (*instruments.srs.SRSDG645.TriggerSource* attribute), 105
- LINE (*instruments.tektronix.TekTDS5xx.Trigger* attribute), 120
- line (*instruments.teledyne.MAUI.TriggerSource* attribute), 130
- line_frequency (*instruments.generic_scp.SCPIInstrument* attribute), 17
- line_frequency (*instruments.hp.HP6632b* attribute), 52
- linear (*instruments.tektronix.TekDPO70000.Math.Mag* attribute), 113
- little_endian (*instruments.tektronix.TekDPO70000.ByteOrder* attribute), 111
- live (*instruments.hp.HP6632b.RemoteInhibit* attribute), 51
- LM (*class in instruments.ondax*), 87
- LM.Status (*class in instruments.ondax*), 87
- load_instruments () (*in module instruments*), 157
- load_resistance (*instruments.agilent.Agilent33220a* attribute), 23
- lock_start (*instruments.toptica.TopMode.Laser* attribute), 151
- locked (*instruments.toptica.TopMode* attribute), 153
- log (*instruments.keithley.Keithley485* attribute), 62
- loop (*instruments.srs.SRS830.BufferMode* attribute), 97
- lower (*instruments.hp.HP3456a* attribute), 43
- lower (*instruments.hp.HP3456a.Register* attribute), 41
- lower_limit (*instruments.qubitek.MC1* attribute), 93
- lowpass (*instruments.lakeshore.Lakeshore475.Filter* attribute), 72
- ## M
- m3000 (*instruments.fluke.Fluke3000.Module* attribute), 26
- macro_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- macro_error_180 (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- macro_error_270 (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- macro_execution_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- macro_parameter_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- macro_parameter_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- macro_recursion_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- macro_redefinition_not_allowed (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- manual (*instruments.keithley.Keithley2182.TriggerMode* attribute), 67
- manual (*instruments.keithley.Keithley6514.ArmSource* attribute), 69
- manual (*instruments.tektronix.TekDPO70000.HorizontalMode* attribute), 113
- manual (*instruments.thorlabs.SC10.Mode* attribute), 144
- math (*instruments.rigol.RigolDS1000Series* attribute), 96
- math (*instruments.tektronix.TekDPO4104* attribute), 109

- math (*instruments.tektronix.TekDPO70000 attribute*), 117
- math (*instruments.tektronix.TekTDS224 attribute*), 118
- math (*instruments.tektronix.TekTDS5xx attribute*), 121
- math (*instruments.teledyne.MAUI attribute*), 131
- Math1 (*instruments.tektronix.TekTDS5xx.Source attribute*), 119
- Math2 (*instruments.tektronix.TekTDS5xx.Source attribute*), 120
- Math3 (*instruments.tektronix.TekTDS5xx.Source attribute*), 120
- math_mode (*instruments.hp.HP3456a attribute*), 43
- MAUI (*class in instruments.teledyne*), 121
- MAUI.Channel (*class in instruments.teledyne*), 122
- MAUI.Channel.Coupling (*class in instruments.teledyne*), 122
- MAUI.DataSource (*class in instruments.teledyne*), 123
- MAUI.Math (*class in instruments.teledyne*), 124
- MAUI.Math.Operators (*class in instruments.teledyne*), 124
- MAUI.Measurement (*class in instruments.teledyne*), 127
- MAUI.Measurement.State (*class in instruments.teledyne*), 127
- MAUI.MeasurementParameters (*class in instruments.teledyne*), 128
- MAUI.TriggerSource (*class in instruments.teledyne*), 129
- MAUI.TriggerState (*class in instruments.teledyne*), 130
- MAUI.TriggerType (*class in instruments.teledyne*), 130
- max1 (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10 (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100 (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100kilo (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100Mega (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100micro (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100milli (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100nano (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max100pico (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10kilo (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10Mega (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10micro (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10milli (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10nano (*instruments.gentec_eo.Blu.Scale attribute*), 28
- max10pico (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1kilo (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1Mega (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1micro (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1milli (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1nano (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max1pico (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3 (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30 (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300 (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300kilo (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300Mega (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300micro (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300milli (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300nano (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max300pico (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30kilo (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30Mega (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30micro (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30milli (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30nano (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max30pico (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3kilo (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3Mega (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3micro (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3milli (*instruments.gentec_eo.Blu.Scale attribute*), 29
- max3nano (*instruments.gentec_eo.Blu.Scale attribute*), 29

- 29
- max3pico (*instruments.gentec_eo.Blu.Scale* attribute), 29
- max_acceleration (*instruments.newport.NewportESP301Axis* attribute), 84
- max_base_velocity (*instruments.newport.NewportESP301Axis* attribute), 84
- max_deceleration (*instruments.newport.NewportESP301Axis* attribute), 84
- max_power (*instruments.thorlabs.TC200* attribute), 149
- max_temperature (*instruments.thorlabs.TC200* attribute), 149
- max_velocity (*instruments.newport.NewportESP301Axis* attribute), 85
- max_voltage (*instruments.thorlabs.LCC25* attribute), 147
- maximum (*instruments.agilent.Agilent33220a.LoadResistance* attribute), 23
- maximum (*instruments.generic_scp.SCPIMultimeter.InputRange* attribute), 18
- maximum (*instruments.generic_scp.SCPIMultimeter.Resolution* attribute), 18
- maximum (*instruments.generic_scp.SCPIMultimeter.SampleCount* attribute), 19
- maximum (*instruments.generic_scp.SCPIMultimeter.TriggerCount* attribute), 19
- maximum (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- maximum_current (*instruments.ondax.LM* attribute), 87
- MC1 (*class in instruments.qubitekk*), 92
- MC1.MotorType (*class in instruments.qubitekk*), 92
- mean (*instruments.hp.HP3456a* attribute), 44
- mean (*instruments.hp.HP3456a.Register* attribute), 41
- mean (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- measure () (*instruments.abstract_instruments.Multimeter* method), 12
- measure () (*instruments.fluke.Fluke3000* method), 26
- measure () (*instruments.generic_scp.SCPIMultimeter* method), 19
- measure () (*instruments.hp.HP3456a* method), 42
- measure () (*instruments.keithley.Keithley195* method), 58
- measure () (*instruments.keithley.Keithley2182* method), 68
- measure () (*instruments.keithley.Keithley2182.Channel* method), 67
- measure () (*instruments.keithley.Keithley485* method), 62
- measure () (*instruments.keithley.Keithley580* method), 64
- measure_mode (*instruments.gentec_eo.Blu* attribute), 32
- measurement (*instruments.tektronix.TekTDS5xx* attribute), 121
- measurement (*instruments.teledyne.MAUI* attribute), 131
- measurement_configuration (*instruments.thorlabs.PM100USB* attribute), 136
- measurement_overrange (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- measurement_state (*instruments.teledyne.MAUI.Measurement* attribute), 128
- messageDict (*instruments.newport.NewportError* attribute), 86
- metric_position (*instruments.qubitekk.MC1* attribute), 93
- MHS5200 (*class in instruments.minghe*), 74
- MHS5200.Channel (*class in instruments.minghe*), 74
- MHS5200.Function (*class in instruments.minghe*), 75
- micro_inch (*instruments.newport.NewportESP301Axis* attribute), 85
- microstep_factor (*instruments.newport.NewportESP301Axis* attribute), 85
- minimum (*instruments.agilent.Agilent33220a.LoadResistance* attribute), 23
- minimum (*instruments.generic_scp.SCPIMultimeter.InputRange* attribute), 18
- minimum (*instruments.generic_scp.SCPIMultimeter.Resolution* attribute), 18
- minimum (*instruments.generic_scp.SCPIMultimeter.SampleCount* attribute), 19
- minimum (*instruments.generic_scp.SCPIMultimeter.TriggerCount* attribute), 19
- minimum (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- missing_parameter (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- missing_parameter (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- mode (*instruments.abstract_instruments.Multimeter* attribute), 12
- mode (*instruments.fluke.Fluke3000* attribute), 27
- mode (*instruments.generic_scp.SCPIMultimeter* attribute), 20

- mode (*instruments.glassman.GlassmanFR* attribute), 37
- mode (*instruments.hp.HP3456a* attribute), 44
- mode (*instruments.hp.HP6624a.Channel* attribute), 46
- mode (*instruments.hp.HP6652a* attribute), 55
- mode (*instruments.hp.HPe3631a* attribute), 56
- mode (*instruments.keithley.Keithley195* attribute), 59
- mode (*instruments.keithley.Keithley2182.Channel* attribute), 67
- mode (*instruments.keithley.Keithley6514* attribute), 71
- mode (*instruments.thorlabs.LCC25* attribute), 147
- mode (*instruments.thorlabs.SC10* attribute), 145
- mode (*instruments.thorlabs.TC200* attribute), 149
- mode (*instruments.yokogawa.Yokogawa7651.Channel* attribute), 156
- mode_hop (*instruments.toptica.TopMode.Laser* attribute), 151
- model (*instruments.toptica.TopMode.Laser* attribute), 151
- model_number (*instruments.thorlabs.ThorLabsAPT* attribute), 136
- motion_timeout (*instruments.thorlabs.APTMotorController.MotorChannel* attribute), 143
- motor_model (*instruments.thorlabs.APTMotorController.MotorChannel* attribute), 143
- motor_type (*instruments.newport.NewportESP301Axis* attribute), 85
- move () (*instruments.newport.NewportESP301Axis* method), 80
- move () (*instruments.qubitekk.MC1* method), 93
- move () (*instruments.thorlabs.APTMotorController.MotorChannel* method), 142
- move_abs () (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* method), 137
- move_indefinitely () (*instruments.newport.NewportESP301Axis* method), 80
- move_jog () (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* method), 137
- move_jog_stop () (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* method), 138
- move_relative (*instruments.newport._Axis* attribute), 78
- move_timeout (*instruments.qubitekk.MC1* attribute), 94
- move_to_hardware_limit () (*instruments.newport.NewportESP301Axis* method), 80
- move_to_limit (*instruments.newport._Axis* attribute), 78
- Multimeter (class in *instruments.abstract_instruments*), 12
- mux_channel (*instruments.picowatt.PicowattAVS47* attribute), 90
- ## N
- n_channels (*instruments.thorlabs.ThorLabsAPT* attribute), 136
- name (*instruments.generic_scp.SCPIInstrument* attribute), 17
- name (*instruments.holzworth.HS9000* attribute), 40
- name (*instruments.hp.HP6652a* attribute), 55
- name (*instruments.rigol.RigolDS1000Series.DataSource* attribute), 95
- name (*instruments.srs.SRSCTC100.Channel* attribute), 101
- name (*instruments.tektronix._TekDPO4104DataSource* attribute), 110
- name (*instruments.tektronix.TekAWG2000.Channel* attribute), 107
- name (*instruments.tektronix.TekDPO70000.DataSource* attribute), 112
- name (*instruments.teledyne.MAUI.DataSource* attribute), 124
- name (*instruments.thorlabs.LCC25* attribute), 147
- name (*instruments.thorlabs.PM100USB.Sensor* attribute), 135
- name (*instruments.thorlabs.SC10* attribute), 145
- name (*instruments.thorlabs.ThorLabsAPT* attribute), 136
- name () (*instruments.thorlabs.TC200* method), 148
- NamedStruct (class in *instruments.named_struct*), 171
- narrow (*instruments.lakeshore.Lakeshore475.Filter* attribute), 73
- neg_index_signals (*instruments.newport.NewportESP301HomeSearchMode* attribute), 86
- neg_limit_signal (*instruments.newport.NewportESP301HomeSearchMode* attribute), 86
- neg_polarity (*instruments.keithley.Keithley580.Polarity* attribute), 64
- negative (*instruments.lakeshore.Lakeshore475.PeakDisplay* attribute), 73
- negative (*instruments.srs.SRSDG645.LevelPolarity* attribute), 104
- new_value_ready (*instruments.gentec_eo.Blu* attribute), 32
- NewportError (class in *instruments.newport*), 86
- NewportESP301 (class in *instruments.newport*), 79
- NewportESP301Axis (class in *instruments.newport*), 80
- NewportESP301HomeSearchMode (class in *instruments.newport*), 86
- no_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16

- no_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- noise (*instruments.abstract_instruments.FunctionGenerator.FunctionGenerator* attribute), 13
- noise (*instruments.agilent.Agilent33220a.Function* attribute), 22
- noise (*instruments.srs.SRS345.Function* attribute), 96
- none (*instruments.srs.SRS830.Mode* attribute), 98
- none (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- normal (*instruments.agilent.Agilent33220a.OutputPolarity* attribute), 23
- normal (*instruments.hp.HP6632b.ALCBandwidth* attribute), 48
- normal (*instruments.keithley.Keithley485.Status* attribute), 61
- normal (*instruments.ondax.LM.Status* attribute), 87
- normal (*instruments.rigol.RigolDS1000Series.AcquisitionType* attribute), 94
- normal (*instruments.tektronix.TekAWG2000.Polarity* attribute), 107
- normal (*instruments.teledyne.MAUI.TriggerState* attribute), 130
- normal (*instruments.thorlabs.LCC25.Mode* attribute), 146
- normal (*instruments.thorlabs.TC200.Mode* attribute), 148
- not_remote (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
- not_remote_idcc (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
- not_remote_idcco (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
- not_remote_idcco_idcc (*instruments.keithley.Keithley485.SRQErrorMask* attribute), 61
- nplc (*instruments.hp.HP3456a* attribute), 44
- nplc (*instruments.hp.HP3456a.Register* attribute), 41
- nplc (*instruments.hp.HP3456a.ValidRange* attribute), 42
- nstest (*instruments.keithley.Keithley6514.ArmSource* attribute), 69
- ntc10k (*instruments.thorlabs.TC200.Sensor* attribute), 148
- null (*instruments.hp.HP3456a.MathMode* attribute), 40
- num_avg (*instruments.tektronix.TekDPO70000.Math* attribute), 114
- num_data_points (*instruments.srs.SRS830* attribute), 101
- number_channels (*instruments.teledyne.MAUI* attribute), 131
- number_functions (*instruments.teledyne.MAUI* attribute), 132
- number_measurements (*instruments.teledyne.MAUI* attribute), 132
- number_of_digits (*instruments.hp.HP3456a* attribute), 44
- number_of_digits (*instruments.hp.HP3456a.Register* attribute), 41
- number_of_readings (*instruments.hp.HP3456a* attribute), 44
- number_of_readings (*instruments.hp.HP3456a.Register* attribute), 41
- number_of_steps (*instruments.newport._Axis* attribute), 78
- numeric_data_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- numeric_data_error (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- numeric_data_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- numeric_data_not_allowed (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- oc_resistance_2wire (*instruments.hp.HP3456a.Mode* attribute), 41
- oc_resistance_4wire (*instruments.hp.HP3456a.Mode* attribute), 41
- off (*instruments.hp.HP3456a.MathMode* attribute), 40
- off (*instruments.hp.HP6632b.DFISource* attribute), 48
- off (*instruments.hp.HP6632b.RemoteInhibit* attribute), 51
- off (*instruments.tektronix.TekDPO70000.AcquisitionState* attribute), 111
- off (*instruments.teledyne.MAUI.Measurement.State* attribute), 127
- offset (*instruments.abstract_instruments.FunctionGenerator* attribute), 14
- offset (*instruments.abstract_instruments.FunctionGenerator.Channel* attribute), 13
- offset (*instruments.generic_scp.SCPIFunctionGenerator* attribute), 22
- offset (*instruments.minghe.MHS5200.Channel* attribute), 75
- offset (*instruments.srs.SRS345* attribute), 97
- offset (*instruments.tektronix.TekAWG2000.Channel* attribute), 107
- offset (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
- offset (*instruments.teledyne.MAUI.Channel* attribute), 123

- on (*instruments.tektronix.TekDPO70000.AcquisitionState attribute*), 111
- on_time (*instruments.topica.TopMode.Laser attribute*), 151
- one_shot (*instruments.srs.SRS830.BufferMode attribute*), 97
- OneHundred (*instruments.tektronix.TekTDS5xx.Bandwidth attribute*), 119
- OneMeg (*instruments.tektronix.TekTDS5xx.Impedance attribute*), 119
- oneshot_onget (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
- oneshot_ontalk (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
- oneshot_onx (*instruments.keithley.Keithley485.TriggerMode attribute*), 62
- op_complete (*instruments.generic_scp.SCPIInstrument attribute*), 17
- open_file() (*instruments.Instrument class method*), 7
- open_from_uri() (*instruments.Instrument class method*), 8
- open_gpibethernet() (*instruments.Instrument class method*), 8
- open_gpibusb() (*instruments.Instrument class method*), 8
- open_serial() (*instruments.Instrument class method*), 9
- open_tcpip() (*instruments.Instrument class method*), 9
- open_test() (*instruments.Instrument class method*), 9
- open_time (*instruments.thorlabs.SC10 attribute*), 145
- open_usb() (*instruments.Instrument class method*), 10
- open_usbtmc() (*instruments.Instrument class method*), 10
- open_visa() (*instruments.Instrument class method*), 10
- open_vx11() (*instruments.Instrument class method*), 10
- operate (*instruments.keithley.Keithley580 attribute*), 65
- operation (*instruments.hp.HP6632b.DFISource attribute*), 48
- operation_complete (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
- operation_complete (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- operator (*instruments.teledyne.MAUI.Math attribute*), 127
- out_of_memory (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- out_trigger (*instruments.thorlabs.SC10 attribute*), 145
- outgoing_binary_format (*instruments.tektronix.TekDPO70000 attribute*), 117
- outgoing_byte_order (*instruments.tektronix.TekDPO70000 attribute*), 117
- outgoing_n_bytes (*instruments.tektronix.TekDPO70000 attribute*), 117
- outgoing_waveform_encoding (*instruments.tektronix.TekDPO70000 attribute*), 117
- output (*instruments.abstract_instruments.signal_generator.SGChannel attribute*), 15
- output (*instruments.agilent.Agilent33220a attribute*), 23
- output (*instruments.glassman.GlassmanFR attribute*), 38
- output (*instruments.holzworth.HS9000.Channel attribute*), 39
- output (*instruments.hp.HP6624a.Channel attribute*), 46
- output (*instruments.hp.HP6652a attribute*), 55
- output (*instruments.hp.HPe3631a attribute*), 57
- output (*instruments.phasematrix.PhaseMatrixFSW0020 attribute*), 89
- output (*instruments.srs.SRSDG645 attribute*), 106
- output (*instruments.yokogawa.Yokogawa7651.Channel attribute*), 156
- output_dfi (*instruments.hp.HP6632b attribute*), 52
- output_dfi_source (*instruments.hp.HP6632b attribute*), 52
- output_mode_must_be_normal (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- output_polarity (*instruments.agilent.Agilent33220a attribute*), 23
- output_position (*instruments.thorlabs.ATPiezoStage.PiezoChannel attribute*), 141
- output_protection_delay (*instruments.hp.HP6632b attribute*), 52
- output_remote_inhibit (*instruments.hp.HP6632b attribute*), 53
- output_sync (*instruments.agilent.Agilent33220a attribute*), 23
- ovdac_selftest (*instruments*), 50

- ments.hp.HP6632b.ErrorCodes* attribute), 50
 overcurrent (*instruments.hp.HP6624a.Channel* attribute), 46
 overcurrent (*instruments.hp.HP6652a* attribute), 55
 overflow (*instruments.keithley.Keithley485.Status* attribute), 61
 overshoot_neg (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
 overshoot_pos (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
 overvoltage (*instruments.hp.HP6624a.Channel* attribute), 46
 overvoltage (*instruments.hp.HP6652a* attribute), 55
 OxfordITC503 (class in *instruments.oxford*), 88
 OxfordITC503.Sensor (class in *instruments.oxford*), 88
- ## P
- p (*instruments.thorlabs.TC200* attribute), 149
 p_value (*instruments.lakeshore.Lakeshore475* attribute), 74
 Padding (class in *instruments.named_struct*), 171
 panel_locked (*instruments.rigol.RigolDS1000Series* attribute), 96
 parameter_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
 parameter_not_allowed (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
 parse_measurement() (*instruments.keithley.Keithley580* static method), 64
 parse_status_word() (*instruments.keithley.Keithley195* static method), 59
 parse_status_word() (*instruments.keithley.Keithley580* method), 64
 partial (*instruments.tektronix.TekDPO70000.TriggerState* attribute), 115
 pass_fail (*instruments.hp.HP3456a.MathMode* attribute), 40
 pattern (*instruments.teledyne.MAUI.TriggerType* attribute), 130
 pause() (*instruments.srs.SRS830* method), 99
 peak (*instruments.lakeshore.Lakeshore475.Mode* attribute), 73
 peak_detect (*instruments.rigol.RigolDS1000Series.AcquisitionType* attribute), 94
 peak_detect (*instruments.tektronix.TekDPO70000.AcquisitionMode* attribute), 111
 peak_to_peak (*instruments.abstract_instruments.FunctionGenerator.VoltageMode* attribute), 13
 peak_to_peak (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
 percent (*instruments.hp.HP3456a.MathMode* attribute), 41
 period (*instruments.generic_scp.SCPIMultimeter.Mode* attribute), 18
 period (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
 periodic (*instruments.lakeshore.Lakeshore475.PeakMode* attribute), 73
 phase (*instruments.abstract_instruments.FunctionGenerator* attribute), 14
 phase (*instruments.abstract_instruments.FunctionGenerator.Channel* attribute), 13
 phase (*instruments.abstract_instruments.signal_generator.SGChannel* attribute), 15
 phase (*instruments.agilent.Agilent33220a* attribute), 23
 phase (*instruments.generic_scp.SCPIFunctionGenerator* attribute), 22
 phase (*instruments.holzworth.HS9000.Channel* attribute), 39
 phase (*instruments.minghe.MHS5200.Channel* attribute), 75
 phase (*instruments.phasematrix.PhaseMatrixFSW0020* attribute), 89
 phase (*instruments.srs.SRS345* attribute), 97
 phase (*instruments.srs.SRS830* attribute), 101
 phase (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
 phase_max (*instruments.holzworth.HS9000.Channel* attribute), 39
 phase_max (*instruments.srs.SRS830* attribute), 101
 phase_min (*instruments.holzworth.HS9000.Channel* attribute), 39
 phase_min (*instruments.srs.SRS830* attribute), 101
 PhaseMatrixFSW0020 (class in *instruments.phasematrix*), 89
 PicowattAVS47 (class in *instruments.picowatt*), 90
 PicowattAVS47.InputSource (class in *instruments.picowatt*), 90
 PicowattAVS47.Sensor (class in *instruments.picowatt*), 90
 pid (*instruments.thorlabs.TC200* attribute), 149
 PM100USB (class in *instruments.thorlabs*), 134
 PM100USB.MeasurementConfiguration (class in *instruments.thorlabs*), 134
 PM100USB.Sensor (class in *instruments.thorlabs*),

- 135
- PM100USB.SensorFlags (class in *instruments.thorlabs*), 135
- points (*instruments.yokogawa.Yokogawa6370* attribute), 155
- polarity (*instruments.keithley.Keithley580* attribute), 66
- polarity (*instruments.srs.SRSDG645.Output* attribute), 104
- polarity (*instruments.tektronix.TekAWG2000.Channel* attribute), 107
- pos_index_signals (*instruments.newport.NewportESP301HomeSearchMode* attribute), 86
- pos_limit_signal (*instruments.newport.NewportESP301HomeSearchMode* attribute), 86
- position (*instruments.newport.NewportESP301Axis* attribute), 85
- position (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
- position (*instruments.tektronix.TekDPO70000.Math* attribute), 114
- position (*instruments.thorlabs.APTMotorController.MotorChannel* attribute), 143
- position_control_closed (*instruments.thorlabs.APTPiezoStage.PiezoChannel* attribute), 141
- position_count (*instruments.thorlabs.APTPiezoInertiaActuator.PiezoChannel* attribute), 140
- position_display_resolution (*instruments.newport.NewportESP301Axis* attribute), 85
- position_encoder (*instruments.thorlabs.APTMotorController.MotorChannel* attribute), 143
- positive (*instruments.keithley.Keithley580.Polarity* attribute), 64
- positive (*instruments.lakeshore.Lakeshore475.PeakDisplay* attribute), 73
- positive (*instruments.srs.SRSDG645.LevelPolarity* attribute), 104
- power (*instruments.abstract_instruments.signal_generator.SignalGenerator* attribute), 15
- power (*instruments.holzworth.HS9000.Channel* attribute), 39
- power (*instruments.ondax.LM* attribute), 87
- power (*instruments.phasematrix.PhaseMatrixFSW0020* attribute), 89
- power (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 134
- power_density (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 134
- power_max (*instruments.holzworth.HS9000.Channel* attribute), 39
- power_min (*instruments.holzworth.HS9000.Channel* attribute), 40
- power_on (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- power_on (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- power_on_status (*instruments.generic_scp.SCPIInstrument* attribute), 17
- power_scan (*instruments.ondax.LM.Status* attribute), 87
- prescale_config (*instruments.srs.SRSDG645.DisplayMode* attribute), 104
- processing_error (*instruments.glassman.GlassmanFR.ErrorCode* attribute), 36
- product () (*instruments.teledyne.MAUI.Math.Operators* method), 126
- production_date (*instruments.toptica.TopMode.Laser* attribute), 152
- program_mnemonic_too_long (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- program_mnemonic_too_long (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- prompt (*instruments.Instrument* attribute), 11
- proportional_gain (*instruments.newport.NewportESP301Axis* attribute), 85
- ptest (*instruments.keithley.Keithley6514.ArmSource* attribute), 69
- ptc100 (*instruments.thorlabs.TC200.Sensor* attribute), 148
- ptc1000 (*instruments.thorlabs.TC200.Sensor* attribute), 148
- pulse (*instruments.agilent.Agilent33220a.Function* attribute), 22
- pulse (*instruments.lakeshore.Lakeshore475.PeakMode* attribute), 73
- pulse (*instruments.tektronix.TekAWG2000.Shape* attribute), 107
- pulse_modulation (*instruments.phasematrix.PhaseMatrixFSW0020* attribute), 89
- pulse_width (*instruments.keithley.Keithley580.Drive* attribute), 64

Q

- Q (*instruments.glassman.GlassmanFR.ResponseCode attribute*), 36
- qualified (*instruments.teledyne.MAUI.TriggerType attribute*), 130
- query() (*instruments.abstract_instruments.FunctionGenerator.Channel method*), 13
- query() (*instruments.glassman.GlassmanFR method*), 36
- query() (*instruments.holzworth.HS9000.Channel method*), 38
- query() (*instruments.hp.HP6624a.Channel method*), 45
- query() (*instruments.Instrument method*), 11
- query() (*instruments.keithley.Keithley580 method*), 64
- query() (*instruments.rigol.RigolDS1000Series.Channel method*), 95
- query() (*instruments.srs.SRSCTC100 method*), 102
- query() (*instruments.tektronix.TekDPO70000.Channel method*), 111
- query() (*instruments.tektronix.TekDPO70000.Math method*), 113
- query() (*instruments.teledyne.MAUI.Channel method*), 122
- query() (*instruments.teledyne.MAUI.Math method*), 127
- query() (*instruments.teledyne.MAUI.Measurement method*), 127
- query_deadlocked (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- query_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- query_interrupted (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- query_lines() (*instruments.fluke.Fluke3000 method*), 27
- query_terminated (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- query_terminated_after_indefinite_response (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- questionable (*instruments.hp.HP6632b.DFISource attribute*), 48
- R
- r (*instruments.hp.HP3456a attribute*), 44
- r (*instruments.hp.HP3456a.Register attribute*), 41
- r (*instruments.srs.SRS830.Mode attribute*), 98
- r() (*instruments.agilent.Agilent34410a method*), 24
- radians (*instruments.tektronix.TekDPO70000.Math.Phase attribute*), 113
- radio (*instruments.qubitekk.MC1.MotorType attribute*), 92
- ram_cal_checksum_failed (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ram_config_checksum_failed (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ram_rd0_checksum_failed (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ram_rst_checksum_failed (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ram_selftest (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ram_state_checksum_failed (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- ramp (*instruments.abstract_instruments.FunctionGenerator.Function attribute*), 13
- ramp (*instruments.agilent.Agilent33220a.Function attribute*), 22
- ramp (*instruments.srs.SRS345.Function attribute*), 96
- ramp (*instruments.tektronix.TekAWG2000.Shape attribute*), 107
- ramp_rate (*instruments.lakeshore.Lakeshore475 attribute*), 74
- ramp_symmetry (*instruments.agilent.Agilent33220a attribute*), 23
- ratio() (*instruments.teledyne.MAUI.Math.Operators method*), 126
- ratio_acv_dcv (*instruments.hp.HP3456a.Mode attribute*), 41
- ratio_acvdv_dcv (*instruments.hp.HP3456a.Mode attribute*), 41
- ratio_dcv_dcv (*instruments.hp.HP3456a.Mode attribute*), 41
- read() (*instruments.Instrument method*), 11
- read() (*instruments.thorlabs.PM100USB method*), 135
- read_data() (*instruments.agilent.Agilent34410a method*), 24
- read_data_buffer() (*instruments.srs.SRS830 method*), 99
- read_data_nvmem() (*instruments.agilent.Agilent34410a method*), 24
- read_done (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- read_done_ovf (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- read_last_data() (*instruments.agilent.Agilent34410a method*), 24
- read_lines() (*instruments.fluke.Fluke3000 method*),

- 27
- `read_measurements()` (*instruments.keithley.Keithley6514* method), 70
- `read_meter()` (*instruments.agilent.Agilent34410a* method), 25
- `read_ovf` (*instruments.keithley.Keithley485.SRQDataMask* attribute), 61
- `read_setup()` (*instruments.newport.NewportESP301Axis* method), 80
- `read_waveform()` (*instruments.rigol.RigolDS1000Series.DataSource* method), 95
- `read_waveform()` (*instruments.tektronix._TekDPO4104DataSource* method), 110
- `read_waveform()` (*instruments.tektronix.TekDPO70000.DataSource* method), 112
- `read_waveform()` (*instruments.teledyne.MAUI.DataSource* method), 123
- `ready` (*instruments.holzworth.HS9000* attribute), 40
- `ready` (*instruments.tektronix.TekDPO70000.TriggerState* attribute), 115
- `real_time` (*instruments.tektronix.TekDPO70000.SamplingMode* attribute), 115
- `reboot()` (*instruments.toptica.TopMode* method), 152
- `recall_state()` (*instruments.holzworth.HS9000.Channel* method), 38
- `reciprocal()` (*instruments.teledyne.MAUI.Math.Operators* method), 126
- `rectangular` (*instruments.hp.HP6632b.SenseWindow* attribute), 51
- `rectangular` (*instruments.tektronix.TekDPO70000.Math.SpectralWindow* attribute), 113
- `ref` (*instruments.rigol.RigolDS1000Series* attribute), 96
- `ref` (*instruments.srs.SRS830.Mode* attribute), 98
- `ref` (*instruments.tektronix.TekDPO4104* attribute), 109
- `ref` (*instruments.tektronix.TekDPO70000* attribute), 117
- `ref` (*instruments.tektronix.TekTDS224* attribute), 118
- `ref` (*instruments.tektronix.TekTDS5xx* attribute), 121
- `ref` (*instruments.teledyne.MAUI* attribute), 132
- `Ref1` (*instruments.tektronix.TekTDS5xx.Source* attribute), 120
- `Ref2` (*instruments.tektronix.TekTDS5xx.Source* attribute), 120
- `Ref3` (*instruments.tektronix.TekTDS5xx.Source* attribute), 120
- `Ref4` (*instruments.tektronix.TekTDS5xx.Source* attribute), 120
- `ref_output` (*instruments.phasematrix.PhaseMatrixFSW0020* attribute), 89
- `reference` (*instruments.picowatt.PicowattAVS47.InputSource* attribute), 90
- `reference()` (*instruments.toptica.TopMode* method), 152
- `relative` (*instruments.abstract_instruments.Multimeter* attribute), 12
- `relative` (*instruments.fluke.Fluke3000* attribute), 27
- `relative` (*instruments.generic_scp.SCPIMultimeter* attribute), 20
- `relative` (*instruments.hp.HP3456a* attribute), 44
- `relative` (*instruments.keithley.Keithley195* attribute), 60
- `relative` (*instruments.keithley.Keithley2182* attribute), 68
- `relative` (*instruments.keithley.Keithley2182.Channel* attribute), 67
- `relative` (*instruments.keithley.Keithley485* attribute), 63
- `relative` (*instruments.keithley.Keithley485.Status* attribute), 61
- `relative` (*instruments.keithley.Keithley580* attribute), 66
- `relay` (*instruments.qubitek.MC1.MotorType* attribute), 92
- `release_panel()` (*instruments.rigol.RigolDS1000Series* method), 96
- `remote` (*instruments.picowatt.PicowattAVS47* attribute), 91
- `remote` (*instruments.thorlabs.LCC25* attribute), 147
- `remote_inhibit` (*instruments.hp.HP6632b.DigitalFunction* attribute), 48
- `repeat` (*instruments.thorlabs.SC10* attribute), 145
- `repeat` (*instruments.thorlabs.SC10.Mode* attribute), 144
- `REPEAT` (*instruments.yokogawa.Yokogawa6370.SweepModes* attribute), 154
- `request_control_event` (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 16
- `request_control_event` (*instruments.hp.HP6632b.ErrorCodes* attribute), 50
- `request_service_bit` (*instruments.hp.HP6632b.DFISource* attribute), 48
- `rescale()` (*instruments.teledyne.MAUI.Math.Operators* method), 126
- `reset()` (*instruments.fluke.Fluke3000* method), 27
- `reset()` (*instruments.generic_scp.SCPIInstrument* method), 17

- reset () (*instruments.glassman.GlassmanFR method*), 37
- reset () (*instruments.holzworth.HS9000.Channel method*), 38
- reset () (*instruments.hp.HP6624a.Channel method*), 46
- reset () (*instruments.hp.HP6652a method*), 54
- reset () (*instruments.newport.NewportESP301 method*), 79
- reset () (*instruments.ondax.LM method*), 87
- reset () (*instruments.phasematrix.PhaseMatrixFSW0020 method*), 89
- reset () (*instruments.qubitek.MC1 method*), 93
- reset_controller () (*instruments.newport.AGUC2 method*), 76
- resistance (*instruments.fluke.Fluke3000.Mode attribute*), 26
- resistance (*instruments.generic_scp.SCPIMultimeter.Mode attribute*), 18
- resistance (*instruments.hp.HP3456a.ValidRange attribute*), 42
- resistance (*instruments.keithley.Keithley195.Mode attribute*), 58
- resistance (*instruments.keithley.Keithley195.ValidRange attribute*), 58
- resistance (*instruments.keithley.Keithley6514.Mode attribute*), 70
- resistance (*instruments.keithley.Keithley6514.ValidRange attribute*), 70
- resistance (*instruments.lakeshore.Lakeshore370.Channel attribute*), 72
- resistance (*instruments.picowatt.PicowattAVS47.Sensor attribute*), 90
- resistance (*instruments.thorlabs.PM100USB.MeasurementConfiguration attribute*), 134
- resistance_2wire (*instruments.hp.HP3456a.Mode attribute*), 41
- resistance_4wire (*instruments.hp.HP3456a.Mode attribute*), 41
- resolution (*instruments.generic_scp.SCPIMultimeter attribute*), 20
- response_settable (*instruments.thorlabs.PM100USB.SensorFlags attribute*), 135
- restore () (*instruments.thorlabs.SC10 method*), 144
- right_limit (*instruments.newport.NewportESP301Axis attribute*), 85
- RigolDS1000Series (*class in instruments.rigol*), 94
- RigolDS1000Series.AcquisitionType (*class in instruments.rigol*), 94
- RigolDS1000Series.Channel (*class in instruments.rigol*), 94
- RigolDS1000Series.Channel.Coupling (*class in instruments.rigol*), 95
- RigolDS1000Series.DataSource (*class in instruments.rigol*), 95
- rise_time_10_90 (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- rise_time_20_80 (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- Rising (*instruments.tektronix.TekTDS5xx.Edge attribute*), 119
- rms (*instruments.abstract_instruments.FunctionGenerator.VoltageMode attribute*), 13
- rms (*instruments.lakeshore.Lakeshore475.Mode attribute*), 73
- rms (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- Modef () (*instruments.teledyne.MAUI.Math.Operators method*), 126
- rox (*instruments.srs.SRSCTC100.SensorType attribute*), 102
- rs232_recv_framing_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- rs232_recv_overrun_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- rs232_recv_parity_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
- rtd (*instruments.srs.SRSCTC100.SensorType attribute*), 102
- run (*instruments.tektronix.TekDPO70000.AcquisitionState attribute*), 111
- run () (*instruments.rigol.RigolDS1000Series method*), 96
- run () (*instruments.tektronix.TekDPO70000 method*), 115
- run () (*instruments.teledyne.MAUI method*), 131
- run_program () (*instruments.newport.NewportESP301 method*), 79
- run_stop (*instruments.tektronix.TekDPO70000.StopAfter attribute*), 115
- runt (*instruments.teledyne.MAUI.TriggerType attribute*), 130

S

- S (*instruments.glassman.GlassmanFR.ResponseCode attribute*), 36
- sample (*instruments.tektronix.TekDPO70000.AcquisitionMode attribute*), 111
- sample_count (*instruments.generic_scp.SCPIMultimeter attribute*),

- 20
- sample_rate (*instruments.srs.SRS830 attribute*), 101
- sample_source (*instruments.generic_scp.SCPIMultimeter attribute*), 20
- sample_timer (*instruments.generic_scp.SCPIMultimeter attribute*), 21
- save () (*instruments.ondax.LM method*), 87
- save () (*instruments.thorlabs.LCC25 method*), 146
- save () (*instruments.thorlabs.SC10 method*), 144
- save_mode () (*instruments.thorlabs.SC10 method*), 144
- save_state () (*instruments.holzworth.HS9000.Channel method*), 39
- sawtooth_down (*instruments.minghe.MHS5200.Function attribute*), 75
- sawtooth_up (*instruments.minghe.MHS5200.Function attribute*), 75
- SC10 (*class in instruments.thorlabs*), 144
- SC10.Mode (*class in instruments.thorlabs*), 144
- scale (*instruments.gentec_eo.Blu attribute*), 32
- scale (*instruments.hp.HP3456a.MathMode attribute*), 41
- scale (*instruments.tektronix.TekDPO70000.Channel attribute*), 112
- scale (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- scale (*instruments.teledyne.MAUI.Channel attribute*), 123
- scale_down () (*instruments.gentec_eo.Blu method*), 30
- scale_factors (*instruments.thorlabs.APTMotorController.MotorChannel attribute*), 143
- scale_up () (*instruments.gentec_eo.Blu method*), 30
- scan () (*instruments.fluke.Fluke3000 method*), 27
- scpi_version (*instruments.generic_scp.SCPIInstrument attribute*), 17
- SCPIFunctionGenerator (*class in instruments.generic_scp*), 21
- SCPIInstrument (*class in instruments.generic_scp*), 15
- SCPIInstrument.ErrorCodes (*class in instruments.generic_scp*), 15
- SCPIMultimeter (*class in instruments.generic_scp*), 18
- SCPIMultimeter.InputRange (*class in instruments.generic_scp*), 18
- SCPIMultimeter.Mode (*class in instruments.generic_scp*), 18
- SCPIMultimeter.Resolution (*class in instruments.generic_scp*), 18
- SCPIMultimeter.SampleCount (*class in instruments.generic_scp*), 19
- SCPIMultimeter.SampleSource (*class in instruments.generic_scp*), 19
- SCPIMultimeter.TriggerCount (*class in instruments.generic_scp*), 19
- SCPIMultimeter.TriggerMode (*class in instruments.generic_scp*), 19
- search_for_home () (*instruments.newport.NewportESP301 method*), 79
- search_for_home () (*instruments.newport.NewportESP301Axis method*), 81
- select_fastest_encoding () (*instruments.tektronix.TekDPO70000 method*), 115
- self_test (*instruments.teledyne.MAUI attribute*), 132
- self_test_ok (*instruments.generic_scp.SCPIInstrument attribute*), 18
- sendcmd () (*instruments.abstract_instruments.FunctionGenerator.Channel method*), 13
- sendcmd () (*instruments.glassman.GlassmanFR method*), 37
- sendcmd () (*instruments.holzworth.HS9000.Channel method*), 39
- sendcmd () (*instruments.hp.HP6624a.Channel method*), 46
- sendcmd () (*instruments.Instrument method*), 11
- sendcmd () (*instruments.keithley.Keithley580 method*), 65
- sendcmd () (*instruments.rigol.RigolDS1000Series.Channel method*), 95
- sendcmd () (*instruments.srs.SRSCTC100 method*), 103
- sendcmd () (*instruments.tektronix.TekDPO70000.Channel method*), 112
- sendcmd () (*instruments.tektronix.TekDPO70000.Math method*), 114
- sendcmd () (*instruments.teledyne.MAUI.Channel method*), 122
- sendcmd () (*instruments.teledyne.MAUI.Math method*), 127
- sendcmd () (*instruments.teledyne.MAUI.Measurement method*), 128
- sense_sweep_interval (*instruments.hp.HP6632b attribute*), 53
- sense_sweep_points (*instruments.hp.HP6632b attribute*), 53
- sense_window (*instruments.hp.HP6632b attribute*), 53

- sensor (*instruments.lakeshore.Lakeshore340* attribute), 71
- sensor (*instruments.oxford.OxfordITC503* attribute), 88
- sensor (*instruments.picowatt.PicowattAVS47* attribute), 91
- sensor (*instruments.thorlabs.PM100USB* attribute), 136
- sensor (*instruments.thorlabs.TC200* attribute), 149
- sensor_type (*instruments.srs.SRSCTC100.Channel* attribute), 102
- sequence (*instruments.tektronix.TekDPO70000.StopAfter* attribute), 115
- serial_number (*instruments.minghe.MHS5200* attribute), 75
- serial_number (*instruments.ondax.LM* attribute), 87
- serial_number (*instruments.thorlabs.PM100USB.Sensor* attribute), 135
- serial_number (*instruments.thorlabs.ThorLabsAPT* attribute), 137
- serial_number (*instruments.toptica.TopMode* attribute), 153
- serial_number (*instruments.toptica.TopMode.Laser* attribute), 152
- set () (*instruments.toptica.TopMode* method), 152
- set_calibration_value () (*instruments.keithley.Keithley580* method), 65
- set_channel_display () (*instruments.srs.SRS830* method), 99
- set_offset_expand () (*instruments.srs.SRS830* method), 99
- set_parameter () (*instruments.teledyne.MAUI.Measurement* method), 128
- set_scale () (*instruments.thorlabs.APTMotorController.MotorChannel* method), 143
- set_settings () (*instruments.thorlabs.LCC25* method), 146
- set_status () (*instruments.glassman.GlassmanFR* method), 37
- setting (*instruments.qubitekk.MC1* attribute), 94
- setup_axis () (*instruments.newport.NewportESP301Axis* method), 81
- SGChannel (class in *instruments.abstract_instruments.signal_generator*), 15
- shape (*instruments.tektronix.TekAWG2000.Channel* attribute), 107
- shifted (*instruments.tektronix.TekDPO70000.Math.FilterMode* attribute), 113
- show_id (*instruments.teledyne.MAUI* attribute), 132
- show_options (*instruments.teledyne.MAUI* attribute), 133
- shut_time (*instruments.thorlabs.SC10* attribute), 145
- shutdown_current (*instruments.ondax.LM.Status* attribute), 87
- shutdown_overheat (*instruments.ondax.LM.Status* attribute), 87
- SignalGenerator (class in *instruments.abstract_instruments.signal_generator*), 14
- sine (*instruments.minghe.MHS5200.Function* attribute), 75
- sine (*instruments.tektronix.TekAWG2000.Shape* attribute), 107
- single (*instruments.hp.HP3456a.TriggerMode* attribute), 42
- single (*instruments.teledyne.MAUI.TriggerState* attribute), 130
- single (*instruments.thorlabs.SC10.Mode* attribute), 144
- SINGLE (*instruments.yokogawa.Yokogawa6370.SweepModes* attribute), 154
- single_shot (*instruments.srs.SRSDG645.TriggerSource* attribute), 105
- single_shot_energy_mode (*instruments.gentec_eo.Blu* attribute), 33
- SingleChannelSG (class in *instruments.abstract_instruments.signal_generator*), 14
- sinusoid (*instruments.abstract_instruments.FunctionGenerator.Function* attribute), 13
- sinusoid (*instruments.agilent.Agilent33220a.Function* attribute), 22
- sinusoid (*instruments.srs.SRS345.Function* attribute), 96
- sinx () (*instruments.teledyne.MAUI.Math.Operators* method), 126
- sleep_time (*instruments.newport.AGUC2* attribute), 77
- slew_rate (*instruments.teledyne.MAUI.TriggerType* attribute), 130
- sources (*instruments.tektronix.TekTDS5xx* attribute), 121
- span (*instruments.yokogawa.Yokogawa6370* attribute), 155
- spectral_center (*instruments.tektronix.TekDPO70000.Math* attribute), 114
- spectral_gatepos (*instruments.tektronix.TekDPO70000.Math* attribute), 114
- spectral_gatewidth (*instruments.tektronix.TekDPO70000.Math* attribute), 114

- 114
- spectral_lock (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_mag (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_phase (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_reflevel (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_reflevel_offset (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_resolution_bandwidth (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_span (*instruments.tektronix.TekDPO70000.Math attribute*), 114
- spectral_suppress (*instruments.tektronix.TekDPO70000.Math attribute*), 115
- spectral_unwrap (*instruments.tektronix.TekDPO70000.Math attribute*), 115
- spectral_window (*instruments.tektronix.TekDPO70000.Math attribute*), 115
- split_unit_str() (*in module instruments.util_fns*), 163
- square (*instruments.abstract_instruments.FunctionGenerator.Function attribute*), 13
- square (*instruments.agilent.Agilent33220a.Function attribute*), 22
- square (*instruments.minghe.MHS5200.Function attribute*), 75
- square (*instruments.srs.SRS345.Function attribute*), 96
- square (*instruments.tektronix.TekAWG2000.Shape attribute*), 107
- square() (*instruments.teledyne.MAUI.Math.Operators method*), 126
- square_root() (*instruments.teledyne.MAUI.Math.Operators method*), 126
- srq_disabled (*instruments.keithley.Keithley485.SRQDataMask attribute*), 61
- srq_disabled (*instruments.keithley.Keithley485.SRQErrorMask attribute*), 61
- SRS345 (*class in instruments.srs*), 96
- SRS345.Function (*class in instruments.srs*), 96
- SRS830 (*class in instruments.srs*), 97
- SRS830.BufferMode (*class in instruments.srs*), 97
- SRS830.Coupling (*class in instruments.srs*), 97
- SRS830.FreqSource (*class in instruments.srs*), 97
- SRS830.Mode (*class in instruments.srs*), 97
- SRSTC100 (*class in instruments.srs*), 101
- SRSTC100.Channel (*class in instruments.srs*), 101
- SRSTC100.SensorType (*class in instruments.srs*), 102
- SRS645 (*class in instruments.srs*), 103
- SRS645.Channels (*class in instruments.srs*), 103
- SRS645.DisplayMode (*class in instruments.srs*), 104
- SRS645.LevelPolarity (*class in instruments.srs*), 104
- SRS645.Output (*class in instruments.srs*), 104
- SRS645.Outputs (*class in instruments.srs*), 104
- SRS645.TriggerSource (*class in instruments.srs*), 105
- ss_external_falling (*instruments.srs.SRS645.TriggerSource attribute*), 105
- ss_external_rising (*instruments.srs.SRS645.TriggerSource attribute*), 105
- start_data_transfer() (*instruments.srs.SRS830 method*), 99
- start_scan() (*instruments.srs.SRS830 method*), 99
- start_sweep() (*instruments.yokogawa.Yokogawa6370 method*), 155
- start_time (*instruments.newport.NewportError attribute*), 86
- start_wl (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- start_wl_max (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- start_wl_min (*instruments.yokogawa.Yokogawa6370 attribute*), 155
- statistic (*instruments.hp.HP3456a.MathMode attribute*), 41
- statistics (*instruments.teledyne.MAUI.Measurement attribute*), 128
- statistics (*instruments.teledyne.MAUI.Measurement.State attribute*), 127
- stats_enabled (*instruments.srs.SRSCTC100.Channel attribute*), 102
- stats_points (*instruments.srs.SRSCTC100.Channel attribute*), 102
- status (*instruments.ondax.LM attribute*), 88
- status (*instruments.thorlabs.TC200 attribute*), 149
- status_bits (*instruments*), 149

ments.thorlabs.APTMotorController.MotorChannel attribute), 144
 std_dev (*instruments.srs.SRSCTC100.Channel attribute*), 102
 stdev (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
 step_amplitude (*instruments.newport._Axis attribute*), 78
 step_delay (*instruments.newport._Axis attribute*), 78
 step_size (*instruments.qubitekk.MC1 attribute*), 94
 stest (*instruments.keithley.Keithley6514.ArmSource attribute*), 70
 stop (*instruments.tektronix.TekDPO70000.AcquisitionState attribute*), 111
 stop (*instruments.teledyne.MAUI.TriggerState attribute*), 130
 stop() (*instruments.newport._Axis method*), 77
 stop() (*instruments.rigol.RigolDS1000Series method*), 96
 stop() (*instruments.tektronix.TekDPO70000 method*), 115
 stop() (*instruments.teledyne.MAUI method*), 131
 stop_motion() (*instruments.newport.NewportESP301Axis method*), 82
 stop_wl (*instruments.yokogawa.Yokogawa6370 attribute*), 155
 stop_wl_max (*instruments.yokogawa.Yokogawa6370 attribute*), 155
 stop_wl_min (*instruments.yokogawa.Yokogawa6370 attribute*), 156
 store_calibration_constants() (*instruments.keithley.Keithley580 method*), 65
 string_data_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 string_data_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 50
 string_data_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 string_data_not_allowed (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
 string_property() (*in module instruments.util_fns*), 170
 subtract (*instruments.qubitekk.CC1 attribute*), 92
 success (*instruments.toptica.TopMode.CharmStatus attribute*), 150
 suffix_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 suffix_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
 suffix_not_allowed (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 suffix_not_allowed (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
 suffix_too_long (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 16
 suffix_too_long (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
 sum() (*instruments.teledyne.MAUI.Math.Operators method*), 126
 sweep_mode (*instruments.yokogawa.Yokogawa6370 attribute*), 156
 syntax_error (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 17
 syntax_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
 system_error (*instruments.hp.HP6632b.ErrorCodes attribute*), 51

T

T0 (*instruments.srs.SRSDG645.Channels attribute*), 104
 T0 (*instruments.srs.SRSDG645.Outputs attribute*), 105
 T1 (*instruments.srs.SRSDG645.Channels attribute*), 104
 t3000 (*instruments.fluke.Fluke3000.Module attribute*), 26
 take_measurement() (*instruments.srs.SRS830 method*), 99
 talk_continuous (*instruments.keithley.Keithley195.TriggerMode attribute*), 58
 talk_continuous (*instruments.keithley.Keithley580.TriggerMode attribute*), 64
 talk_one_shot (*instruments.keithley.Keithley195.TriggerMode attribute*), 58
 talk_one_shot (*instruments.keithley.Keithley580.TriggerMode attribute*), 64
 tau_settable (*instruments.thorlabs.PM100USB.SensorFlags attribute*), 135
 TC200 (*class in instruments.thorlabs*), 148
 TC200.Mode (*class in instruments.thorlabs*), 148
 TC200.Sensor (*class in instruments.thorlabs*), 148
 tec_status (*instruments.toptica.TopMode.Laser attribute*), 152

tek_exponential (*instruments.tektronix.TekDPO70000.Math.SpectralWindow* attribute), 113
 TekAWG2000 (*class in instruments.tektronix*), 107
 TekAWG2000.Channel (*class in instruments.tektronix*), 107
 TekAWG2000.Polarity (*class in instruments.tektronix*), 107
 TekAWG2000.Shape (*class in instruments.tektronix*), 107
 TekDPO4104 (*class in instruments.tektronix*), 108
 TekDPO4104.Coupling (*class in instruments.tektronix*), 108
 TekDPO70000 (*class in instruments.tektronix*), 110
 TekDPO70000.AcquisitionMode (*class in instruments.tektronix*), 110
 TekDPO70000.AcquisitionState (*class in instruments.tektronix*), 111
 TekDPO70000.BinaryFormat (*class in instruments.tektronix*), 111
 TekDPO70000.ByteOrder (*class in instruments.tektronix*), 111
 TekDPO70000.Channel (*class in instruments.tektronix*), 111
 TekDPO70000.Channel.Coupling (*class in instruments.tektronix*), 111
 TekDPO70000.DataSource (*class in instruments.tektronix*), 112
 TekDPO70000.HorizontalMode (*class in instruments.tektronix*), 112
 TekDPO70000.Math (*class in instruments.tektronix*), 113
 TekDPO70000.Math.FilterMode (*class in instruments.tektronix*), 113
 TekDPO70000.Math.Mag (*class in instruments.tektronix*), 113
 TekDPO70000.Math.Phase (*class in instruments.tektronix*), 113
 TekDPO70000.Math.SpectralWindow (*class in instruments.tektronix*), 113
 TekDPO70000.SamplingMode (*class in instruments.tektronix*), 115
 TekDPO70000.StopAfter (*class in instruments.tektronix*), 115
 TekDPO70000.TriggerState (*class in instruments.tektronix*), 115
 TekDPO70000.WaveformEncoding (*class in instruments.tektronix*), 115
 TekTDS224 (*class in instruments.tektronix*), 118
 TekTDS224.Coupling (*class in instruments.tektronix*), 118
 TekTDS5xx (*class in instruments.tektronix*), 119
 TekTDS5xx.Bandwidth (*class in instruments.tektronix*), 119
 TekTDS5xx.Coupling (*class in instruments.tektronix*), 119
 TekTDS5xx.Edge (*class in instruments.tektronix*), 119
 TekTDS5xx.Impedance (*class in instruments.tektronix*), 119
 TekTDS5xx.Source (*class in instruments.tektronix*), 119
 TekTDS5xx.Trigger (*class in instruments.tektronix*), 120
 temp_units (*instruments.lakeshore.Lakeshore475* attribute), 74
 temperature (*instruments.fluke.Fluke3000.Mode* attribute), 26
 temperature (*instruments.generic_scp.SCPIMultimeter.Mode* attribute), 18
 temperature (*instruments.holzworth.HS9000.Channel* attribute), 40
 temperature (*instruments.keithley.Keithley2182.Mode* attribute), 67
 temperature (*instruments.lakeshore.Lakeshore340.Sensor* attribute), 71
 temperature (*instruments.ondax.LM* attribute), 88
 temperature (*instruments.oxford.OxfordITC503.Sensor* attribute), 88
 temperature (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 135
 temperature (*instruments.thorlabs.TC200* attribute), 149
 temperature_control_status (*instruments.toptica.TopMode.Laser* attribute), 152
 temperature_set (*instruments.thorlabs.TC200* attribute), 149
 temperature_status (*instruments.toptica.TopMode* attribute), 154
 termination (*instruments.tektronix.TekDPO70000.Channel* attribute), 112
 terminator (*instruments.Instrument* attribute), 12
 test_mode() (*instruments.thorlabs.LCC25* method), 146
 th10k (*instruments.thorlabs.TC200.Sensor* attribute), 148
 thermistor (*instruments.srs.SRSC7C100.SensorType* attribute), 102
 thermistor_c (*instruments.hp.HP3456a.MathMode* attribute), 41
 thermistor_f (*instruments.hp.HP3456a.MathMode*

- attribute), 41
- theta (*instruments.srs.SRS830.Mode* attribute), 98
- ThorLabsAPT (*class in instruments.thorlabs*), 136
- ThorLabsAPT.APTChannel (*class in instruments.thorlabs*), 136
- threshold (*instruments.tektronix.TekDPO70000.Math* attribute), 115
- time_div (*instruments.teledyne.MAUI* attribute), 133
- timeout (*instruments.Instrument* attribute), 12
- timer (*instruments.generic_scp.SCPIMultimeter.SampleSource* attribute), 19
- timer (*instruments.keithley.Keithley2182.TriggerMode* attribute), 67
- timer (*instruments.keithley.Keithley6514.ArmSource* attribute), 70
- timestamp (*instruments.newport.NewportError* attribute), 86
- tlink (*instruments.keithley.Keithley6514.ArmSource* attribute), 70
- tlink (*instruments.keithley.Keithley6514.TriggerMode* attribute), 70
- too_many_digits (*instruments.generic_scp.SCPIInstrument.ErrorCodes* attribute), 17
- too_many_digits (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
- too_many_errors (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
- too_many_sweep_points (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
- too_much_data (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
- top (*instruments.teledyne.MAUI.MeasurementParameters* attribute), 129
- TopMode (*class in instruments.toptica*), 150
- TopMode.CharmStatus (*class in instruments.toptica*), 150
- TopMode.Laser (*class in instruments.toptica*), 150
- trace (*instruments.teledyne.MAUI.DataSource* attribute), 124
- trajectory (*instruments.newport.NewportESP301Axis* attribute), 85
- trend() (*instruments.teledyne.MAUI.Math.Operators* method), 126
- triangle (*instruments.abstract_instruments.FunctionGenerator.FunctionGenerator* attribute), 13
- triangle (*instruments.srs.SRS345.Function* attribute), 97
- triangle (*instruments.tektronix.TekAWG2000.Shape* attribute), 108
- triangular (*instruments.minghe.MHS5200.Function* attribute), 75
- trigger (*instruments.thorlabs.SC10* attribute), 145
- trigger() (*instruments.generic_scp.SCPIInstrument* method), 17
- trigger() (*instruments.hp.HP3456a* method), 43
- trigger() (*instruments.keithley.Keithley195* method), 59
- trigger() (*instruments.keithley.Keithley580* method), 65
- trigger() (*instruments.yokogawa.Yokogawa7651* method), 157
- trigger_continuous (*instruments.keithley.Keithley580.TriggerMode* attribute), 64
- trigger_count (*instruments.generic_scp.SCPIMultimeter* attribute), 21
- trigger_coupling (*instruments.tektronix.TekTDS5xx* attribute), 121
- trigger_delay (*instruments.generic_scp.SCPIMultimeter* attribute), 21
- trigger_delay (*instruments.teledyne.MAUI* attribute), 133
- trigger_holdoff (*instruments.srs.SRSDG645.DisplayMode* attribute), 104
- trigger_level (*instruments.gentec_eo.Blu* attribute), 33
- trigger_level (*instruments.tektronix.TekTDS5xx* attribute), 121
- trigger_line (*instruments.srs.SRSDG645.DisplayMode* attribute), 104
- trigger_mode (*instruments.abstract_instruments.Multimeter* attribute), 12
- trigger_mode (*instruments.fluke.Fluke3000* attribute), 27
- trigger_mode (*instruments.generic_scp.SCPIMultimeter* attribute), 21
- trigger_mode (*instruments.hp.HP3456a* attribute), 44
- trigger_mode (*instruments.keithley.Keithley195* attribute), 60
- trigger_mode (*instruments.keithley.Keithley2182.Channel* attribute), 67
- trigger_mode (*instruments.keithley.Keithley485* attribute), 63
- trigger_mode (*instruments.keithley.Keithley580* attribute), 66

- trigger_mode (*instruments.keithley.Keithley6514 attribute*), 71
- trigger_mode (*instruments.qubitekk.CC1 attribute*), 92
- trigger_one_shot (*instruments.keithley.Keithley580.TriggerMode attribute*), 64
- trigger_rate (*instruments.srs.SRSDG645 attribute*), 106
- trigger_rate (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- trigger_single_shot (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- trigger_slope (*instruments.tektronix.TekTDS5xx attribute*), 121
- trigger_source (*instruments.srs.SRSDG645 attribute*), 106
- trigger_source (*instruments.tektronix.TekTDS5xx attribute*), 121
- trigger_source (*instruments.teledyne.MAUI attribute*), 133
- trigger_state (*instruments.tektronix.TekDPO70000 attribute*), 118
- trigger_state (*instruments.teledyne.MAUI attribute*), 134
- trigger_threshold (*instruments.srs.SRSDG645.DisplayMode attribute*), 104
- trigger_type (*instruments.teledyne.MAUI attribute*), 134
- tv (*instruments.teledyne.MAUI.TriggerType attribute*), 130
- Twenty (*instruments.tektronix.TekTDS5xx.Bandwidth attribute*), 119
- TwoHundred (*instruments.tektronix.TekTDS5xx.Bandwidth attribute*), 119
- type (*instruments.thorlabs.PM100USB.Sensor attribute*), 135
- U**
- uint (*instruments.tektronix.TekDPO70000.BinaryFormat attribute*), 111
- un_initialized (*instruments.toptica.TopMode.CharmStatus attribute*), 150
- undefined_command (*instruments.glassman.GlassmanFR.ErrorCode attribute*), 36
- undefined_header (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 17
- undefined_header (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
- unexpected_number_of_parameters (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 17
- unexpected_number_of_parameters (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
- unit (*instruments.keithley.Keithley6514 attribute*), 71
- unit_string (*instruments.tektronix.TekDPO70000.Math attribute*), 115
- unitful_property() (*in module instruments.util_fns*), 169
- unitless_property() (*in module instruments.util_fns*), 168
- units (*instruments.keithley.Keithley2182 attribute*), 68
- units (*instruments.newport.NewportESP301Axis attribute*), 85
- units (*instruments.srs.SRSCTC100.Channel attribute*), 102
- upload_waveform() (*instruments.tektronix.TekAWG2000 method*), 108
- upper (*instruments.hp.HP3456a attribute*), 44
- upper (*instruments.hp.HP3456a.Register attribute*), 41
- upper_limit (*instruments.qubitekk.MC1 attribute*), 94
- URI_SCHEMES (*instruments.Instrument attribute*), 11
- usb_state (*instruments.gentec_eo.Blu attribute*), 33
- user (*instruments.agilent.Agilent33220a.Function attribute*), 22
- user_multiplier (*instruments.gentec_eo.Blu attribute*), 33
- user_offset (*instruments.gentec_eo.Blu attribute*), 34
- user_request_event (*instruments.generic_scp.SCPIInstrument.ErrorCodes attribute*), 17
- user_request_event (*instruments.hp.HP6632b.ErrorCodes attribute*), 51
- V**
- V (*instruments.glassman.GlassmanFR.ResponseCode attribute*), 36
- value (*instruments.srs.SRSCTC100.Channel attribute*), 102
- variance (*instruments.hp.HP3456a attribute*), 45
- variance (*instruments.hp.HP3456a.Register attribute*), 41
- vdac_idac_selftest1 (*instruments.hp.HP6632b.ErrorCodes attribute*), 51

vdac_idac_selftest2 (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
 vdac_idac_selftest3 (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
 vdac_idac_selftest4 (*instruments.hp.HP6632b.ErrorCodes* attribute), 51
 velocity (*instruments.newport.NewportESP301Axis* attribute), 85
 vernier (*instruments.rigol.RigolDS1000Series.Channel* attribute), 95
 version (*instruments.gentec_eo.Blu* attribute), 34
 version (*instruments.glassman.GlassmanFR* attribute), 38
 VERT_DIVS (*instruments.tektronix.TekDPO70000* attribute), 115
 voltage (*instruments.glassman.GlassmanFR* attribute), 38
 voltage (*instruments.glassman.GlassmanFR.Mode* attribute), 36
 voltage (*instruments.hp.HP3456a.ValidRange* attribute), 42
 voltage (*instruments.hp.HP6624a* attribute), 47
 voltage (*instruments.hp.HP6624a.Channel* attribute), 46
 voltage (*instruments.hp.HP6624a.Mode* attribute), 47
 voltage (*instruments.hp.HP6652a* attribute), 55
 voltage (*instruments.hp.HPe3631a* attribute), 57
 voltage (*instruments.keithley.Keithley6220* attribute), 69
 voltage (*instruments.keithley.Keithley6514.Mode* attribute), 70
 voltage (*instruments.keithley.Keithley6514.ValidRange* attribute), 70
 voltage (*instruments.newport.NewportESP301Axis* attribute), 85
 voltage (*instruments.thorlabs.PM100USB.MeasurementConfiguration* attribute), 135
 voltage (*instruments.yokogawa.Yokogawa7651* attribute), 157
 voltage (*instruments.yokogawa.Yokogawa7651.Channel* attribute), 156
 voltage (*instruments.yokogawa.Yokogawa7651.Mode* attribute), 157
 voltage1 (*instruments.thorlabs.LCC25* attribute), 147
 voltage1 (*instruments.thorlabs.LCC25.Mode* attribute), 146
 voltage2 (*instruments.thorlabs.LCC25* attribute), 147
 voltage2 (*instruments.thorlabs.LCC25.Mode* attribute), 146
 voltage_ac (*instruments.fluke.Fluke3000.Mode* attribute), 26
 voltage_ac (*instruments.generic_scp.SCPIMultimeter.Mode* attribute), 18
 voltage_ac (*instruments.keithley.Keithley195.Mode* attribute), 58
 voltage_ac (*instruments.keithley.Keithley195.ValidRange* attribute), 58
 voltage_alc_bandwidth (*instruments.hp.HP6632b* attribute), 53
 voltage_dc (*instruments.fluke.Fluke3000.Mode* attribute), 26
 voltage_dc (*instruments.generic_scp.SCPIMultimeter.Mode* attribute), 18
 voltage_dc (*instruments.keithley.Keithley195.Mode* attribute), 58
 voltage_dc (*instruments.keithley.Keithley195.ValidRange* attribute), 58
 voltage_dc (*instruments.keithley.Keithley2182.Mode* attribute), 67
 voltage_max (*instruments.hp.HPe3631a* attribute), 57
 voltage_min (*instruments.hp.HPe3631a* attribute), 57
 voltage_range (*instruments.hp.HPe3631a* attribute), 57
 voltage_sense (*instruments.glassman.GlassmanFR* attribute), 38
 voltage_sense (*instruments.hp.HP6624a* attribute), 47
 voltage_sense (*instruments.hp.HP6624a.Channel* attribute), 46
 voltage_sense (*instruments.hp.HP6652a* attribute), 55
 voltage_sense (*instruments.hp.HPe3631a* attribute), 57
 voltage_trigger (*instruments.hp.HP6632b* attribute), 53

W

wait_for_motion() (*instruments.newport.NewportESP301Axis* method), 82
 wait_for_position() (*instruments.newport.NewportESP301Axis* method), 82
 wait_for_stop() (*instruments.newport.NewportESP301Axis* method), 82
 wait_to_continue() (*instruments.generic_scp.SCPIInstrument* method), 17
 waiting (*instruments.ondax.LM.Status* attribute), 87
 waiting_stable_temperature (*instruments.ondax.LM.Status* attribute), 87
 waveform_db (*instruments.tektronix.TekDPO70000.AcquisitionMode* attribute), 111

- waveform_name (*instruments.tektronix.TekAWG2000 attribute*), 108
- wavelength (*instruments.gentec_eo.Blu attribute*), 34
- wavelength (*instruments.toptica.TopMode.Laser attribute*), 152
- wavelength() (*instruments.yokogawa.Yokogawa6370 method*), 155
- wavelength() (*instruments.yokogawa.Yokogawa6370.Channel method*), 154
- wavelength_settable (*instruments.thorlabs.PM100USB.SensorFlags attribute*), 135
- wide (*instruments.lakeshore.Lakeshore475.Filter attribute*), 73
- width (*instruments.teledyne.MAUI.TriggerType attribute*), 130
- width_50_neg (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- width_50_pos (*instruments.teledyne.MAUI.MeasurementParameters attribute*), 129
- window (*instruments.qubitekk.CC1 attribute*), 92
- write() (*instruments.Instrument method*), 11
- ## X
- X (*instruments.newport.AGUC2.Axes attribute*), 76
- x (*instruments.srs.SRS830.Mode attribute*), 98
- x_continuous (*instruments.keithley.Keithley195.TriggerMode attribute*), 58
- x_one_shot (*instruments.keithley.Keithley195.TriggerMode attribute*), 58
- xnoise (*instruments.srs.SRS830.Mode attribute*), 98
- ## Y
- y (*instruments.hp.HP3456a attribute*), 45
- y (*instruments.hp.HP3456a.Register attribute*), 41
- Y (*instruments.newport.AGUC2.Axes attribute*), 76
- y (*instruments.srs.SRS830.Mode attribute*), 98
- y_offset (*instruments.tektronix._TekDPO4104DataSource attribute*), 110
- y_offset (*instruments.tektronix.TekDPO4104 attribute*), 109
- ynoise (*instruments.srs.SRS830.Mode attribute*), 98
- Yokogawa6370 (*class in instruments.yokogawa*), 154
- Yokogawa6370.Channel (*class in instruments.yokogawa*), 154
- Yokogawa6370.SweepModes (*class in instruments.yokogawa*), 154
- Yokogawa6370.Traces (*class in instruments.yokogawa*), 155
- Yokogawa7651 (*class in instruments.yokogawa*), 156
- Yokogawa7651.Channel (*class in instruments.yokogawa*), 156
- Yokogawa7651.Mode (*class in instruments.yokogawa*), 156
- ## Z
- z (*instruments.hp.HP3456a attribute*), 45
- z (*instruments.hp.HP3456a.Register attribute*), 42
- zero_check (*instruments.keithley.Keithley485 attribute*), 63
- zero_check (*instruments.keithley.Keithley6514 attribute*), 71
- zero_correct (*instruments.keithley.Keithley6514 attribute*), 71
- zero_offset (*instruments.gentec_eo.Blu attribute*), 35
- zero_position() (*instruments.newport._Axis method*), 77
- zero_position_count (*instruments.newport.NewportESP301HomeSearchMode attribute*), 86
- zerocheck (*instruments.keithley.Keithley485.Status attribute*), 62