
Lake Shore Python Driver Documentation

Release 1.10.0

Lake Shore Cryotronics, Inc.

Mar 23, 2026

CONTENTS

1	Supported Products	3
2	Table of contents	5
2.1	Installation	5
2.1.1	Python version	5
2.1.2	Install the Lake Shore Python driver	5
2.1.3	Installing the driver through Spyder	5
2.2	Getting Started	6
2.2.1	A simple example	6
2.2.2	Making Connections	6
2.2.3	Commands and queries	7
2.2.4	SCPI commands and queries	7
2.3	Advanced	7
2.3.1	Thread Safety	7
2.3.2	Logging	7
2.3.3	Status Registers	8
2.3.4	Instrument initialization options	9
2.3.5	Alternative instrument connections	9
2.3.6	Python 2 compatibility	9
2.4	Supported Instruments	9
2.4.1	Magnetics Instruments	9
2.4.2	Magnet System Power Supplies	20
2.4.3	Materials Characterization	32
2.4.4	Temperature Controllers	90
2.4.5	Temperature Monitors	234
2.4.6	Sources	260
	Python Module Index	271
	Index	273

The [Lake Shore](#) python driver allows users to quickly and easily communicate with Lake Shore instruments. It automatically establishes a connection and provides a variety of functions specific to the product such as configuring settings and acquiring measurements. This driver is created and maintained by Lake Shore. Please visit the [github page](#) to report issues or request features.

Begin by completing the [Installation](#) process then read up on [Getting Started](#) with the driver.

SUPPORTED PRODUCTS

Some products are fully supported by the driver and do not require knowledge of the remote interface commands and queries. Instruments with basic support will establish a connection but require familiarity with the product's commands and queries.

Visit the *Supported Instruments* section to view a complete list of products supported by the driver.

TABLE OF CONTENTS

2.1 Installation

2.1.1 Python version

The Lake Shore Python driver is compatible with Python 3.7 and above.

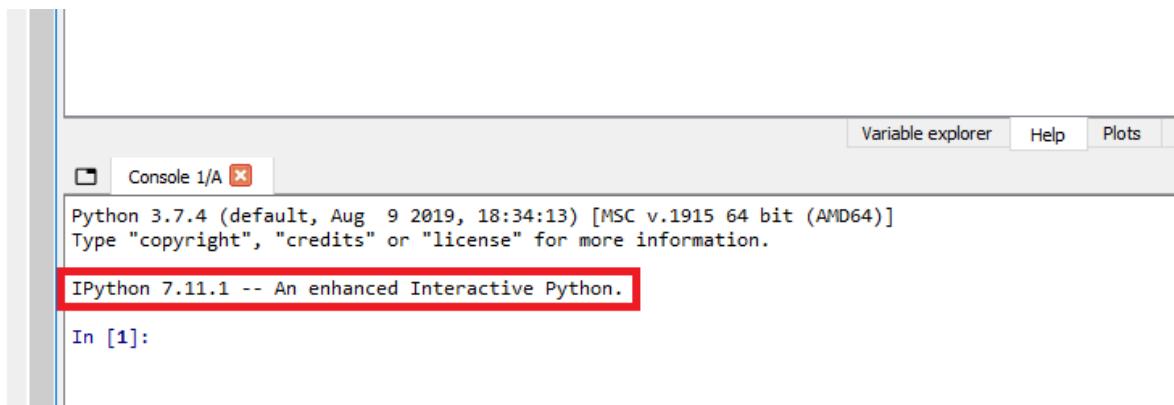
2.1.2 Install the Lake Shore Python driver

To install the driver simply open a terminal (command prompt) window and type:

```
pip install lakeshore
```

2.1.3 Installing the driver through Spyder

The driver can be installed directly within the Spyder IDE. To do this, first ensure that Spyder is using version 7.3 or greater of IPython. You can check the version by looking at the console window when opening the IDE, see image below for details:



If the version is below 7.3, open an anaconda prompt and type:

```
pip install IPython --upgrade
```

Back in the Spyder console, type:

```
pip install lakeshore
```

The driver is now installed! Now take a look through *Getting Started* to begin communicating with your instrument(s).

Updating the driver

If you have an old version of the Lake Shore Python driver and want to upgrade to the newest version, open a terminal (command prompt) window and type:

```
pip install --upgrade lakeshore
```

2.2 Getting Started

This page assumes that you have completed *Installation* of the Lake Shore Python driver. It is intended to give a basic understanding of how to use the driver to communicate with an instrument.

2.2.1 A simple example

```
from lakeshore import Model155

my_instrument = Model155()
print(my_instrument.query('*IDN?'))
```

2.2.2 Making Connections

Connecting to a specific instrument

The driver attempts to connect to an instrument when an instrument class object is created. When no arguments are passed, the driver will connect to the first available instrument.

If multiple instruments are connected you may target a specific device in one of two ways. Either by specifying the serial number of the instrument:

```
from lakeshore import Teslameter

my_specific_instrument = Teslameter(serial_number='LSA12AB')
```

or the COM port it is connected to:

```
from lakeshore import FastHall

my_specific_instrument = FastHall(com_port='COM7')
```

Some instruments have configurable baud rates. For these instruments the baud rate is a required parameter:

```
from lakeshore import Model372

my_instrument = Model372(9600)
```

Connecting over TCP

By default, the driver will try to connect to the instrument over a serial USB connection.

Connecting to an instrument over TCP requires knowledge of its IP address. The IP address can typically be found through the front panel interface and used like so:

```
from lakeshore import Model155

my_network_connected_instrument = Model155(ip_address='10.1.2.34')
```

2.2.3 Commands and queries

All Lake Shore instruments supported by the Python driver have `command()` and `query()` methods.

The Python driver makes it simple to send the instrument a command or query:

```
from lakeshore import Model155

my_instrument = Model155()

my_instrument.command('SOURCE:FUNCTION:MODE SIN')
print(my_instrument.query('SOURCE:FUNCTION:MODE?'))
```

2.2.4 SCPI commands and queries

Grouping multiple commands & queries

Instruments that support SCPI allow for multiple commands or queries, simply separate them with commas:

```
from lakeshore import Teslometer

my_instrument = Teslometer()
# Set the averaging window to 250 ms, get the DC field measurement, and get the
# temperature measurement.
response = my_instrument.query('SENSE:AVERAGE:COUNT 25', 'FETCH:DC?', 'FETCH:TEMP?')
```

The commands will execute in the order they are listed. The response to each query will be delimited by semicolons in the order they are listed.

Checking for SCPI errors

For instruments that support SCPI, both the command and query methods will automatically check the SCPI error queue for invalid commands or parameters. If you would like to disable error checking, such as in situations where you need a faster response rate, it can be turned off with an optional argument:

```
from lakeshore import Teslometer

my_instrument = Teslometer()
z_axis_measurement = my_instrument.query('FETCH:DC? Z', check_errors=False)
```

2.3 Advanced

2.3.1 Thread Safety

While an instrument can only be instantiated once, all methods on an instrument are thread safe. Multiple python threads with a reference to an instrument may simultaneously call the instrument methods.

2.3.2 Logging

For debugging your application, it can be useful to see a log of transactions with the instrument(s). All commands/queries are logged to a logger named *lakeshore*.

For example, you can print this log to stdout like this:

```
import logging
import sys

lake_shore_log = logging.getLogger('lakeshore')
lake_shore_log.addHandler(logging.StreamHandler(stream=sys.stdout))
lake_shore_log.setLevel(logging.INFO)
```

2.3.3 Status Registers

Every XIP instrument implements the SCPI status system which is derived from the status system called out in chapter 11 of the IEEE 488.2 standard. This system is useful for efficiently monitoring the state of an instrument. However the system is also fairly complex. Refer to the instrument manual available on [our website](#) before diving in.

Reading a register

Each register and register mask can be read by a corresponding *get* function. The function returns an object that contains the state of each register bit. For example:

```
from lakeshore import Teslameter

my_instrument = Teslameter()
print(dut.get_operation_events())
```

will return the following:

```
{'no_probe': False, 'overload': False, 'ranging': False, 'ramp_done': False, 'no_data_on_
↳breakout_adapter': False}
```

Modifying a register mask

Modifying a register mask can be done in one of two ways. Either by using the *modify* functions like so:

```
from lakeshore import PrecisionSource

my_instrument = PrecisionSource()
my_instrument.modify_standard_event_register_mask('command_error', True)
```

or by using the *set* functions to define the states of all bits in the register:

```
from lakeshore import PrecisionSource, PrecisionSourceQuestionableRegister

my_instrument = PrecisionSource()
register_mask = PrecisionSourceQuestionableRegister(voltage_source_in_current_limit=True,
                                                    current_source_in_voltage_compliance=True,
                                                    calibration_error=False,
                                                    inter_processor_communication_error=False)

my_instrument.set_questionable_event_enable_mask(register_mask)
```

2.3.4 Instrument initialization options

Keep communication errors on initialization

By default the error flags or queue will be reset upon connecting to an instrument. If this behavior is not desired use the following optional parameter like so:

```
from lakeshore import Teslameter

my_instrument = Teslameter(clear_errors_on_init=False)
```

2.3.5 Alternative instrument connections

Alternative connections to the instrument can be used by the driver. Create your connection and pass it into the constructor for your instrument. The connection must have write, query, and clear methods for this to work.

GPIB connection with PyVISA

```
import pyvisa
from lakeshore import SSMSystem

# Set up GPIB connection to instrument
rm = pyvisa.ResourceManager()
M81_connection = rm.open_resource('GPIB0::12::INSTR')

# Pass connection to instrument constructor
my_M81 = SSMSystem(connection=M81_connection)
print(my_M81.query('*IDN?'))
```

2.3.6 Python 2 compatibility

Python 2 is no longer supported by the python software foundation. The most recent version of this driver that is fully compatible with python 2 is version 1.4. If your application requires the use of the python 2 interpreter, use version 1.4.

2.4 Supported Instruments

The following Lake Shore Instruments are presently supported by the python driver.

2.4.1 Magnetism Instruments

F41 & F71 Teslameters

The Lake Shore single-axis (F41) and multi-axis (F71) Teslameters provide highly accurate field strength measurements.

More information about the instrument can be found [on our website](#) including the manual which has a list of all SCPI commands and queries.

Example Scripts

Below are a few example scripts for the Teslameters that use the Lake Shore Python driver.

Streaming F41/F71 teslameter data to a CSV file

```
from lakeshore import Teslameter

# Connect to the first available Teslameter over USB
my_teslameter = Teslameter()

# Configure the instrument to be in DC field mode and give it a moment to settle
my_teslameter.command('SENSE:MODE DC')

# Query the probe serial number
probe_serial_number = my_teslameter.query('PROBE:SNUMBER?')

# Query the probe temperature
probe_temperature = my_teslameter.query('FETCH:TEMPERATURE?')

# Create a file to write data into.
file = open("teslameter_data.csv", "w")

# Write header info including the instrument serial number, probe serial number, and
# temperature.
file.write('Header Information\n')
file.write('Instrument serial number:,' + my_teslameter.serial_number + '\n')
file.write('Probe serial number:,' + probe_serial_number + '\n')
file.write('Probe temperature:,' + probe_temperature + '\n\n')

# Collect 10 seconds of 10 ms data points and write them to the csv file
my_teslameter.log_buffered_data_to_file(10, 10, file)

# Close the file so that it can be used by the function
file.close()
```

Instrument class methods

```
class lakeshore.teslameter.Teslameter(serial_number=None, com_port=None, baud_rate=115200,
                                       flow_control=True, timeout=2.0, ip_address=None,
                                       tcp_port=7777, **kwargs)
```

A class object representing a Lake Shore F41 or F71 Teslameter.

```
stream_buffered_data(length_of_time_in_seconds, sample_rate_in_ms)
```

Yield a generator object for the buffered field data.

Useful for getting the data in real time when doing a lengthy acquisition.

Args:

length_of_time_in_seconds (float):

The period of time over which to stream the data.

sample_rate_in_ms (int):

The averaging window (sampling period) of the instrument.

Returns:

A generator object that returns the data as datapoint tuples.

get_buffered_data_points(*length_of_time_in_seconds, sample_rate_in_ms*)

Returns a list of named tuples that contain the buffered data.

Args:

length_of_time_in_seconds (float):

The period of time over which to collect the data.

sample_rate_in_ms (int):

The averaging window (sampling period) of the instrument.

Returns:

The data as a list of datapoint tuples.

log_buffered_data_to_file(*length_of_time_in_seconds, sample_rate_in_ms, file*)

Creates or appends a CSV file with the buffered data and excel-friendly timestamps.

Args:

length_of_time_in_seconds (float):

The period of time over which to collect the data.

sample_rate_in_ms (int):

The averaging window (sampling period) of the instrument.

file (file_object):

Field measurement data will be written to this file object in a CSV format.

get_dc_field()

Returns the DC field reading.

get_dc_field_xyz()

Returns the DC field reading.

get_rms_field()

Returns the RMS field reading.

get_rms_field_xyz()

Returns the RMS field reading.

get_frequency()

Returns the field frequency reading.

get_max_min()

Returns the maximum and minimum field readings respectively.

get_max_min_peaks()

Returns the maximum and minimum peak field readings respectively.

reset_max_min()

Resets the maximum and minimum field readings to the present field reading.

get_temperature()

Returns the temperature reading.

get_probe_information()

Returns a dictionary of probe data.

get_relative_field()

Returns the relative field value.

tare_relative_field()

Copies the current field reading to the relative baseline value.

get_relative_field_baseline()

Returns the relative field baseline value.

set_relative_field_baseline(*baseline_field*)

Configures the relative baseline value.

Args:

baseline_field (float):

A field units value that will act as the zero field for the relative measurement.

configure_field_measurement_setup(*mode='DC', autorange=True, expected_field=None, averaging_samples=20*)

Configures the field measurement settings.

Args:

mode (str):

Modes are as follows: “DC”, “AC” (0.1 - 500 Hz), and “HIFR” (50 Hz - 100 kHz).

autorange (bool):

Chooses whether the instrument automatically selects the best range for the measured value.

expected_field (float):

When autorange is False, the expected_field is the largest field expected to be measured. It sets the lowest instrument field range capable of measuring the value.

averaging_samples (int):

The number of field samples to average. Each sample is 10 milliseconds of field information.

get_field_measurement_setup()

Returns the mode, autoranging state, range, and number of averaging samples as a dictionary.

configure_temperature_compensation(*temperature_source='PROBE', manual_temperature=None*)

Configures how temperature compensation is applied to the field readings.

Args:

temperature_source (str):

Determines where the temperature measurement is drawn from. Options are: “PROBE” (Compensation is based on measurement of a thermistor in the probe), “MTEM” (Compensation is based on a manual temperature value provided by the user), “NONE” (Temperature compensation is not applied).

manual_temperature (float):

Sets the temperature provided by the user for MTEMP (manual temperature) source in Celsius.

get_temperature_compensation_source()

Returns the source of temperature measurement for field compensation.

get_temperature_compensation_manual_temp()

Returns the manual temperature setting value in Celsius.

configure_field_units(*units='TESLA'*)

Configures the field measurement units of the instrument.

Args:

units (str):

A unit of magnetic field. Options are: “TESLA”, or “GAUSS”.

get_field_units()

Returns the magnetic field units of the instrument.

configure_field_control_limits(*voltage_limit=10.0, slew_rate_limit=10.0*)

Configures the limits of the field control output.

Args:**voltage_limit (float):**

The maximum voltage permitted at the field control output. Must be between 0 and 10V.

slew_rate_limit (float):

The maximum rate of change of the field control output voltage in volts per second.

get_field_control_limits()

Returns the field control output voltage limit and slew rate limit.

configure_field_control_output_mode(*mode='CLLOOP', output_enabled=True*)

Configure the field control mode and state.

Args:**mode (str):**

Determines whether the field control is in open or closed loop mode. Options: “CLLOOP” (closed loop control), or “OPLOOP” (open loop control, voltage output).

output_enabled (bool):

Turn the field control voltage output on or off.

get_field_control_output_mode()

Returns the mode and state of the field control output.

configure_field_control_pid(*gain=None, integral=None, ramp_rate=None*)

Configures the closed loop control parameters of the field control output.

Args:**gain (float):**

Also known as P or Proportional in PID control. This controls how strongly the control output reacts to the present error. Note that the integral value is multiplied by the gain value.

integral (float):

Also known as I or Integral in PID control. This controls how strongly the control output reacts to the past error *history*.

ramp_rate (float):

This value controls how quickly the present field set-point will transition to a new set-point. The ramp rate is configured in field units per second.

get_field_control_pid()

Returns the gain, integral, and ramp rate.

set_field_control_setpoint(*setpoint*)

Sets the field control setpoint value in field units.

get_field_control_setpoint()

Returns the field control setpoint.

set_field_control_open_loop_voltage(*output_voltage*)

Sets the field control open loop voltage.

get_field_control_open_loop_voltage()

Returns the field control open loop voltage.

set_analog_output(*analog_output_mode*)

Configures what signal is provided by the analog output BNC.

set_analog_output_signal(*analog_output_mode*)

Configures what signal is provided by the analog output BNC.

Args:

analog_output_mode (str):

Configures what signal is provided by the analog output BNC. Options: “OFF” (output off), “XRAW” (raw amplified X channel Hall voltage), “YRAW” (raw amplified Y channel Hall voltage), “ZRAW” (raw amplified Z channel Hall voltage), “XCOR” (Corrected X channel field measurement), “YCOR” (Corrected Y channel field measurement), “ZCOR” (Corrected Z channel field measurement), or “MCOR” (Corrected magnitude field measurement)

configure_corrected_analog_output_scaling(*scale_factor*, *baseline*)

Configures the conversion between field reading and analog output voltage.

Args:

scale_factor (float):

Scale factor in volts per unit field.

baseline (float):

The field value at which the analog output voltage is zero.

get_corrected_analog_output_scaling()

Returns the scale factor and baseline of the corrected analog out.

get_analog_output()

Returns what signal is being provided by the analog output.

get_analog_output_signal()

Returns what signal is being provided by the analog output.

enable_high_frequency_filters()

Applies filtering to the high frequency RMS measurements.

disable_high_frequency_filters()

Turns off filtering of the high frequency mode measurements.

set_frequency_filter_type(*filter_type*)

Configures which filter is applied to the high frequency measurements.

Args:

filter_type (str):

Options: “LPASS” (low pass filter), “HPASS” (high pass filter), or “BPASS” (band pass filter).

get_frequency_filter_type()

Returns the type of filter that is or will be applied to the high frequency measurements.

get_low_pass_filter_cutoff()

Returns the cutoff frequency setting of the low pass filter.

set_low_pass_filter_cutoff(*cutoff_frequency*)

Configures the low pass filter cutoff.

Args:

cutoff_frequency (float):

Options: NONE, F10, F30, F100, F300, F1000, F3000, or F10000 F10 = 10 Hz, etc.

get_high_pass_filter_cutoff()

Returns the cutoff frequency setting of the low pass filter.

set_high_pass_filter_cutoff(*cutoff_frequency*)

Configures the high pass filter cutoff.

Args:

cutoff_frequency (float):

Options: NONE, F10, F30, F100, F300, F1000, F3000, or F10000 F10 = 10 Hz, etc.

get_band_pass_filter_center()

Returns the center of the band pass filter.

set_band_pass_filter_center(*center_frequency*)

Configures the band pass filter parameters.

Args:

center_frequency (float):

The frequency at which the gain of the filter is 1.

enable_qualifier()

Enables the qualifier.

disable_qualifier()

Disables the qualifier.

is_qualifier_condition_met()

Returns whether the qualifier condition is met.

enable_qualifier_latching()

Enables the qualifier condition latching.

disable_qualifier_latching()

Disables the qualifier condition latching.

get_qualifier_latching_setting()

Returns whether the qualifier latches.

set_qualifier_latching_setting(*latching*)

Sets whether the qualifier latches.

Args:

latching (bool):

Determines whether the qualifier latches.

reset_qualifier_latch()

Resets the condition status of the qualifier.

get_qualifier_configuration()

Returns the threshold mode and field threshold values.

configure_qualifier(*mode, lower_field, upper_field=None*)

Sets the threshold condition of the qualifier.

Args:

mode (str):

The type of threshold condition used by the qualifier. Options: “OVER”, “UNDER”, “BETWEEN”, “OUTSIDE”, “ABS BETWEEN”, or “ABS OUTSIDE”.

lower_field (float):

The lower field value threshold used by the qualifier.

upper_field (float):

The upper field value threshold used by the qualifier. Not used for OVER or UNDER.

command(**commands, check_errors=True*)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):

Any number of SCPI commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(*ip_address, tcp_port, timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

Resets all system information such as settings, wi-fi connections, date and time, etc.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_service_request_enable_mask()

Returns the named bits of the status byte service request enable register.

This register determines which bits propagate to the master summary status bit.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

get_standard_events()

Returns the names of the standard event register bits and their values.

get_status_byte()

Returns named bits of the status byte register and their values.

modify_operation_register_mask(*bit_name, value*)

Gets the operation condition register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_questionable_register_mask(*bit_name, value*)

Gets the questionable condition register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_service_request_mask(*bit_name, value*)

Gets the service request enable mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_standard_event_register_mask(*bit_name, value*)

Gets the standard event register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

query(*queries, check_errors=True)

Sends an SCPI query or multiple queries to the instrument and return the response(s).

Args:

queries (str):

Any number of SCPI queries or commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

The instrument query response as a string.

reset_measurement_settings()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_operation_event_enable_mask(register_mask)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:

register_mask ([Instrument]OperationRegister):

An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(register_mask)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

set_service_request_enable_mask(register_mask)

Configures values of the service request enable register bits.

This register determines which bits propagate to the master summary bit.

Args:

register_mask (StatusByteRegister):

A StatusByteRegister class object with all bits configured true or false.

set_standard_event_enable_mask(register_mask)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Status register classes

This page outlines the objects and classes used to interact with registers in the Teslameter driver.

class lakeshore.teslameter.**TeslameterOperationRegister**(*no_probe, overload, ranging, ramp_done, no_data_on_breakout_adapter*)

Class object representing the operation status register.

class lakeshore.teslameter.**TeslameterQuestionableRegister**(*x_axis_sensor_error, y_axis_sensor_error, z_axis_sensor_error, probe_eeprom_read_error, temperature_compensation_error, invalid_probe, field_control_slew_rate_limit, field_control_at_voltage_limit, calibration_error, heartbeat_error*)

Class object representing the questionable status register.

class lakeshore.teslameter.**StatusByteRegister**(*error_available, questionable_summary, message_available_summary, event_status_summary, master_summary, operation_summary*)

Class object representing the status byte register.

__init__(*error_available, questionable_summary, message_available_summary, event_status_summary, master_summary, operation_summary*)

class lakeshore.teslameter.**StandardEventRegister**(*operation_complete, query_error, device_specific_error, execution_error, command_error, power_on*)

Class object representing the standard event register.

__init__(*operation_complete, query_error, device_specific_error, execution_error, command_error, power_on*)

Model 425 Gaussmeter

The Model 425 Gaussmeter provides field strength measurements.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Instrument methods

```
class lakeshore.model_425.Model1425(serial_number=None, com_port=None, baud_rate=57600,  
    data_bits=7, stop_bits=1, parity='O', flow_control=False,  
    handshaking=False, timeout=2.0, ip_address=None, tcp_port=7777,  
    **kwargs)
```

A class object representing the Lake Shore Model 425 Gaussmeter.

```
command(command_string)
```

Send a command to the instrument.

Args:

command_string (str):

A serial command.

```
connect_tcp(ip_address, tcp_port, timeout)
```

Establishes a TCP connection with the instrument on the specified IP address.

```
connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None,  
    parity=None, timeout=None, handshaking=None, flow_control=None)
```

Establish a serial USB connection.

```
disconnect_tcp()
```

Disconnect the TCP connection.

```
disconnect_usb()
```

Disconnect the USB connection.

```
query(query_string)
```

Send a query to the instrument and return the response.

Args:

query_string (str):

A serial query ending in a question mark.

Returns:

The instrument query response as a string.

```
write(command_string)
```

Alias of `command`. Send a command to the instrument.

Args:

command_string (str):

A serial command.

2.4.2 Magnet System Power Supplies

Model 643 & 648 Electromagnet Power Supplies

The Lake Shore Model 643 (2.5 kW) and Model 648 (9.1 kW) Electromagnet Power Supplies provide a highly accurate linear, bipolar current source.

More information about the [Model 643](#) and [Model 648](#) can be found on our [website](#) including the manual which has a list of all commands and queries.

Example Scripts

Below are a few example scripts for the Electromagnet Power Supplies that use the Lake Shore Python driver.

Outputting current and measuring voltage on a Model 643

```
from lakeshore import Model643
from time import sleep

# Connect to 643 over USB
my_power_supply = Model643()

# Setup current and ramp rate limits
my_power_supply.set_limits(20.000, 3.000)

# Start outputting current from the power supply
my_power_supply.set_current(10.000)

# Wait for power supply to ramp up to specified output current
sleep(10)

# Record real-time output voltage at power supply terminals
measured_voltage = my_power_supply.get_measured_voltage()

print(f"Measured Voltage: {measured_voltage}")

# Set current output back to 0 Amps
my_power_supply.set_current(0)
```

Setting up magnet water and outputting current from a Model 648

```
from lakeshore import Model648

# Connect to 648 over USB
my_power_supply = Model648()

# Setup current and ramp rate limits
my_power_supply.set_limits(120.000, 40.000)

# Set magnet water to Auto
my_power_supply.set_magnet_water(2)

# Setup ramp rate for current output
my_power_supply.set_ramp_rate(25.000)

# Start outputting current from the power supply
my_power_supply.set_current(95.000)
```

Querying the hardware error status register from a Model 648

```

from lakeshore import Model648

# Connect to 648 over USB
my_power_supply = Model648()

# Query for hardware errors register mask
register_mask = my_power_supply.get_hardware_error_enable_mask()

# Mask all hardware errors (Masked bits do not affect the parent register)
register_mask.from_integer(0)

# Unmask desired error bits
register_mask.output_over_current = True
register_mask.output_over_voltage = True
register_mask.temperature_fault = True

# Set hardware errors mask
my_power_supply.set_hardware_error_enable_mask(register_mask)

# Query for hardware errors (Ignores the bits masked above)
hardware_error = my_power_supply.get_status_byte().hardware_errors_summary
if hardware_error:
    print(my_power_supply.get_hardware_error_condition())
else:
    print("No hardware errors!")

```

Instrument class methods

```

class lakeshore.em_power_supply.ElectromagnetPowerSupply(serial_number=None, com_port=None,
                                                         baud_rate=57600, data_bits=7,
                                                         stop_bits=1, parity='O',
                                                         flow_control=False, handshaking=False,
                                                         timeout=2.0, ip_address=None,
                                                         tcp_port=7777, **kwargs)

```

Class object representing a Lake Shore Model 643 or 648 electromagnet power supply.

```

class EMPowerSupplyServiceRequestEnableRegister(operational_errors_summary,
                                                hardware_errors_summary, message_available,
                                                event_summary, operation_summary)

```

Class object representing the service request enable register LSB to MSB.

```

classmethod from_integer(integer_representation)

```

Creates the register object from an integer representation value.

```

to_integer()

```

Translates the register object to an integer representation value.

```

class EMPowerSupplyStatusByteRegister(operational_errors_summary, hardware_errors_summary,
                                       message_available, event_summary, service_request,
                                       operation_summary)

```

Class object representing the status byte register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

class EPowerSupplyStandardEventStatusRegister(*operation_complete, query_error, execution_error, command_error, power_on*)

Class object representing the standard event status register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

class EPowerSupplyOperationEventRegister(*compliance, ramp_done, power_limit*)

Class object representing the operation event register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

class EPowerSupplyHardwareErrorsRegister(*output_control_failure, dac_processor_not_responding, output_over_current, output_over_voltage, temperature_fault, output_stage_protect*)

Class object representing the hardware errors register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

class EPowerSupplyOperationalErrorsRegister(*calibration_error, external_current_program_error, temperature_high, low_line_voltage, high_line_voltage, magnet_flow_switch_fault, power_supply_flow_switch_fault, remote_enable_fault*)

Class object representing the operational errors register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

command(*command_string, check_errors=True*)

Send a command to the instrument.

Args:

command_string (str): A serial command
check_errors (bool): Specifies if command errors should be checked. Optional parameter. Defaults to True.

query(*query_string*, *check_errors=True*)

Send a query to the instrument and return the response

Args:

query_string (str): A serial query ending in a question mark.

check_errors (bool): Specifies if command errors should be checked. Optional parameter. Defaults to True.

Returns:

str: The instrument query response as a string.

set_limits(*max_current*, *max_ramp_rate*)

Sets the upper setting limits for output current, and output current ramp rate.

This is a software limit that will limit the setting of the values. Only limits internal setting of the current.

Args:

max_current (float): The maximum output current setting allowed. The Model 643 bounds are 0.0000 - 70.1000

A. The Model 648 bounds are 0.0000 - 135.1000 A.

max_ramp_rate (float): The maximum output current ramp rate setting allowed (0.0001 - 50.000 A/s).

get_limits()

Returns the upper setting limits for output current, and output current ramp rate.

This is a software limit that limits the setting of the values. Only limits the internal setting of the current.

Returns:

list[float]: List of [output_current, output_current_ramp_rate].

set_ramp_rate(*ramp_rate*)

Sets the output current ramp rate.

This value will be used in both the positive and negative directions. Setting value is limited by `set_limit`.

Args:

ramp_rate (float): The rate at which the current will ramp when a new output current setting is entered
(0.0001 - 50.000 A/s).

get_ramp_rate()

Returns the output current ramp rate.

This value is used in both the positive and negative directions.

Returns:

float: The rate at which the current will ramp when a new output current setting is entered.

set_ramp_segment(*segment, current, ramp_rate*)

Sets the current and ramp rate of one of the ramp segments.

Args:

segment (int): Specifies the ramp segment to be modified (1 - 5). *current* (float): Specifies the upper output current setting that will use this segment. *ramp_rate* (float): Specifies the rate at which the current will ramp. (0.0001 - 50.000 A/s).

get_ramp_segment(*segment*)

Returns the current and ramp rate of a specific ramp segment.

Args:

segment (int): Specifies the ramp segment to be modified (1 - 5).

Returns:

list[float]: List of current and ramp_rate settings. [current, ramp_rate].

set_ramp_segments_enable(*state*)

Specifies if ramp segments are to be used.

Ramp segments are used to change the output current ramp rate based on the output current.

Args:

state (bool): The state of the ramp segments enable. 0=Disabled and 1=Enabled.

get_ramp_segments_enable()

Returns if ramp segments are to be used.

Ramp segments are used to change the output current ramp rate based on the output current.

Returns:

bool: Whether ramp segments are enabled. 0=Disabled and 1=Enabled.

set_current(*current*)

Sets the output current setting.

The setting value is limited by *set_limit*.

Args:

current (float): The output current value that the output will ramp to at the present ramp rate.

The Model

643 bounds are 0.0000 - +/-70.1000 A. The Model 648 bounds are 0.0000 - +/-135.1000 A.

get_current()

Returns the output current setting.

Returns:

float: The output current value that the output will ramp to at the present ramp rate.

get_measured_current()

Returns actual measured output current.

Returns:

float: Measured output current.

get_measured_voltage()

Returns actual output voltage measured at the power supply terminals.

Returns:

float: Measured output voltage.

stop_output_current_ramp()

Stops the output current ramp.

Stops within 2 s of sending command. TO restart the ramp, use the set_current method to set a new output current set-point.

set_internal_water(mode)

Configures the internal water mode.

Args:

mode (int): Internal water mode (0, 1, 2, or 3). 0 = Manual-Off, 1 = Manual-On, 2 = Auto, 3 = Disabled.

get_internal_water()

Returns the internal water mode.

Returns:

int: Internal water mode. 0 = Manual-Off, 1 = Manual-On, 2 = Auto, 3 = Disabled.

set_magnet_water(mode)

Configures the magnet water mode.

Args:

mode (int): Magnet water mode. (0, 1, 2, or 3). 0 = Manual-Off, 1 = Manual-On, 2 = Auto, 3 = Disabled.

get_magnet_water()

Returns the magnet water mode.

Returns:

int: Magnet water mode. 0 = Manual-Off, 1 = Manual-On, 2 = Auto, 3 = Disabled.

set_display_brightness(brightness_level)

Specifies display brightness.

Args:

brightness_level (int): The display brightness. 0=25%, 1=50%, 2=75%, 3=100%.

get_display_brightness()

Returns display brightness.

Returns:

int: The display brightness. 0=25%, 1=50%, 2=75%, 3=100%.

set_front_panel_lock(lock_state, code)

Sets the lock status of the front panel keypad.

Args:

lock_state (int): The lock state to be set (0, 1, or 2). 0=unlock, 1=lock, and 2=lock limits. code (int): Keypad lock code required to make changes to the lock state of the front panel.

get_front_panel_status()

Returns what lock state the front panel keypad is in.

Returns:

int: The state of the front panel keypad lock (0, 1, or 2). 0=unlock, 1=lock, 2=lock limits.

get_front_panel_lock_code()

Returns the lock code for the front panel.

Returns:

int: Front panel lock code.

set_programming_mode(mode)

Sets the current programming mode of the instrument.

Args:

mode (int): Programming mode (0, 1, or 2). 0=Internal, 1=External, 2=Sum.

get_programming_mode()

Returns the current programming mode of the instrument.

Returns:

int: Programming mode. 0=Internal, 1=External, 2=Sum.

set_ieee_488(terminator, eoi_enable, address)

Configures the IEEE-488 interface.

Args:

terminator(int): the terminator. 0=<CR><LF>, 1=<LF><CR>, 2=<LF>, 3=no terminator (must have EOI enabled). eoi_enable(int): Sets EOI (End of Interrupt) mode. 0=Enabled, 1=Disabled. address (int): Specifies IEEE address. 1 - 30(0 and 31 are reserved).

get_iee_488()

Returns IEEE-488 interface configuration.

Returns:

list[int]: [terminator, eoi_enable, address]

terminator(int): the terminator. 0=<CR><LF>, 1=<LF><CR>, 2=<LF>, 3=no terminator (must have EOI enabled). eoi_enable(int): Sets EOI (End of Interrupt) mode. 0=Enabled, 1=Disabled. address (int): Specifies IEEE address. 1 - 30(0 and 31 are reserved).

set_ieee_interface_mode(mode)

Sets the interface mode of the instrument.

Args:

mode (int): Interface mode. 0, 1 or 2. 0=local, 1=remote, and 2=remote with local lockout.

get_ieee_interface_mode()

Returns the interface mode of the instrument.

Returns:

int: Interface mode of the instrument. 0, 1 or 2. 0=local, 1=remote, and 2=remote with local lockout.

set_factory_defaults()

Sets all configuration values to factory defaults and resets the instrument.

The instrument must be at zero amps for this command to work.

reset_instrument()

Sets the controller parameters to power-up settings.

Use the set_factory_defaults command to set factory-defaults.

clear_interface()

Clears the event registers in all register groups. Also clears the error queue.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation event Register, and terminates all pending operations. Clears the interface, but not the instrument. The related instrument command is `reset_instrument`.

get_self_test()

Returns result of instrument self test completed at power up.

Returns:

bool: True means errors found, and False means no errors found.

set_service_request_enable_mask(*register_mask*)

Configures the Service Request Enable Register.

The Service Request Enable Register determines which summary bits of the Status Byte may set bit 6 (RQS/MSS) of the Status Byte to generate a Service Request.

Args:

`register_mask` (`ElectromagnetPowerSupply.EMPowerSupplyServiceRequestEnableRegister`): Register configuration object.

get_service_request_enable_mask()

Returns Service Request Enable Register configuration.

The Service Request Enable Register determines which summary bits of the Status Byte may set bit 6 (RQS/MSS) of the Status Byte to generate a Service Request.

Returns:

`ElectromagnetPowerSupply.EMPowerSupplyServiceRequestEnableRegister`: Register configuration object.

get_status_byte()

Returns state of the Status Byte Register.

The Status Byte register, typically referred to as the Status Byte, is a non-latching, read-only register that contains all the summary bits from the register sets.

Returns:

`ElectromagnetPowerSupply.EMPowerSupplyStatusByteRegister`: Register state object.

set_standard_event_status_enable_mask(*register_mask*)

Configures Standard Event Status Enable Register group.

The Standard Event Status Enable Register determines which bits in the Standard Event Status Register will set the summary bit in the Status Byte (bit 5).

Args:

`register_mask` (`ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister`):
Register configuration
object.

get_standard_event_status_enable_mask()

Returns Standard Event Status Enable Register configuration.

The Standard Event Status Enable Register determines which bits in the Standard Event Status Register will set the summary bit in the Status Byte (bit 5).

Returns:

ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister: Register configuration object.

get_standard_event_status_event()

Returns state of the Standard Event Status Register.

Bits in this register correspond to various system events and latch when the event occurs. When an event bit is set, subsequent events corresponding to that bit are ignored. Set bits remain latched until the register is reset by this query or `clear_interface`.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister: Register state object.

set_operation_event_enable_mask(*register_mask*)

Configures the Operational Event Enable Register.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding operational status bit in the Operational Status Register. This determines which status bits can set the corresponding summary bit in the Status Byte Register.

Args:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Register configuration object.

get_operation_event_enable_mask()

Returns Operational Event Enable Register configuration.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding operational status bit in the Operational Status Register. This determines which status bits can set the corresponding summary bit in the Status Byte Register.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Register configuration object.

get_operation_event_condition()

Returns the real-time state of the operation event bits.

The condition register constantly monitors the instrument status. The data bits are real-time and are not latched or buffered. The register is read-only.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Object with each of the register bits' status.

get_operation_event_event()

Returns the latched state of the operation event bits.

Bits in the event register correspond to various system events and latch when the event occurs. When an event bit is set, subsequent events corresponding to that bit are ignored.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Object with each of the register bits' status.

set_hardware_error_enable_mask(*register_mask*)

Sets which hardware error bits will set the summary bit in the Status Byte Register.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding error bits in the Error Status Register. This determines which status bits can set the corresponding summary bits in the Status Byte Register.

Args:

register_mask (ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister):
Register mask configuration
object.

get_hardware_error_enable_mask()

Returns which hardware error bits will set the summary bit in the Status Byte Register.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding error bits in the Error Status Register. This determines which status bits can set the corresponding summary bits in the Status Byte Register.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Register mask configuration object.

get_hardware_error_condition()

Returns the real-time state of the hardware error bits.

The condition register constantly monitors the instrument status. The data bits are real-time and are not latched or buffered. The register is read-only.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Object with each of the register bits' status.

get_hardware_error_event()

Returns the latched state of the hardware error bits.

Bits in the event register correspond to various system events and latch when the event occurs. When an event bit is set, subsequent events corresponding to that bit are ignored.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Object with each of the register bits' status.

set_operational_error_enable_mask(*register_mask*)

Sets which operational error bits will set the summary bit in the Status Byte Register.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding error bits in the Error Status Register. This determines which status bits can set the corresponding summary bits in the Status Byte Register.

Args:

register_mask (**ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister**):
Register mask configuration
 object.

get_operational_error_enable_mask()

Returns which operational error bits will set the summary bit in the Status Byte Register.

Each bit has a bit weighting and represents the enable/disable mask of the corresponding error bits in the Error Status Register. This determines which status bits can set the corresponding summary bits in the Status Byte Register.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Register mask configuration object.

get_operational_error_condition()

Returns the real-time state of the operational error bits.

The condition register constantly monitors the instrument status. The data bits are real-time and are not latched or buffered. The register is read-only.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationalErrorsRegister: Object with each of the register bits' status.

get_operational_error_event()

Returns the latched state of the operational error bits.

Bits in the event register correspond to various system events and latch when the event occurs. When an event bit is set, subsequent events corresponding to that bit are ignored.

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationalErrorsRegister: Object with each of the register bits' status.

connect_tcp(*ip_address, tcp_port, timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (**str**):
 A serial command.

2.4.3 Materials Characterization

M91 Fast Hall Controller

The Lake Shore M91 Fast Hall controller makes high speed Hall measurements for materials characterization.

More information about the instrument can be found [on our website](#) including the manual which has a list of all SCPI commands and queries.

Example Scripts

Below are a few example scripts for the M91 Fast Hall Controller that use the Lake Shore Python driver.

Fast Hall Full Sample Analysis

```
from lakeshore import FastHall
from lakeshore import ContactCheckOptimizedParameters, ResistivityLinkParameters,
↪FastHallLinkParameters

# Connect to the first available FastHall over USB
my_fast_hall = FastHall()

# Create an optimized contact check settings object that limits the max current to 1 mA
ccheck_settings = ContactCheckOptimizedParameters(max_current=1e-3)

# Define the DC magnetic field strength of the measurement
magnetic_field_strength = 0.1

# Create a resistivity and FastHall measurement linked settings objects
resistivity_settings = ResistivityLinkParameters()
fasthall_settings = FastHallLinkParameters(magnetic_field_strength)

# Run the optimized contact check to automatically determine the best parameters for the
↪sample
ccheck_results = my_fast_hall.run_complete_contact_check_optimized(ccheck_settings)

# Run a resistivity measurement linking the parameters determined by the contact check
resistivity_results = my_fast_hall.run_complete_resistivity_link(resistivity_settings)

# Prompt the user to insert the sample into the defined field
input("Insert sample into " + str(magnetic_field_strength) + " Tesla field")

# Run the FastHall measurement linking information from the resistivity and contact
↪check measurements
fasthall_results = my_fast_hall.run_complete_fasthall_link(fasthall_settings)

# Dump the data into a text file
results_file = open("sample_analysis.txt", "w")
results_file.write("Contact check results:\n" + str(ccheck_results))
results_file.write("\nResistivity results:\n" + str(resistivity_results))
results_file.write("\nFastHall results:\n" + str(fasthall_results))
```

Fast Hall Record Contact Check Data

```

from lakeshore import FastHall, ContactCheckManualParameters

# Connect to the first available FastHall over USB
my_fast_hall = FastHall()

# Create a contact check parameters object with desired settings to run a manual Contact_
↳ Check measurement
ccheck_settings = ContactCheckManualParameters(excitation_type='CURRENT',
                                                excitation_start_value=-10e-6,
                                                excitation_end_value=10e-6,
                                                compliance_limit=1.5,
                                                number_of_points=20)

# Create a file to write data into
file = open("fasthall_data.csv", "w")

# Write header info including the type of test and specific measurements being taken
file.write('Contact Check of Van der Pauw Sample with Varying Current Excitation Ranges\n'
↳ ')
file.write('Contact Pair 1-2\n')
file.write(' , Offset, Slope, R Squared, R Squared Passed, In Compliance, Voltage_
↳ Overload, Current Overload\n')

# Create a list of current excitation ranges
excitation_ranges = [10e-3, 10e-4, 10e-5, 10e-6]

# Run a separate Contact Check measurement and collect results for each range in the_
↳ list of excitation ranges
for range_value in excitation_ranges:

    # Set the value of the excitation range
    ccheck_settings.excitation_range = range_value

    # Write the specific excitation range that is being used
    file.write('Excitation Range: ' + str(range_value) + '\n')

    # Run a complete contact check measurement using the settings with the updated_
↳ excitation range
    results = my_fast_hall.run_complete_contact_check_manual(ccheck_settings, sample_
↳ type="VDP")

    # Collect the measurement results that correlate to the first contact pair (Contact_
↳ Pair 1-2)
    contact_pair_results = results.get('ContactPairIVResults')
    pair_one_results = contact_pair_results[0]

    # Obtain contact pair result values, then convert them into a list and then a string
    logged_keys = ['Offset', 'Slope', 'RSquared', 'RSquaredPass', 'InCompliance',
↳ 'VoltageOverload', 'CurrentOverload']
    logged_values = [pair_one_results[key] for key in logged_keys]
    logged_string = ','.join(str(value) for value in logged_values)

```

(continues on next page)

(continued from previous page)

```
# Write the result values to the file
file.write(',') + logged_string + '\n')

# Close the file so that it can be used by the function
file.close()
```

Instrument class methods

```
class lakeshore.fast_hall_controller.FastHall(serial_number=None, com_port=None,
                                              baud_rate=921600, flow_control=True, timeout=2.0,
                                              ip_address=None, tcp_port=7777, **kwargs)
```

A class object representing a Lake Shore M91 Fast Hall controller.

get_contact_check_running_status()

Indicates if the contact check measurement is running.

get_fasthall_running_status()

Indicates if the FastHall measurement is running.

get_four_wire_running_status()

Indicates if the four wire measurement is running.

get_resistivity_running_status()

Indicates if the resistivity measurement is running.

get_dc_hall_running_status()

Indicates if the DC Hall measurement is running.

get_dc_hall_waiting_status()

Indicates if the DC hall measurement is running.

continue_dc_hall()

Continues the DC hall measurement if it's in a waiting state.

start_contact_check_vdp_optimized(settings)

Automatically determines excitation value and ranges. Then runs contact check on all 4 pairs.

Args:

settings(ContactCheckOptimizedParameters):

start_contact_check_vdp(settings)

Performs a contact check measurement on contact pairs 1-2, 2-3, 3-4, and 4-1.

Args:

settings(ContactCheckManualParameters):

start_contact_check_hbar(settings)

Performs a contact check measurement on contact pairs 5-6, 5-1, 5-2, 5-3, 5-4, and 6-1

Args:

settings(ContactCheckManualParameters):

start_fasthall_vdp(settings)

Performs a FastHall measurement.

Args:

settings (FastHallManualParameters):

start_fasthall_link_vdp(*settings*)

Starts a FastHall measurement with provided link parameters.

Performs a FastHall measurement that uses the last run contact check measurement's excitation type, compliance limit, blanking time, excitation range, and the largest absolute value of the start and end excitation values along with the last run resistivity measurement's resistivity average and sample thickness.

Args:

settings (FastHallLinkParameters):

start_four_wire(*settings*)

Performs a Four wire measurement.

Excitation is sourced from Contact Point 1 to Contact Point 2. Voltage is measured/sensed between contact point 3 and contact point 4.

Args:

settings(FourWireParameters):

start_dc_hall_vdp(*settings*)

Performs a DC hall measurement for a Hall Bar sample.

Args:

settings(DCHallParameters):

start_dc_hall_hbar(*settings*)

Performs a DC hall measurement for a Hall Bar sample.

Args:

settings(DCHallParameters):

start_resistivity_vdp(*settings*)

Performs a resistivity measurement on a Van der Pauw sample.

Args:

settings(ResistivityManualParameters):

start_resistivity_link_vdp(*settings*)

Performs a resistivity measurement with provided link settings.

Performs a resistivity measurement that uses the last run contact check measurement's excitation type, compliance limit, blanking time, excitation range, and the largest absolute value of the start and end excitation values.

Args:

settings(ResistivityLinkParameters):

start_resistivity_hbar(*settings*)

Performs a resistivity measurement on a hall bar sample.

Args:

settings(ResistivityManualParameters):

get_contact_check_setup_results()

Returns an object representing the setup results of the last run Contact Check measurement.

get_contact_check_measurement_results()

Returns a dictionary representing the results of the last run Contact Check measurement.

get_fasthall_setup_results()

Returns an object representing the setup results of the last run FastHall measurement.

get_fasthall_measurement_results()

Returns a dictionary representing the results of the last run FastHall measurement.

get_four_wire_setup_results()

Returns an object representing the setup results of the last run Four Wire measurement.

get_four_wire_measurement_results()

Returns a dictionary representing the results of the last run Four Wire measurement.

get_dc_hall_setup_results()

Returns a dictionary representing the setup results of the last run Hall measurement.

get_dc_hall_measurement_results()

Returns a dictionary representing the results of the last run Hall measurement.

get_resistivity_setup_results()

Returns an object representing the setup results of the last run Resistivity measurement.

get_resistivity_measurement_results()

Returns a dictionary representing the results of the last run Resistivity measurement.

run_complete_contact_check_optimized(*settings*)

Performs a contact check measurement and then returns the corresponding measurement results.

Args:

settings(ContactCheckOptimizedParameters):

Returns:

The measurement results as a dictionary.

run_complete_contact_check_manual(*settings*, *sample_type*)

Performs a manual contact check measurement and then returns the corresponding measurement results.

Args:

settings (ContactCheckManualParameters):

Object with settings for FastHall link setup.

sample_type (str):

Indicates sample type. Options: "VDP" (Van der Pauw sample), or "HBAR" (Hall Bar sample).

Returns:

The measurement results as a dictionary.

run_complete_fasthall_link(*settings*)

Performs a FastHall Link measurement and then returns the corresponding measurement results.

Args:

settings(FastHallLinkParameters):

Object with settings for FastHall link setup.

Returns:

The measurement results as a dictionary.

run_complete_fasthall_manual(*settings*)

Performs a manual FastHall measurement and then returns the corresponding measurement results.

Args:

settings(FastHallManualParameters):

Object with settings for FastHall link setup.

Returns:

The measurement results as a dictionary.

run_complete_four_wire(*settings*)

Performs a Four Wire measurement and then returns the corresponding measurement results.

Args:

settings(FourWireParameters):

Returns:

The measurement results as a dictionary.

run_complete_dc_hall(*settings*, *sample_type*)

Performs a DC Hall measurement and then returns the corresponding measurement results.

Args:**settings**(DCHallParameters):

Object with settings for FastHall link setup.

sample_type(str):

Indicates sample type. Options: “VDP” (Van der Pauw sample), or “HBAR” (Hall Bar sample).

Returns:

The measurement results as a dictionary.

run_complete_resistivity_link(*settings*)

Performs a resistivity link measurement and then returns the corresponding measurement results.

Args:

settings(ResistivityLinkParameters):

Returns:

The measurement results as a dictionary.

run_complete_resistivity_manual(*settings*, *sample_type*)

Performs a manual resistivity measurement and then returns the corresponding measurement results.

Args:**settings**(ResistivityManualParameters):

Object with settings for manual resistivity setup.

sample_type(str):

Indicates sample type. Options are: “VDP” (Van der Pauw sample), or “HBAR” (Hall Bar sample).

Returns:

The measurement results as a dictionary.

reset_contact_check_measurement()

Resets the measurement to a not run state, canceling any running measurement.

reset_fasthall_measurement()

Resets the measurement to a not run state, canceling any running measurement.

reset_four_wire_measurement()

Resets the measurement to a not run state, canceling any running measurement.

reset_dc_hall_measurement()

Resets the measurement to a not run state, canceling any running measurement.

reset_resistivity_measurement()

Resets the measurement to a not run state, canceling any running measurement.

command(*commands, check_errors=True)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):

Any number of SCPI commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

Resets all system information such as settings, wi-fi connections, date and time, etc.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_service_request_enable_mask()

Returns the named bits of the status byte service request enable register.

This register determines which bits propagate to the master summary status bit.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

get_standard_events()

Returns the names of the standard event register bits and their values.

get_status_byte()

Returns named bits of the status byte register and their values.

modify_operation_register_mask(*bit_name, value*)

Gets the operation condition register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_questionable_register_mask(*bit_name, value*)

Gets the questionable condition register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_service_request_mask(*bit_name, value*)

Gets the service request enable mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_standard_event_register_mask(*bit_name, value*)

Gets the standard event register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

query(queries, check_errors=True*)**

Sends an SCPI query or multiple queries to the instrument and return the response(s).

Args:**queries (str):**

Any number of SCPI queries or commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.
Optional Parameter.

Returns:

The instrument query response as a string.

reset_measurement_settings()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:

register_mask ([Instrument]OperationRegister):

An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

set_service_request_enable_mask(*register_mask*)

Configures values of the service request enable register bits.

This register determines which bits propagate to the master summary bit.

Args:

register_mask (StatusByteRegister):

A StatusByteRegister class object with all bits configured true or false.

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Settings classes

This page outlines the classes and objects used to interact with various settings and methods of the M91.

```
class lakeshore.fast_hall_controller.FastHallOperationRegister(settling, ranging,
measurement_complete,
waiting_for_trigger,
field_control_ramping,
field_measurement_enabled,
transient)
```

Class object representing the operation status register.

```
class lakeshore.fast_hall_controller.FastHallQuestionableRegister(source_in_compliance_or_at_current_limit,
field_control_slew_rate_limit,
field_control_at_voltage_limit,
current_measurement_overload,
voltage_measurement_overload,
invalid_probe,
invalid_calibration,
inter_processor_communication_error,
field_measurement_communication_error,
probe_eeprom_read_error,
r2_less_than_minimum_allowable)
```

Class object representing the questionable status register.

```
class lakeshore.fast_hall_controller.ContactCheckManualParameters(excitation_type,
excitation_start_value,
excitation_end_value,
compliance_limit,
number_of_points,
excitation_range='AUTO',
measurement_range='AUTO',
min_r_squared=0.9999,
blanking_time=0.002)
```

Class object representing parameters used for manual Contact Check run methods.

Special-members

```
__init__
```

```
class lakeshore.fast_hall_controller.ContactCheckOptimizedParameters(max_current=0.1,
max_voltage=10,
number_of_points=11,
min_r_squared=0.9999)
```

Class object representing parameters used for optimized Contact Check run methods.

Special-members

```
__init__
```

```
class lakeshore.fast_hall_controller.FastHallManualParameters(excitation_type, excitation_value,
                                                         user_defined_field,
                                                         compliance_limit,
                                                         excitation_range='AUTO', excita-
                                                         tion_measurement_range='AUTO',
                                                         measurement_range='AUTO',
                                                         max_samples=100,
                                                         resistivity="NaN",
                                                         blanking_time=0.002,
                                                         averaging_samples=60,
                                                         sample_thickness=0,
                                                         min_hall_voltage_snr=30)
```

Class object representing parameters used for running manual FastHall measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.FastHallLinkParameters(user_defined_field,
                                                         measurement_range='AUTO',
                                                         max_samples=100,
                                                         min_hall_voltage_snr=30,
                                                         averaging_samples=60,
                                                         sample_thickness='DEF')
```

Class object representing parameters used for running FastHall Link measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.FourWireParameters(contact_point1, contact_point2,
                                                         contact_point3, contact_point4,
                                                         excitation_type, excitation_value,
                                                         compliance_limit,
                                                         excitation_range='AUTO',
                                                         measurement_range='AUTO',
                                                         excitation_measurement_range='AUTO',
                                                         blanking_time=0.002, max_samples=100,
                                                         min_snr=30, excitation_reversal=True)
```

Class object representing parameters used for running Four Wire measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.DCHallParameters(excitation_type, excitation_value,
                                                         compliance_limit, averaging_samples,
                                                         user_defined_field, excitation_range='AUTO',
                                                         excitation_measurement_range='AUTO',
                                                         measurement_range='AUTO',
                                                         with_field_reversal=True, resistivity="NaN",
                                                         blanking_time=0.002, sample_thickness=0)
```

Class object representing parameters used for running DC Hall measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.ResistivityManualParameters(excitation_type,
                                                             excitation_value,
                                                             compliance_limit,
                                                             excitation_range='AUTO',
                                                             excita-
                                                             tion_measurement_range='AUTO',
                                                             measurement_range='AUTO',
                                                             max_samples=100,
                                                             blanking_time=0.002,
                                                             sample_thickness=0,
                                                             min_snr=30, **kwargs)
```

Class object representing parameters used for running manual Resistivity measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.ResistivityLinkParameters(measurement_range='AUTO',
                                                             sample_thickness=0, min_snr=30,
                                                             max_samples=100)
```

Class object representing parameters used for running manual Resistivity measurements.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.StatusByteRegister(error_available, questionable_summary,
                                                         message_available_summary,
                                                         event_status_summary, master_summary,
                                                         operation_summary)
```

Class object representing the status byte register.

Special-members

`__init__`

```
class lakeshore.fast_hall_controller.StandardEventRegister(operation_complete, query_error,
                                                           device_specific_error, execution_error,
                                                           command_error, power_on)
```

Class object representing the standard event register.

Special-members

`__init__`

M81 Synchronous Source Measure System

Instrument methods are grouped into three classes: SSMSystem, SourceModule, and MeasureModule

Example Scripts

Below are a few example scripts for the M81 SSM system that use the Lake Shore Python driver.

Making a lock in measurement of resistance using a BCS-10 and VM-10

```
from lakeshore import SSMSystem
from time import sleep
from math import sqrt
```

(continues on next page)

(continued from previous page)

```

# Connect to instrument via USB
my_M81 = SSMSystem()

# Instantiate source and measure modules
balanced_current_source = my_M81.get_source_module(1)
voltage_measure = my_M81.get_measure_module(1)

# Set the source frequency to 13.7 Hz
balanced_current_source.set_frequency(13.7)

# Set the source current peak amplitude to 1 mA
balanced_current_source.set_current_amplitude(0.001)

# Set the voltage measure module to reference the source 1 module with a 100 ms time_
↳constant
voltage_measure.setup_lock_in_measurement('S1', 0.1)

# Enable the source output
balanced_current_source.enable()

# Wait for 15 time constants before taking a measurement
sleep(1.5)
lock_in_magnitude = voltage_measure.get_lock_in_r()

# Get the amplitude of the current source
peak_current = balanced_current_source.get_current_amplitude()

# Calculate the resistance
resistance = lock_in_magnitude * sqrt(2) / peak_current
print("Resistance: {} ohm".format(resistance))

```

List settings profiles and restore a profile

```

from lakeshore import SSMSystem

# Connect to instrument via USB
my_M81 = SSMSystem()

# Print a list of saved settings profiles
print(my_M81.settings_profiles.get_list())

# Check that a specific profile can be applied with the present modules
profile_name = "Transistor IV sweep"
if my_M81.settings_profiles.get_valid_for_restore(profile_name):
    my_M81.settings_profiles.restore(profile_name)
else:
    print("The connected modules don't match the profile. Please check the profile and_
↳try again.")

```

Stream data

```

from lakeshore import SSMSystem

# Connect to instrument via USB
my_M81 = SSMSystem()

# stream 5,000 samples of synchronized data at 1,000 samples per second
streamed_data = my_M81.get_data(1000, 5000,
                                (my_M81.DataSourceMnemonic.RELATIVE_TIME, 1),
                                (my_M81.DataSourceMnemonic.SOURCE_OFFSET, 1),
                                (my_M81.DataSourceMnemonic.MEASURE_X, 1),
                                (my_M81.DataSourceMnemonic.MEASURE_Y, 1),
                                (my_M81.DataSourceMnemonic.MEASURE_DC, 2))

# format the data and print to the console
for point in streamed_data:
    print(f'Time in seconds: {point[0]}')
    print(f'Source 1 offset: {point[1]}')
    print(f'Measure 1 in-phase indication: {point[2]}')
    print(f'Measure 1 out-of-phase indication: {point[3]}')
    print(f'Measure 2 DC indication: {point[4]}\n')

```

Sweep a BCS-10 from 0 mA to 100 mA

```

from lakeshore import SSMSystem

# Connect to instrument via USB
my_ssm = SSMSystem()

# Set up a BCS-10 in channel S1 in SC shape with a manual range of 100 mA
s1_bcs = my_ssm.get_source_module(1)

s1_bcs.set_shape('DC')
s1_bcs.configure_current_range(False, max_level=.1)

# Configure the sweep settings to sweep 0 mA to 100 mA with a dwell time of 1 ms
sweep_configuration = SSMSystem.SourceSweepSettings(sweep_type=my_ssm.SourceSweepType.
    ←CURRENT_AMPLITUDE,
                                                    start=0.0,
                                                    stop=0.1,
                                                    points=1000,
                                                    dwell=.001,
                                                    direction=my_ssm.SourceSweepSettings.
    ←Direction.UP,
                                                    round_trip=False)
s1_bcs.set_sweep_configuration(sweep_configuration)

# stream 1,000 samples of synchronized data at 1,000 samples per second and
    ←simultaneously start the sweep
s1_bcs.enable()
stream_data = my_ssm.get_data(1000, 1000,

```

(continues on next page)

(continued from previous page)

```

[my_ssm.DataSourceMnemonic.SOURCE_AMPLITUDE, 1],
[my_ssm.DataSourceMnemonic.MEASURE_DC, 1])

print(stream_data)

```

Ramp, Sweep, and collect data with a VS-10

```

from lakeshore import SSMSystem
from time import sleep

# Connect to instrument via USB and reset settings
my_ssm = SSMSystem()
my_ssm.reset_measurement_settings()

# Set up a VS-10 in channel S1
s1_vs_10 = my_ssm.get_source_module(1)
s1_vs_10.enable()

# Configure the VS-10 to smoothly ramp to 2 volts at 0.25 volts per second
s1_vs_10.set_voltage_ramp_configuration(2, slew_rate=0.25)
# Initiate the ramp
my_ssm.initiate_sweeps()

print('Ramping to start')
# Poll the instrument waiting for the ramp to finish
while s1_vs_10.get_source_sweep_state():
    sleep(0.1)

print('Running data sweep')
# Automatically do a sweep on S1 with data collected from M1
swept_data = s1_vs_10.do_dc_sweep_step_and_measure(2, 5)

print(swept_data)

```

SSMS instrument methods

```

class lakeshore.ssm_system.SSMSystem(serial_number=None, com_port=None, baud_rate=921600,
                                     flow_control=True, timeout=5.0, ip_address=None, tcp_port=7777,
                                     **kwargs)

```

Class for interaction with the M81 instrument.

load_modules()

Loads all unloaded modules. Connected modules must be loaded before they can be used.

get_num_measure_channels()

Returns the number of measure channels supported by the instrument.

get_num_source_channels()

Returns the number of source channels supported by the instrument

get_source_module(port_number)

Returns a SourceModule instance for the given port number.

get_source_pod(*port_number*)

Alias of `get_source_module`.

get_source_module_by_name(*module_name*)

Return the `SourceModule` instance that matches the specified name.

get_measure_module(*port_number*)

Returns a `MeasureModule` instance for the given port number.

get_measure_pod(*port_number*)

Alias of `get_measure_module`.

get_measure_module_by_name(*module_name*)

Return the `MeasureModule` instance that matches the specified name.

get_multiple(**data_sources*)

This function is deprecated. Use `fetch_multiple()` instead.

Deprecated since version 1.5.4: Use `fetch_multiple` instead.

get_multiple_min_max_values(**data_sources*)

Gets a synchronized minimum and maximum value for each specified data source.

Args:

data_sources (str, int):

Pairs of (DATASOURCE_MNEMONIC, CHANNEL_INDEX).

stream_data(*rate*, *num_points*, **data_sources*)

Generator object to stream data from the instrument.

Args:

rate (int):

Desired transfer rate in points/sec.

num_points (int):

Number of points to return. None to stream indefinitely.

data_sources (SSMSYSTEMDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Yields:

A single row of stream data as a tuple.

get_data(*rate*, *num_points*, **data_sources*)

Like `stream_data`, but returns a list.

Args:

rate (int):

Desired transfer rate in points/sec.

num_points (int):

Number of points to return.

data_sources (SSMSYSTEMDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:

All available stream data as a list of tuples.

log_data_to_csv_file(*rate, num_points, file, *data_sources, **kwargs*)

Like stream_data, but logs directly to a CSV file.

Args:

rate (int):

Desired transfer rate in points/sec.

file (IO):

File to log to.

num_points (int):

Number of points to log.

data_sources (SSMSystemDataSourceMnemonic or str, int):

Pairs of (DATA_SOURCE, CHANNEL_INDEX).

write_header (bool):

If true, a header row is written with column names.

get_ref_in_edge()

Returns the active edge of the reference input. 'RISing' or 'FALLing'.

set_ref_in_edge(*edge*)

Sets the active edge of the reference input.

Args:

edge (str):

The new active edge ('RISing', or 'FALLing').

get_ref_out_source()

Returns the channel used for the reference output. 'S1', 'S2', or 'S3'.

set_ref_out_source(*ref_out_source*)

Sets the channel used for the reference output.

Args:

ref_out_source (str):

The new reference out source ('S1', 'S2', or 'S3').

get_ref_out_state()

Returns the enable state of reference out.

set_ref_out_state(*ref_out_state*)

Sets the enable state of reference out.

Args:

ref_out_state (bool):

The new reference out state (True to enable reference out, False to disable reference out).

enable_ref_out()

Sets the enable state of reference out to True.

disable_ref_out()

Sets the enable state of reference out to False.

configure_ref_out(*ref_out_source, ref_out_state=True*)

Configure the reference output.

Args:

ref_out_source (str):

The new reference out source ('S1', 'S2', or 'S3').

ref_out_state (bool):

The new reference out state (True to enable reference out, False to disable reference out).

get_mon_out_mode()

Returns the channel used for the monitor output. 'M1', 'M2', 'M3', or 'MANUAL'.

set_mon_out_mode(mon_out_source)

Sets the channel used for the monitor output.

Args:**mon_out_source (str):**

The new monitor out source ('M1', 'M2', 'M3', or 'MANUAL').

get_mon_out_state()

Returns the enable state of monitor out.

set_mon_out_state(mon_out_state)

Sets the enable state of monitor out.

Args:**mon_out_state (bool):**

The new monitor out state (True to enable monitor out, False to disable monitor out).

enable_mon_out()

Sets the enable state of monitor out to True.

disable_mon_out()

Sets the enable state of monitor out to False.

configure_mon_out(mon_out_source, mon_out_state=True)

Configure the monitor output.

Args:**mon_out_source (str):**

The new monitor out source ('M1', 'M2', or 'M3').

mon_out_state (bool):

The new monitor out state (True to enable monitor out, False to disable monitor out).

get_mon_out_scale()

Returns the monitor out scaling factor of the configured module.

get_head_cal_datetime()

Returns the date and time of the head calibration.

get_head_cal_temperature()

Returns the temperature of the head calibration.

get_head_self_cal_status()

Returns the status of the last head self calibration.

get_head_self_cal_datetime()

Returns the datetime of the last head self calibration.

get_head_self_cal_temperature()

Returns the temperature of the last head self calibration.

run_head_self_calibration()

Runs a self calibration for the head.

reset_head_self_calibration()

“Restore the factory self calibration.

set_mon_out_manual_level(*manual_level*)

Set the manual level of monitor out when the mode is MANUAL.

Args:

manual_level (float):

The new monitor out manual level.

get_mon_out_manual_level()

Returns the manual level of monitor out.

configure_mon_out_manual_mode(*manual_level*, *mon_out_state=True*)

Configures the monitor output for manual mode.

Args:

manual_level (float):

The new monitor out manual level.

mon_out_state (bool):

The new monitor out state (True to enable monitor out, False to disable monitor out).

get_line_frequency()

Returns the line frequency in Hz.

get_detected_line_frequency()

Returns the detected line frequency in Hz.

get_line_frequency_detection_error_status()

Returns the line frequency detection error status. True if the frequency is out of bounds.

fetch_multiple(data_sources*)**

Gets a list of the latest values corresponding to the input data sources, and returns them quickly.

Args:

data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:

Tuple of values corresponding to the given data sources.

read_multiple(data_sources*)**

Initiates measurement of new values corresponding to the input data sources.

Returns values after the measurement is complete.

Args:

data_sources (SSMSystemReadDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:

Tuple of values corresponding to the given data sources.

initiate_sweeps()

Initiates sweeps across all channels.

abort_sweeps()

Aborts in progress sweeps across all channels.

reset_sweeps()

Resets the source sweep settings for the specified module to their power on defaults.

class DataSourceMnemonic(*values)

Enumeration of M81 data source mnemonics.

class ExcitationType(*values)

Class object representing the possible excitation types of a source module.

class ReadDataSourceMnemonic(*values)

Enumeration of M81 read data source mnemonics.

class ReferenceModule(*values)

Class object representing the available resistance source modules

class ResistanceExcitationType(*values)

Class object representing the possible excitation types that create a valid resistance configuration.

class ResistanceMode(*values)

Class object representing the possible resistance modes.

class SourceSweepSettings(sweep_type, start, stop, points, dwell, direction=Direction.UP, round_trip=False, spacing=SweepSpacing.LINEAR, initial_delay=0.0, blanking=0.0)

Class to configure a parameter sweep on a source module.

class Direction(*values)

Class object representing the possible directions for sweeping.

class SweepSpacing(*values)

Class object representing the possible types of sweep spacing.

class SourceSweepType(*values)

Class representing the available sweep types for a source module.

command(*commands, check_errors=True)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):

Any number of SCPI commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

Resets all system information such as settings, wi-fi connections, date and time, etc.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_service_request_enable_mask()

Returns the named bits of the status byte service request enable register.

This register determines which bits propagate to the master summary status bit.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

get_standard_events()

Returns the names of the standard event register bits and their values.

get_status_byte()

Returns named bits of the status byte register and their values.

modify_operation_register_mask(*bit_name, value*)

Gets the operation condition register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_questionable_register_mask(*bit_name, value*)

Gets the questionable condition register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_service_request_mask(*bit_name, value*)

Gets the service request enable mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_standard_event_register_mask(*bit_name, value*)

Gets the standard event register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

query(**queries, check_errors=True*)

Sends an SCPI query or multiple queries to the instrument and return the response(s).

Args:**queries (str):**

Any number of SCPI queries or commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

The instrument query response as a string.

reset_measurement_settings()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:**register_mask ([Instrument]OperationRegister):**

An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

set_service_request_enable_mask(*register_mask*)

Configures values of the service request enable register bits.

This register determines which bits propagate to the master summary bit.

Args:

register_mask (StatusByteRegister):

A StatusByteRegister class object with all bits configured true or false.

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Source Module methods

class lakeshore.ssm_source_module.**SourceModule**(*module_number, device*)

Class for interaction with a specific source channel of the M81 instrument.

get_multiple(**data_sources*)

This function is deprecated. Use `fetch_multiple()` instead.

fetch_multiple(**data_sources*)

Gets a list of the latest values corresponding to the input data sources for this module.

Args:

data_sources (SSMSystemDataSourceMnemonic or str):

Variable length list of data sources.

Returns:

Tuple of values corresponding to the given data sources for this module.

get_name()

Returns the user-settable name of the module.

set_name(*new_name*)

Set the name of the module.

get_notes()

Returns the user-settable notes of the module.

set_notes(*new_note*)

Set the notes of the module.

get_model()

Returns the model of the module (i.e. BCS-10).

get_serial()

Returns the serial number of the module (i.e. LSA1234).

get_hw_version()

Returns the hardware version of the module.

get_self_cal_status()

Returns the status of the last self calibration of the module.

run_self_cal()

Run a self calibration for the module.

reset_self_cal()

Restore factory self calibration for the module.

get_enable_state()

Returns the output state of the module.

set_enable_state(*state*)

Sets the enable state of the module.

Args:**state (bool):**

The new output state.

enable()

Sets the enable state of the module to True.

disable()

Sets the enable state of the module to False.

get_source_function()

Returns the excitation function of the module. 'CURRENT' or 'VOLTAGE'.

set_source_function(*excitation_mode*)

Sets the excitation function of the module.

Args:**excitation_mode (SSMSystem.ExcitationType):**

The new excitation mode ('CURRENT' or 'VOLTAGE').

go_to_current_mode()

Sets the excitation mode of the module to 'CURRENT'.

go_to_voltage_mode()

Sets the excitation mode of the module to 'VOLTAGE'.

get_shape()

Returns the signal shape of the module. 'DC' or 'SINUSOID'.

set_shape(*shape*)

Sets the signal shape of the module.

Args:

shape (str):

The new signal shape ('DC', 'SINUSOID', 'TRIANGLE', 'SQUARE').

get_frequency()

Returns the excitation frequency of the module.

set_frequency(*frequency*)

Sets the excitation frequency of the module.

Args:

frequency (float):

The new excitation frequency.

get_sync_state()

Returns whether the source channel synchronization feature is engaged

If true, this channel will ignore its own frequency, and instead track the frequency of the synchronization source. If false, this channel will generate its own frequency.

get_sync_source()

Returns the channel used for frequency synchronization.

get_sync_phase_shift()

Returns the phase shift applied between the synchronization source and this channel.

configure_sync(*source, phase_shift, enable_sync=True*)

Configure the source channel synchronization feature.

Args:

source (str):

The channel used for synchronization ('S1', 'S2', 'S3', or 'RIN'). This module will follow the frequency set for the specified channel if sync is enabled.

phase_shift (float):

The phase shift applied between the synchronization source and this channel in degrees.

enable_sync (bool):

If true, this channel will ignore its own frequency, and instead track the frequency of the synchronization source. If false, this channel will generate its own frequency.

get_duty()

Returns the duty cycle of the module.

set_duty(*duty*)

Sets the duty cycle of the module.

Args:

duty (float):

The new duty cycle.

get_coupling()

Returns the coupling type of the module. 'AC' or 'DC'.

set_coupling(*coupling*)

Sets the coupling of the module.

Args:

coupling (str):

The new coupling type ('AC', or 'DC').

use_ac_coupling()

Sets the coupling type of the module to 'AC'.

use_dc_coupling()

Sets the coupling type of the module to 'DC'.

set_automatic_coupling(*enable: bool*)

Enable or disable automatic coupling for the module.

Args:

enable (bool): True to enable automatic coupling, False to disable.

get_automatic_coupling() → bool

Query the automatic coupling state of the module.

Returns:

bool: True if automatic coupling is enabled, False otherwise.

get_guard_state()

Returns the guard state of the module.

set_guard_state(*guard_state*)

Sets the guard state of the module.

Args:

guard_state (bool):

The new guard state (True to enable guards, False to disable guards).

enable_guards()

Sets the guard state of the module to True.

disable_guards()

Sets the guard state of the module to False.

get_cmf_source()

Returns the Common Mode Feedback (CMF) source. 'INTERNAL', or 'EXTERNAL'.

set_cmr_source(*cmr_source*)

This function is deprecated. Use `set_cmf_source()` instead.

Deprecated since version 1.9.0: Use `set_cmf_source` instead.

get_cmr_source()

This function is deprecated. Use `get_cmf_source()` instead.

Deprecated since version 1.9.0: Use `get_cmf_source` instead.

set_cmf_source(*cmf_source*)

Sets the Common Mode Feedback (CMF) source.

Args:

cmf_source (str):

The new CMF source ('INTERNAL', or 'EXTERNAL').

get_cmf_state()

Returns the Common Mode Feedback (CMF) state of the module.

get_cmr_state()

This function is deprecated. Use `get_cmf_state()` instead.

Deprecated since version 1.9.0: Use `get_cmf_state` instead.

set_cmf_state(*cmf_state*)

Sets the Common Mode Feedback (CMF) state of the module.

Args:

cmf_state (bool):

The new CMF state (True to enable CMF, False to disable CMF).

set_cmr_state(*cmr_state*)

This function is deprecated. Use `set_cmf_state()` instead.

Deprecated since version 1.9.0: Use `set_cmf_state` instead.

enable_cmf()

Sets the CMF state of the module to True.

enable_cmr()

This function is deprecated. Use `enable_cmf()` instead.

Deprecated since version 1.9.0: Use `enable_cmf` instead.

disable_cmf()

Sets the CMF state of the module to False.

disable_cmr()

This function is deprecated. Use `disable_cmf()` instead.

Deprecated since version 1.9.0: Use `disable_cmf` instead.

configure_cmf(*cmf_source*, *cmf_state=True*)

Configure Common Mode Feedback (CMF).

Args:

cmf_source (str):

The new CMF source ('INTERNAL', or 'EXTERNAL').

cmf_state (bool):

The new CMF state (True to enable CMF, False to disable CMF).

configure_cmr(*cmr_source*, *cmr_state=True*)

This function is deprecated. Use `configure_cmf()` instead.

Deprecated since version 1.9.0: Use `configure_cmf` instead.

get_current_range()

Returns the present current range of the module in Amps.

get_i_range()

Returns the present current range of the module in Amps

Deprecated since version 1.5.4: Use `get_current_range` instead.

get_current_ac_range()

Returns the present AC current range of the module in Amps.

get_i_ac_range()

Returns the present AC current range of the module in Amps

Deprecated since version 1.5.4: Use `get_current_ac_range` instead.

get_current_dc_range()

Returns the present DC current range of the module in Amps.

get_i_dc_range()

Returns the present DC current range of the module in Amps

Deprecated since version 1.5.4: Use `get_current_dc_range` instead.

get_current_autorange_status()

Returns whether automatic selection of the current range is enabled for this module.

get_i_autorange_status()

Returns whether automatic selection of the current range is enabled for this module

Deprecated since version 1.5.4: Use `get_current_autorange_status` instead.

configure_current_range(*autorange, max_level=None, max_ac_level=None, max_dc_level=None*)

Sets up current ranging for this module.

Args:**autorange (bool):**

True to enable automatic range selection. False for manual ranging.

max_level (float):

The largest current that needs to be sourced.

max_ac_level (float):

The largest AC current that needs to be sourced. Separate AC and DC ranges are only available on some modules.

max_dc_level (float):

The largest DC current that needs to be sourced. Separate AC and DC ranges are only available on some modules.

configure_i_range(*autorange, max_level=None, max_ac_level=None, max_dc_level=None*)

Sets up current ranging for this module

Args:**autorange (bool):**

True to enable automatic range selection. False for manual ranging.

max_level (float):

The largest current that needs to be sourced.

max_ac_level (float):

The largest AC current that needs to be sourced. Separate AC and DC ranges are only available on some modules.

max_dc_level (float):

The largest DC current that needs to be sourced. Separate AC and DC ranges are only available on some modules.

Deprecated since version 1.5.4: Use `configure_current_range` instead.

get_current_amplitude()

Returns the current amplitude for the module in Amps.

get_i_amplitude()

Returns the current amplitude for the module in Amps

Deprecated since version 1.5.4: Use `get_current_amplitude` instead.

set_current_amplitude(*amplitude*)

Sets the current amplitude for the module.

Args:

amplitude (float):

The new current amplitude in Amps.

set_i_amplitude(*amplitude*)

Sets the current amplitude for the module

Args:

amplitude (float):

The new current amplitude in Amps

Deprecated since version 1.5.4: Use `set_current_amplitude` instead.

get_current_offset()

Returns the current offset for the module in Amps.

get_i_offset()

Returns the current offset for the module in Amps

Deprecated since version 1.5.4: Use `get_current_offset` instead.

set_current_offset(*offset*)

Sets the current offset for the module.

Args:

offset (float):

The new current offset in Amps.

set_i_offset(*offset*)

Sets the current offset for the module

Args:

offset (float):

The new current offset in Amps

Deprecated since version 1.5.4: Use `set_current_offset` instead.

apply_dc_current(*level*, *output_enable=True*)

Apply DC current.

Args:

level (float):

DC current level in Amps.

output_enable (bool):

Turns the module output on if true; off if false.

apply_ac_current(*frequency, amplitude, offset=0.0, output_enable=True*)

Apply AC current.

Args:

frequency (float):

Excitation frequency in Hz.

amplitude (float):

Current amplitude in Amps.

offset (float):

Current offset in Amps.

output_enable (bool):

Turns the module output on if true; off if false.

get_current_limit()

Returns the current limit enforced by the module in Amps.

get_i_limit()

Returns the current limit enforced by the module in Amps

Deprecated since version 1.5.4: Use get_current_limit instead.

set_current_limit(*current_limit*)

Sets the current limit enforced by the module.

Args:

current_limit (float):

The new limit to apply in Amps.

set_i_limit(*i_limit*)

Sets the current limit enforced by the module

Args:

i_limit (float):

The new limit to apply in Amps

Deprecated since version 1.5.4: Use set_current_limit instead.

get_current_limit_status()

Returns whether the current limit circuitry is presently engaged.

This limits the current sourced by the module.

get_i_limit_status()

Returns whether the current limit circuitry is presently engaged and limiting the current sourced by the module

Deprecated since version 1.5.4: Use get_current_limit_status instead.

get_voltage_range()

Returns the present voltage range of the module in Volts.

get_voltage_ac_range()

Returns the present AC voltage range of the module in Volts.

get_voltage_dc_range()

Returns the present DC voltage range of the module in Volts.

get_voltage_autorange_status()

Returns whether automatic selection of the voltage range is enabled for this module.

configure_voltage_range(*autorange*, *max_level=None*, *max_ac_level=None*, *max_dc_level=None*)

Sets up voltage ranging for this module.

Args:

autorange (bool):

True to enable automatic range selection. False for manual ranging.

max_level (float):

The largest voltage that needs to be sourced.

max_ac_level (float):

The largest AC voltage that needs to be sourced. Separate AC and DC ranges are only available on some modules.

max_dc_level (float):

The largest DC voltage that needs to be sourced. Separate AC and DC ranges are only available on some modules.

get_voltage_amplitude()

Returns the voltage amplitude for the module in Volts.

set_voltage_amplitude(*amplitude*)

Sets the voltage amplitude for the module.

Args:

amplitude (float):

The new voltage amplitude in Volts.

get_voltage_offset()

Returns the voltage offset for the module in Volts.

set_voltage_offset(*offset*)

Sets the voltage offset for the module.

Args:

offset (float):

The new voltage offset in Volts.

apply_dc_voltage(*level*, *output_enable=True*)

Apply DC voltage.

Args:

level (float):

DC voltage level in Volts.

output_enable (bool):

Turns the module output on if true; off if false.

apply_ac_voltage(*frequency*, *amplitude*, *offset=0.0*, *output_enable=True*)

Apply AC voltage.

Args:

frequency (float):

Excitation frequency in Hz.

amplitude (float):

Voltage amplitude in Volts.

offset (float):

Voltage offset in Volts.

output_enable (bool):

Turns the module output on if true; off if false.

get_voltage_limit()

Returns the voltage limit enforced by the module in Volts.

set_voltage_limit(*voltage_limit*)

Sets the voltage limit enforced by the module.

Args:

voltage_limit (float):

The new limit to apply in Volts.

get_voltage_limit_status()

Returns whether the voltage limit circuitry is presently engaged.

This limits the voltage at the output of the module.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:

register_mask ([Instrument]OperationRegister):

An instrument specific OperationRegister class object with all bits configured true or false.

get_identify_state()

Returns the identification state for the given pod.

set_identify_state(*state*)

Configures the identification state for the given pod.

Args:

state (bool):

The desired state for the LED, 1 for identify, 0 for normal state.

get_dark_mode_state()

Returns the dark mode state for the given pod.

set_dark_mode_state(*state*)

Configures the dark mode state for the given pod.

Args:

state (bool):

The desired operation for the LED, 1 for normal mode, 0 for dark mode.

get_voltage_output_limit_high()

Returns the present voltage high output limit.

set_voltage_output_limit_high(*limit*)

Configures the high voltage output limit.

The voltage output limits are software defined limits preventing the user from entering an output which could potentially damage the module's load. When the shape is not DC, the limit is applied to the sum of the offset and amplitude. The high voltage output limit is bounded between -10 V and 10 V, and must be greater than the low voltage output limit.

Args:

limit (float):

The desired high output limit.

get_voltage_output_limit_low()

Returns the present voltage low output limit.

set_voltage_output_limit_low(*limit*)

Configures the low voltage output limit.

The voltage output limits are software defined limits preventing the user from entering an output which could potentially damage the module's load. When the shape is not DC, the limit is applied to the sum of the offset and amplitude. The low voltage output limit is bounded between -10 V and 10 V, and must be less than the high voltage output limit.

Args:**limit (float):**

The desired low voltage output limit.

get_current_output_limit_high()

Returns the present current high output limit.

set_current_output_limit_high(limit)

Configures the high current output limit.

The current output limits are software defined limits preventing the user from entering an output which could potentially damage the module's load. When the shape is not DC, the limit is applied to the sum of the offset and amplitude. The high current output limit is bounded between -10 V and 10 V, and must be greater than the low current output limit.

Args:**limit (float):**

The desired high output limit.

get_current_output_limit_low()

Returns the present current low output limit.

set_disable_on_compliance(disable_on_compliance)

Configures the module for disable on compliance.

When disable on compliance is turned on, the module will disable output when in compliance. Otherwise, the module will continue to output, even when in compliance.

Args:**disable_on_compliance (bool):**

1 for the module to disable when in compliance; 0 for the module to remain enabled, even in compliance.

get_disable_on_compliance()

Returns the present state of disable on compliance.

set_current_output_limit_low(limit)

Configures the low current output limit.

The current output limits are software defined limits preventing the user from entering an output which could potentially damage the module's load. When the shape is not DC, the limit is applied to the sum of the offset and amplitude. The low current output limit is bounded between -10 V and 10 V, and must be less than the high current output limit.

Args:**limit (float):**

The desired low current output limit.

reset_settings()

Resets the settings for the module to their power on defaults.

unload()

Unloads the module.

get_load_state()

Returns the loaded state for the module.

get_self_cal_datetime()

Returns the self calibration date and time for the module.

get_self_cal_temperature()

Returns the self calibration temperature for the module.

get_source_sweep_step_size(*sweep_type*)

Returns the step size of the source sweep for the module.

Step size is a calculated parameter derived from the relevant sweep type's span and points.

Args:

sweep_type (SourceSweepType):

The type of sweep for which to return the step size.

get_source_sweep_time()

Returns the overall runtime of the source sweep for the module in seconds.

Sweep time is a calculated parameter derived from the dwell time and number of points.

get_source_sweep_state()

Returns the state of the source sweep on the module.

set_sweep_configuration(*sweep_settings*)

Configures a source sweep for the module.

Args:

sweep_settings (SourceSweepSettings):

The configuration for a specific sweep on the module.

get_sweep_configuration(*sweep_type*)

Returns a SourceSweepSettings of the present sweep configuration for the module.

Args:

sweep_type (SourceSweepType):

The sweep type for which to return the sweep settings.

disable_all_sweeping()

Disables all source signals that support sweeping on the module.

disable_sweeping(*sweep_type*)

Disables the sweeping of the specified sweep type on the module.

Args:

sweep_type (SourceSweepType):

The type of sweep to disable.

set_voltage_ramp_configuration(*stop_amplitude*, *start_amplitude*=None, *slew_rate*=1.0, *round_trip*=False)

Sets up a parameter sweep that ramps the voltage output to the desired amplitude.

Uses the smallest possible step size.

Args:

stop_amplitude (float):

The voltage amplitude of the output when the ramp completes.

start_amplitude (float):

The voltage amplitude of the output when the ramp starts. Default is the present amplitude setting.

slew_rate (float):

The rate in volts per second to ramp the output. Default is 1 volt per second.

round_trip (bool):

Ramps to the stop amplitude then ramps back to the start amplitude when true

set_current_ramp_configuration(*stop_amplitude, start_amplitude=None, slew_rate=0.001, round_trip=False*)

Sets up a parameter sweep that ramps the current output to the desired amplitude.

Uses the smallest possible step size.

Args:

stop_amplitude (float):

The current amplitude of the output when the ramp completes.

start_amplitude (float):

The current amplitude of the output when the ramp starts. Default is the present amplitude setting.

slew_rate (float):

The rate in amps per second to ramp the output. Default is 1 mA per second.

round_trip (bool):

Ramps to the stop amplitude then ramps back to the start amplitude when true

do_dc_sweep_step_and_measure(*start_amplitude, stop_amplitude, *optional_custom_data_sources, dwell_time='AUTO', number_of_points='AUTO', settle_time=0.0, initial_delay=0.0, sweep_spacing='LINEAR', round_trip=False, include_source_amplitude=True, include_relative_time=False*)

Immediately do a DC source sweep and returns data for each point in the sweep.

Args:

start_amplitude (float):

The starting source value of the sweep in volts or amps

stop_amplitude (float):

The ending source value of the sweep in volts or amps

dwell_time (float):

Time duration of the measurement portion of each step in seconds, defaults to AUTO

number_of_points (int):

How many steps to take going from the start to the stop amplitude. AUTO chooses a decade step size that results in between 100 and 1,000 points

optional_custom_data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX). Defaults to DC on the source's matching measure channel if unspecified. If specified only the specified custom data sources will be collected. If specified the corresponding measure channel will not be configured automatically.

settle_time (float):

This is the user blanking time. How long to wait before taking a measurement after a step change. Defaults to zero.

initial_delay (float):

Time to wait before starting the sweep, defaults to zero.

sweep_spacing (str or SweepSpacing enum):

LINEAR or LOGARITHMIC sweep

round_trip (bool):

Sweeps to the stop amplitude then sweeps back to the start amplitude when true. Returns double the sweep number of points when true.

include_source_amplitude (bool):

The source amplitude is included in the returned data when true. Not relevant if using custom data sources.

include_relative_time (bool):

The time since the start of the sweep is in the returned data when true. Not relevant if using custom data sources.

get_voltage_sense_mode()

Returns the source voltage source sense for the specified module as either LOCAL or REMote

set_voltage_sense_mode(*state*)

Sets the source voltage source sense for the specified module

Args:

state (str):

Voltage source sense ('LOCAL' or 'REMote')

get_readback_dc()

Returns the DC source readback measurement

get_readback_rms()

Returns the RMS source readback measurement

get_readback_nplcycles()

Returns the source readback averaging time in number of power line cycles (NPLC) of the module

get_requested_readback_nplcycles()

Returns the requested source readback averaging time in number of power line cycles (NPLC) of the specified module

get_actual_readback_nplcycles()

Returns the actual source readback averaging time in number of power line cycles (NPLC) of the specified module

set_readback_nplcycles(*value*)

Sets the source readback averaging time in number of power line cycles (NPLC) of the module.

The NPLC value must be between 0.01 and 600.00.

Args:

value (float):

The averaging time in number of power line cycles (NPLC)

do_ac_amplitude_sweep_step_and_measure(*start_amplitude*, *stop_amplitude*, *ac_shape*, *source_frequency*, *source_offset=0*, *optional_custom_data_sources=None*, *kwargs*)**

Sweep AC amplitude at fixed frequency and offset.

Args:**start_amplitude (float):**

The starting amplitude of the sweep in volts or amps.

stop_amplitude (float):

The ending amplitude of the sweep in volts or amps.

ac_shape (str):

The AC waveform shape, e.g., 'SINUSOID', 'SQUARE', 'TRIANGLE'.

source_frequency (float):

The AC frequency in Hz.

source_offset (float):

The AC offset in volts or amps. Default is 0.

optional_custom_data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

kwargs:

Additional keyword arguments to pass to the do_ac_parameter_sweep_step_and_measure method.

Returns:

list: A list of tuples containing the measured data for each point in the sweep.

do_ac_frequency_sweep_step_and_measure(*start_frequency*, *stop_frequency*, *source_amplitude*,
ac_shape, *source_offset*=0,
optional_custom_data_sources=None, ****kwargs**)

Sweep AC frequency at fixed amplitude and offset.

Args:**start_frequency (float):**

The starting frequency of the sweep in Hz.

stop_frequency (float):

The ending frequency of the sweep in Hz.

source_amplitude (float):

The AC amplitude in volts or amps.

ac_shape (str):

The AC waveform shape, e.g., 'SINUSOID', 'SQUARE', 'TRIANGLE'.

source_offset (float):

The AC offset in volts or amps. Default is 0.

optional_custom_data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

kwargs:

Additional keyword arguments to pass to the do_ac_parameter_sweep_step_and_measure method.

Returns:

list: A list of tuples containing the measured data for each point in the sweep.

do_ac_offset_sweep_step_and_measure(*start_offset*, *stop_offset*, *ac_shape*, *source_amplitude*,
source_frequency, *optional_custom_data_sources*=None,
****kwargs**)

Sweep AC offset at fixed amplitude and frequency.

Args:

start_offset (float):

The starting offset of the sweep in volts or amps.

stop_offset (float):

The ending offset of the sweep in volts or amps.

source_amplitude (float):

The AC amplitude in volts or amps.

ac_shape (str):

The AC waveform shape, e.g., 'SINUSOID', 'SQUARE', 'TRIANGLE'.

source_frequency (float):

The AC frequency in Hz.

optional_custom_data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

kwargs:

Additional keyword arguments to pass to the do_ac_parameter_sweep_step_and_measure method.

Returns:

list: A list of tuples containing the measured data for each point in the sweep.

do_ac_parameter_sweep_step_and_measure(*ac_sweep_parameter, start_value, stop_value, *optional_custom_data_sources, ac_shape='SINUSOID', source_amplitude=None, source_frequency=None, source_offset=None, dwell_time='AUTO', number_of_points='AUTO', settle_time=0.0, initial_delay=0.0, sweep_spacing='LINEAR', round_trip=False, include_source_sweep_parameter=True, include_relative_time=False*)

Immediately runs an AC parameter(amplitude, frequency, offset) source sweep and returns data for each point in the sweep.

Args:

ac_sweep_parameter (str):

The AC parameter to sweep, 'OFFSET', 'FREQUENCY', 'AMPLITUDE'.

ac_shape (str):

The AC waveform shape, e.g., 'SINUSOID', 'SQUARE', 'TRIANGLE'.

start_value (float):

The starting value of the AC parameter sweep, e.g., amplitude(volts/amps), offset(volts/amps), or frequency(Hz).

stop_value (float):

The ending value of the AC parameter sweep, e.g., amplitude(volts/amps), offset(volts/amps), or frequency(Hz).

source_amplitude (float):

The AC voltage amplitude in volts or amps.

source_frequency (float):

The AC frequency in Hz.

source_offset (float):

The AC offset in volts or amps.

dwell_time (float):

Time duration of the measurement portion of each step in seconds, defaults to AUTO.

number_of_points (int):

How many steps to take going from the start to the stop amplitude. AUTO chooses a decade step size that results in between 100 and 1,000 points

optional_custom_data_sources (SSMSystemDataSourceMnemonic or str, int):

Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX). Defaults to AC on the source's matching measure channel if unspecified. If specified only the specified custom data sources will be collected. If specified the corresponding measure channel will not be configured automatically.

settle_time (float):

This is the user blanking time. How long to wait before taking a measurement after a step change. Defaults to zero.

initial_delay (float):

Time to wait before starting the sweep, defaults to zero.

sweep_spacing (str or SweepSpacing enum):

LINEAR or LOGARITHMIC sweep.

round_trip (bool):

Sweeps to the stop amplitude then sweeps back to the start amplitude when true. Returns double the sweep number of points when true.

include_source_sweep_parameter (bool):

The source sweep parameter is included in the returned data when true. Not relevant if using custom data sources.

include_relative_time (bool):

The time since the start of the sweep is included in the returned data when true. Not relevant if using custom data sources.

Measure Module methods

```
class lakeshore.ssm_measure_module.MeasureModule(module_number, device)
```

Class for interaction with a specific measure channel of the M81 instrument.

get_name()

Returns the user-settable name of the module.

set_name(*new_name*)

Set the name of the module.

get_notes()

Returns the user-settable notes of the module.

set_notes(*new_note*)

Set the notes of the module.

get_model()

Returns the model of the module (i.e. VM-10).

get_serial()

Returns the serial number of the module (i.e. LSA1234).

get_hw_version()

Returns the hardware version of the module.

get_self_cal_status()

Returns the status of the last self calibration of the module.

run_self_cal()

Run a self calibration for the module.

reset_self_cal()

Restore factory self calibration for the module.

get_averaging_time()

Returns the averaging time of the module in Power Line Cycles. Not relevant in lock-in mode.

get_requested_averaging_time()

Returns the requested averaging time of the module in Power Line Cycles. Not relevant in lock-in mode.

get_actual_averaging_time()

Returns the actual averaging time of the module in Power Line Cycles. Not relevant in lock-in mode.

set_averaging_time(*nplc*)

Sets the averaging time of the module. Not relevant in lock-in mode.

Args:

nplc (float):

The new number of power line cycles to average.

get_mode()

Returns the measurement mode of the module. 'DC', 'AC', or 'LIA'.

set_mode(*mode*)

Sets the measurement mode of the module.

Args:

mode (str):

The new measurement mode ('DC', 'AC', or 'LIA').

get_coupling()

Return input coupling of the module. 'AC' or 'DC'.

set_coupling(*coupling*)

Sets the input coupling of the module.

Args:

coupling (str):

The new input coupling ('AC' or 'DC').

use_ac_coupling()

Sets the input coupling of the module to 'AC'.

use_dc_coupling()

Sets the input coupling of the module to 'DC'.

get_input_configuration()

Returns the input configuration of the module. 'AB', 'A', or 'GROUND'.

set_input_configuration(*input_configuration*)

Sets the input configuration of the module.

Args:

input_configuration (str):

The new input configuration ('AB', 'A', or 'GROUND').

enable_bias_voltage()

Enables the bias voltage applied to the amplifier.

disable_bias_voltage()

Disables the bias voltage applied to the amplifier.

get_bias_voltage_enabled()

Return whether the bias voltage is enabled.

get_bias_voltage()

Return the bias voltage applied on the amplifier input in Volts.

set_bias_voltage(*bias_voltage*)

Sets the bias voltage applied on the amplifier input.

Args:**bias_voltage (float):**

 The new bias voltage in Volts.

get_filter_state()

Returns whether the hardware filter is engaged.

get_lowpass_corner_frequency()

Returns the analog low pass filter cutoff frequency.

'NONE', 'F10', 'F30', 'F100', 'F300', 'F1000', 'F3000', or 'F10000'.

get_lowpass_rolloff()

Returns the analog low pass filter roll-off.

'R6' or 'R12'.

get_highpass_corner_frequency()

Returns the high pass filter cutoff frequency.

'NONE', 'F10', 'F30', 'F100', 'F300', 'F1000', or 'F3000'.

get_highpass_rolloff()

Returns the high pass filter roll-off.

'R6' or 'R12'.

get_gain_allocation_strategy()

Returns the gain allocation strategy used for the hardware filter.

'NOISE', or 'RESERVE'.

set_gain_allocation_strategy(*optimization_type*)

Sets the gain allocation strategy used for the hardware filter.

Args:**optimization_type (str):**

 The new optimization type ('NOISE', or 'RESERVE').

configure_input_lowpass_filter(*corner_frequency*, *rolloff*='R12')

Configure the analog input low pass filter.

Args:

corner_frequency (str):

The low pass corner frequency. ('NONE', 'F10', 'F30', 'F100', 'F300', 'F1000', 'F3000', or 'F10000'). F10 = 10 Hz, etc.

rolloff (str):

The low pass roll-off. ('R6' or 'R12'). R6 = 6 dB/Octave, R12 = 12 dB/Octave.

configure_input_highpass_filter(*corner_frequency*, *rolloff*='R12')

Configure the analog input high pass filter.

Args:

corner_frequency (str):

The high pass corner frequency. ('NONE', 'F10', 'F30', 'F100', 'F300', 'F1000', or 'F3000'). F10 = 10 Hz, etc.

rolloff (str):

The high pass roll-off ('R6' or 'R12'). R6 = 6 dB/Octave, R12 = 12 dB/Octave.

disable_input_filters()

Disables the hardware filters.

get_current_range()

Returns the current range in Amps.

get_i_range()

Returns the current range in Amps

Deprecated since version 1.5.4: Use `get_current_range` instead

get_current_autorange_status()

Returns whether auto-ranging is enabled for the module.

get_i_autorange_status()

Returns whether autoranging is enabled for the module

Deprecated since version 1.5.4: Use `get_current_autorange_status` instead

configure_current_range(*autorange*, *max_level*=None)

Configure current ranging for the module.

Args:

autorange (bool):

True to enable real time range decisions by the module. False for manual ranging.

max_level (float):

The largest current that needs to be measured by the module in Amps.

configure_i_range(*autorange*, *max_level*=None)

Configure current ranging for the module

Deprecated since version 1.5.4: Use `configure_current_range` instead

Args:

autorange (bool):

True to enable real time range decisions by the module. False for manual ranging.

max_level (float):

The largest current that needs to be measured by the module in Amps.

get_voltage_range()

Returns the voltage range in Volts.

get_voltage_autorange_status()

Returns whether auto-ranging is enabled for the module.

configure_voltage_range(*autorange, max_level=None*)

Configure voltage ranging for the module.

Args:**autorange (bool):**

True to enable real time range decisions by the module. False for manual ranging.

max_level (float):

The largest voltage that needs to be measured by the module in Volts.

get_reference_source()

Returns the lock-in reference source. 'S1', 'S2', 'S3', 'RIN'.

set_reference_source(*reference_source*)

Sets the lock-in reference source

Args:**reference_source (str):**

The new reference source ('S1', 'S2', 'S3', 'RIN')

get_reference_harmonic()

Returns the lock-in reference harmonic.

set_reference_harmonic(*harmonic*)

Sets the lock-in reference harmonic.

Args:**harmonic (int):**

The new reference harmonic. 1 is the fundamental frequency, 2 is twice the fundamental frequency, etc.

get_reference_phase_shift()

Returns the lock-in reference phase shift in degrees.

set_reference_phase_shift(*phase_shift*)

Sets the lock-in reference phase shift.

Args:**phase_shift (float):**

The new reference phase shift in degrees.

auto_phase()

Executes a one time adjustment of the reference phase shift so that the present phase measurement is zero.

get_lock_in_time_constant()

Returns the lock-in time constant in seconds.

set_lock_in_time_constant(*time_constant*)

Sets the lock-in time constant.

Args:

time_constant (float):

The new time constant in seconds.

get_lock_in_settle_time(*settle_percent=0.01*)

Returns the lock-in settle time in seconds.

Args:

settle_percent (float):

The desired percent signal has settled to in percent. A value of *0.1* is interpreted as 0.1 %.

get_lock_in_equivalent_noise_bandwidth()

Returns the equivalent noise bandwidth (ENBW) in Hz.

get_lock_in_rolloff()

Returns the lock-in (PSD) output filter roll-off for the present module. 'R6', 'R12', 'R18' or 'R24'.

set_lock_in_rolloff(*rolloff*)

Sets the lock-in (PSD) output filter roll-off.

Args:

rolloff (str):

The new (PSD) output filter roll-off ('R6', 'R12', 'R18' or 'R24').

get_lock_in_lowpass_state()

Returns the state of the lock-in (PSD) output IIR filter.

get_lock_in_iir_state()

This function is deprecated. Use `get_lock_in_lowpass_state()` instead.

Deprecated since version 1.9.0: Use `get_lock_in_lowpass_state` instead.

set_lock_in_lowpass_state(*state*)

Sets the state of the lock-in (PSD) output IIR filter.

Args:

state (bool):

The new state of the (PSD) output IIR filter.

set_lock_in_iir_state(*state*)

This function is deprecated. Use `set_lock_in_lowpass_state()` instead.

Deprecated since version 1.9.0: Use `set_lock_in_lowpass_state` instead.

enable_lock_in_lowpass()

Sets the state of the lock-in (PSD) output IIR filter to True.

enable_lock_in_iir()

This function is deprecated. Use `enable_lock_in_lowpass()` instead.

Deprecated since version 1.9.0: Use `enable_lock_in_lowpass` instead.

disable_lock_in_lowpass()

Sets the state of the lock-in (PSD) output IIR filter to False.

disable_lock_in_iir()

This function is deprecated. Use `disable_lock_in_lowpass()` instead.

Deprecated since version 1.9.0: Use `disable_lock_in_lowpass` instead.

get_lock_in_averaging_state()

Returns the state of the lock-in (PSD) output averaging (FIR) filter.

get_lock_in_fir_state()

This function is deprecated. Use `get_lock_in_averaging_state()` instead.

Deprecated since version 1.9.0: Use `get_lock_in_averaging_state` instead.

set_lock_in_averaging_state(*state*)

Sets the state of the lock-in (PSD) output averaging (FIR) filter.

Args:**state (bool):**

The new state of the (PSD) output averaging (FIR) filter.

set_lock_in_fir_state(*state*)

This function is deprecated. Use `set_lock_in_averaging_state()` instead.

Deprecated since version 1.9.0: Use `set_lock_in_averaging_state` instead.

enable_lock_in_averaging()

Sets the state of the lock-in (PSD) output averaging (FIR) filter to True.

enable_lock_in_fir()

This function is deprecated. Use `enable_lock_in_averaging()` instead.

Deprecated since version 1.9.0: Use `enable_lock_in_averaging` instead.

disable_lock_in_averaging()

Sets the state of the lock-in (PSD) output averaging (FIR) filter to False.

disable_lock_in_fir()

This function is deprecated. Use `disable_lock_in_averaging()` instead.

Deprecated since version 1.9.0: Use `disable_lock_in_averaging` instead.

get_lock_in_reference_cycles()

Returns the number of averaging (FIR) cycles.

get_lock_in_fir_cycles()

This function is deprecated. Use `get_lock_in_reference_cycles()` instead.

Deprecated since version 1.9.0: Use `get_lock_in_reference_cycles` instead.

set_lock_in_reference_cycles(*cycles*)

Sets the number of averaging (FIR) cycles.

Args:**cycles (int):**

The desired number of averaging (FIR) cycles, between 1 and 100.

set_lock_in_fir_cycles(*cycles*)

This function is deprecated. Use `set_lock_in_reference_cycles()` instead.

Deprecated since version 1.9.0: Use `set_lock_in_reference_cycles` instead.

setup_dc_measurement(*nplc=1*)

Set up the module for DC measurement.

Args:

nplc (float):

The new number of power line cycles to average.

setup_ac_measurement(*nplc=1*)

Set up the module for AC measurement.

Args:

nplc (float):

The new number of power line cycles to average.

setup_lock_in_measurement(*reference_source, time_constant, rolloff='R24', reference_phase_shift=0.0, reference_harmonic=1, use_fir=True*)

Set up the module for lock-in measurement.

Args:

reference_source (str):

Lock-in reference source ('S1', 'S2', 'S3', 'RIN').

time_constant (float):

Time constant in seconds.

rolloff (str):

Lock-in (PSD) output filter roll-off ('R6', 'R12', 'R18' or 'R12').

reference_phase_shift (float):

Lock-in reference phase shift in degrees.

reference_harmonic (int):

Lock-in reference harmonic. 1 is the fundamental frequency, 2 is twice the fundamental frequency, etc.

use_averaging (FIR) (bool):

Enable or disable the (PSD) output averaging (FIR) filter.

zero_relative_baseline()

Sets the present measurement as the baseline value for calculating relative readings.

set_relative_baseline(*baseline*)

Sets the relative baseline.

get_relative_baseline()

Returns the relative baseline.

get_multiple(**data_sources*)

This function is deprecated. Use `fetch_multiple()` instead.

get_dc()

Returns the DC measurement in module units.

get_dc_relative()

Returns the relative DC measurement in module units.

get_dc_minimum()

Returns the minimum DC indication in module units.

get_dc_maximum()

Returns the maximum DC indication in module units.

get_rms()

Returns the RMS measurement in module units.

get_rms_relative()

Returns the relative RMS measurement in module units.

get_rms_minimum()

Returns the minimum RMS indication in module units.

get_rms_maximum()

Returns the maximum RMS indication in module units.

get_peak_to_peak()

Returns the peak to peak measurement in module units.

get_peak_to_peak_minimum()

Returns the minimum peak to peak indication in module units.

get_peak_to_peak_maximum()

Returns the maximum peak to peak indication in module units.

get_positive_peak()

Returns the positive peak measurement in module units.

get_positive_peak_minimum()

Returns the minimum positive peak indication in module units.

get_positive_peak_maximum()

Returns the maximum positive peak indication in module units.

get_negative_peak()

Returns the negative peak measurement in module units.

get_negative_peak_minimum()

Returns the minimum negative peak indication in module units.

get_negative_peak_maximum()

Returns the maximum negative peak indication in module units.

get_lock_in_x()

Returns the present X measurement from the lock-in.

get_lock_in_x_minimum()

Returns the minimum X indication from the lock in.

get_lock_in_x_maximum()

Returns the maximum X indication from the lock in.

get_lock_in_y()

Returns the present Y measurement from the lock-in.

get_lock_in_y_minimum()

Returns the minimum Y indication from the lock in.

get_lock_in_y_maximum()

Returns the maximum Y indication from the lock in.

get_lock_in_r()

Returns the present magnitude measurement from the lock-in.

get_lock_in_r_minimum()

Returns the minimum magnitude indication from the lock in.

get_lock_in_r_maximum()

Returns the maximum magnitude indication from the lock in.

get_lock_in_theta()

Returns the present angle measurement from the lock-in.

get_lock_in_theta_minimum()

Returns the minimum angle indication from the lock in.

get_lock_in_theta_maximum()

Returns the maximum angle indication from the lock in.

get_lock_in_frequency()

Returns the present detected frequency from the Phase Locked Loop (PLL).

get_lock_in_dc()

Returns the present dc measurement from the lock-in.

get_lock_in_dc_immediate()

Returns the immediate dc measurement from the lock-in.

get_pll_lock_status()

Returns the present lock status of the PLL. True if locked, False if unlocked.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_overload_status()

Returns whether the module is presently in overload or not

get_settling_status()

Returns whether the module is presently settling or not

get_unlocked_status()

Returns whether the module is presently unlocked or not

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:**register_mask ([Instrument]OperationRegister):**

An instrument specific OperationRegister class object with all bits configured true or false.

get_identify_state()

Returns the identification state for the given pod.

set_identify_state(*state*)

Sets the identification state for the given pod.

Args:**state (bool):**

The desired state for the LED, 1 for identify, 0 for normal state.

get_dark_mode_state()

Returns the dark mode state for the given pod.

set_dark_mode_state(*state*)

Configures the dark mode state for the given pod.

Args:**state (bool):**

The desired operation for the LED, 1 for normal mode, 0 for dark mode.

get_frequency_range_threshold()

Returns the frequency range threshold for the module.

Frequency range threshold normalized to the -3 db point. For example, a value of 0.1 means 10 % of the -3 db point.

set_frequency_range_threshold(*threshold*)

Sets the frequency range threshold for the module.

When the modules range is set to Auto, a range such that the frequency of the signal does not exceed the given percentage of the bandwidth of the range will be chosen.

Args:

threshold (float):

Frequency range threshold normalized to the -3 db point with a valid range of 0.0 to 1.0. For example, a value of 0.1 means 10 % of the -3 db point.

get_digital_high_pass_filter_state()

Returns the state of the digital high pass filter for lock-in mode.

set_digital_high_pass_filter_state(*state*)

Sets the state of the digital high pass filter for lock-in mode.

Args:

state (bool):

The desired state for the digital lock-in high pass filter. 1 for enabled, 0 for disabled.

get_resistance()

Returns the present resistance measurement in Ohms.

A valid source must be configured. When resistance excitation type is DC, the returned value is the DC resistance. When resistance excitation type is AC, the returned value is the in-phase resistance.

Returns:

float: Present resistance measurement in Ohms.

get_resistance_dc()

Returns the present DC resistance measurement in Ohms.

A valid source must be configured. The resistance excitation type must be DC.

Returns:

float: Present DC resistance measurement in Ohms.

get_resistance_inphase()

Returns the present in-phase component of the AC resistance measurement in Ohms.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present in-phase component of the AC resistance measurement in Ohms.

get_resistance_quadrature()

Returns the present quadrature component of the AC resistance measurement in Ohms.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present quadrature component of the AC resistance measurement in Ohms.

get_resistance_magnitude()

Returns the present magnitude of the AC resistance measurement in Ohms.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present magnitude of the AC resistance measurement in Ohms.

get_resistance_phase()

Returns the present phase of the AC resistance measurement in degrees.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present phase of the AC resistance measurement in degrees.

get_resistance_compensated()

Returns the present compensated resistance of the resistance measurement in Ohms.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present compensated resistance measurement in Ohms.

get_resistance_capacitance()

Returns the present capacitance of the resistance measurement in Farads.

A valid source must be configured. The resistance excitation type must be AC.

Returns:

float: Present capacitance measurement in Farads.

set_resistance_source(*source_module*)

Configures the resistance feature to use a specified source module to calculate resistance.

Args:

source_module (SourceModule): The channel used for calculating resistance.

get_resistance_source()

Returns the present source module being used to calculate resistance.

Returns:

SourceModule: The channel used for calculating resistance.

set_resistance_excitation_type(*excitation_type*)

Sets the present resistance excitation type of the module.

For “DC”, the measure mode is DC and source shape is DC. For “AC”, the measure mode is Lock-in and source shape is Sine.

Args:

excitation_type (ResistanceExcitationType): The desired resistance excitation type.

get_resistance_excitation_type()

Returns the present resistance excitation type of the module.

This represents the combination of the measure module mode and the selected source module shape. For “DC”, the measure mode is DC and source shape is DC. For “AC”, the measure mode is Lock-in and source shape is Sine.

Returns:

ResistanceExcitationType: The resistance excitation type.

set_resistance_mode(*resistance_mode*)

Sets the resistance optimization mode of the module.

Args:

resistance_mode (ResistanceMode): The desired resistance optimization mode.

get_resistance_mode()

Returns the preset resistance optimization mode of the module.

Returns:

ResistanceMode: The present resistance optimization mode. "NOISE" or "POWER".

set_resistance_range(*resistance_range*)

Sets the resistance range of the module.

Args:

resistance_range (float): The desired resistance range in Ohms.

get_resistance_range()

Returns the present resistance range of the module.

Returns:

float: The resistance range in Ohms.

set_resistance_auto_range(*state*)

Sets the resistance auto-range state of the module.

Args:

state (bool): The desired state of resistance auto-range. True to enable; else false.

get_resistance_auto_range()

Gets the state of resistance auto-range on the module.

Returns:

bool: The state of resistance auto-range on the module.

set_resistance_optimization_state(*optimization_state*)

Sets the state of resistance optimization on the module

Args:

optimization_state (bool): The desired state of resistance optimization. True if optimizing for resistance, else False.

get_resistance_optimization_state()

Returns the present state of optimization on the module.

When optimization is enabled, the instrument will set other settings based on the selected resistance range and mode settings.

Returns:

bool: The state of resistance optimization. True if optimizing for resistance, else False.

set_resistance_observation_time_state(*state*)

Sets the state of the observation time on the module.

Args:

state (bool): The state of resistance observation time.

get_resistance_observation_time_state()

Gets the present state of the observation time on the module.

Returns:

bool: The state of resistance observation time.

set_resistance_observation_time_requested(*requested_time*)

Sets the requested observation time on the module.

Args:

requested_time (float): The requested observation time.

get_resistance_observation_time_requested()

Gets the present requested observation time on the module.

Returns:

float: The requested observation time.

get_resistance_observation_time_actual()

Gets the present actual observation time for the resistance calculation.

This is a best-fit calculation based on the requested observation time and reference frequency.

Returns:

float: The actual observation time.

get_resistance_observation_time_enbw()

Gets the present equivalent noise bandwidth (ENBW) for the actual resistance observation time.

Returns:

float: The calculated equivalent noise bandwidth.

get_resistance_relative_dc()

Gets the present relative dc resistance calculation. This is calculated by subtracting the baseline resistance value from the present dc resistance calculation.

Returns:

float: The calculated relative dc resistance.

set_resistance_relative_baseline(*baseline*)

Sets the present relative baseline value to be subtracted from the dc resistance calculation in order to
calculate the relative dc resistance.

Args:

baseline (float): The requested baseline.

get_resistance_relative_baseline()

Gets the present relative resistance baseline that is used to calculate relative dc resistance.

Returns:

float: The resistance baseline.

zero_dc_resistance()

Sets the baseline value for calculating relative resistance to the present DC resistance value. This sets the relative resistance calculation to zero in an ideal configuration.

clear_relative_resistance_baseline()

Sets the baseline value for calculating relative resistance to 0. The resistance calculation and relative resistance calculation are equivalent when the baseline is zero.

reset_settings()

Resets the settings for the module to their power on defaults.

unload()

Unloads the module.

get_load_state()

Returns the loaded state for the module.

fetch_multiple(*data_sources)

Returns a list of the latest values corresponding to the input data sources for this module quickly

Args:

data_sources (SSMSystemDataSourceMnemonic or str):

Variable length list of data sources.

Returns:

Tuple of values corresponding to the given data sources for this module.

read_multiple(*data_sources)

Returns new values after measurement based on present input data source.

Initiates measurement of new values corresponding to the input data sources for this module and returns them after the measurement is complete.

Args:

data_sources (SSMSystemReadDataSourceMnemonic or str):

Variable length list of data sources.

Returns:

Tuple of values corresponding to the given data sources for this module.

get_self_cal_datetime()

Returns the self calibration date and time for the module.

get_self_cal_temperature()

Returns the self calibration temperature for the module.

get_measure_function()

Returns whether the measure function is current or voltage

set_measure_function(value)

Sets the measure function of the module.

Args:

value (str):

The new measure function ('CURRent' or 'VOLTage').

get_voltage_sense_mode()

Returns the voltage measure sense of the specified module

set_voltage_sense_mode(value)

Sets the voltage measure sense of the specified module

Args:

value (str):
LOCAL for 2-wire local REMote for 4-wire remote

Settings Profiles methods

class lakeshore.ssm_settings_profiles.**SettingsProfiles**(*device*)

Class for interaction with settings profiles.

get_summary(*name*)
Returns a list containing a profile's description and module models.

Args:

name (str):
Name of the profile to query.

create(*name, description=""*)
Create a new profile using the present instrument configuration.

Args:

name (str):
Unique name to give the profile.

description (str):
Optional description of the profile.

get_list()
Returns a list of the saved profile names.

get_description(*name*)
Returns a profile's description.

Args:

name (str):
Name of the profile to get the description for.

set_description(*name, description*)
Sets a profile's description. Any existing description will be overwritten.

Args:

name (str):
Name of the profile to get the description for.

description (str):
The new description of the profile.

get_json(*name*)
Returns a JSON object of a given profile.

Args:

name (str):
Name of the profile.

rename(*name, new_name*)
Rename a profile. New name must be unique.

Args:

name (str):
The name of the profile to rename.

new_name (str):
The new name of the profile.

update(*name*)
Update a profile with the present instrument configuration.

Args:
name (str):
The name of the profile to update.

get_valid_for_restore(*name*)
Returns if a profile is valid to restore.

Args:
name (str):
The name of the profile to validate.

restore(*name*)
Restore a profile.

Args:
name (str):
The name of the profile to restore.

delete(*name*)
Delete a profile.

Args:
name (str):
The name of the profile to delete.

delete_all()
Delete all profiles.

Instrument registers

This page outlines the registers used to interact with various settings and methods of the M81.

```
class lakeshore.ssm_system.SSMSystemOperationRegister(s1_summary, s2_summary, s3_summary,  
                                                    m1_summary, m2_summary, m3_summary,  
                                                    data_stream_in_progress)
```

Class object representing the operation status register.

```
class lakeshore.ssm_system.SSMSystemQuestionableRegister(s1_summary, s2_summary, s3_summary,  
                                                       m1_summary, m2_summary,  
                                                       m3_summary, critical_startup_error,  
                                                       critical_runtime_error, heartbeat,  
                                                       calibration, data_stream_overflow)
```

Class object representing the questionable status register.

```
class lakeshore.ssm_base_module.SSMSystemModuleQuestionableRegister(read_error=False, unrecognized_pod_error=False, port_direction_error=False, factory_calibration_failure=False, self_calibration_failure=False)
```

Class object representing the questionable status register of a module.

```
class lakeshore.ssm_source_module.SSMSystemSourceModuleOperationRegister(v_limit, i_limit, sweeping)
```

Class object representing the operation status register of a source module.

```
class lakeshore.ssm_measure_module.SSMSystemMeasureModuleOperationRegister(overload, settling, unlocked)
```

Class object representing the operation status register of a measure module.

Enumeration Objects

This section describes the Enum type objects that have been created for the M81 SSM.

```
class lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMnemonic(*values)
```

Enumeration of M81 data source mnemonics.

```
RELATIVE_TIME = 'RTIME'
```

```
SOURCE_AMPLITUDE = 'SAMplitude'
```

```
SOURCE_OFFSET = 'SOFFset'
```

```
SOURCE_FREQUENCY = 'SFrequency'
```

```
SOURCE_RANGE = 'SRANge'
```

```
SOURCE_VOLTAGE_LIMIT = 'SVLimit'
```

```
SOURCE_CURRENT_LIMIT = 'SILimit'
```

```
SOURCE_IS_SWEEPING = 'SSweeping'
```

```
MEASURE_DC = 'MDC'
```

```
MEASURE_RMS = 'MRMS'
```

```
MEASURE_POSITIVE_PEAK = 'MPPeak'
```

```
MEASURE_NEGATIVE_PEAK = 'MNPeak'
```

```
MEASURE_PEAK_TO_PEAK = 'MPTPeak'
```

```
MEASURE_X = 'MX'
```

```
MEASURE_Y = 'MY'
```

```
MEASURE_R = 'MR'
```

```
MEASURE_THETA = 'MTHeta'
```

```
MEASURE_RANGE = 'MRANge'
```

```
MEASURE_OVERLOAD = 'MOverload'  
MEASURE_SETTLING = 'MSETtling'  
MEASURE_UNLOCK = 'MUNLock'  
MEASURE_REFERENCE_FREQUENCY = 'MRFRequency'  
GENERAL_PURPOSE_INPUT_STATES = 'GPISStates'  
GENERAL_PURPOSE_OUTPUT_STATES = 'GPOStates'
```

2.4.4 Temperature Controllers

Model 335 Cryogenic Temperature Controller

The Model 335 measures and controls cryogenic temperature environments.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example Scripts

Setting a temperature curve

```
import matplotlib.pyplot as plt  
from lakeshore import Model224, Model224CurveHeader  
  
# Connect to a temperature instrument (the Model 224 in this case) over USB  
myinstrument = Model224()  
  
# Configure a curve by first setting its header parameters. First, set the name and  
↳ serial number of the curve.  
# Then, select the units used to set map the sensor units to temperature units. Set a  
↳ temperature limit, and  
# then specify whether the coefficients are positive or negative.  
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.  
↳ VOLTS_PER_KELVIN, 300.0,  
                                     myinstrument.CurveTemperatureCoefficients.POSITIVE)  
myinstrument.set_curve_header(25, curve_header_25)  
  
# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set  
↳ to equal a Kelvin value of  
# 276.0  
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)  
  
# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.↳  
↳ The instrument generates  
# a new curve using your entered data points and the selected standard curve  
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,↳  
↳ 30, "SN123", (276, 10),  
               (300, 5), (310, 2))  
  
# Use the get_curve method to get all the data points for a curve as a list. This can↳  
↳ then be used to create a plot
```

(continues on next page)

(continued from previous page)

```

# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that
# curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)

```

Recording data with the Model 335

```

from lakeshore import Model335, Model335InputSensorSettings

# Connect to the first available Model 335 temperature controller over USB using a baud
# rate of 57600
my_model_335 = Model335(57600)

# Create a new instance of the input sensor settings class
sensor_settings = Model335InputSensorSettings(my_model_335.InputSensorType.DIODE, True,
# False,
# my_model_335.InputSensorUnits.KELVIN,
# my_model_335.DiodeRange.TWO_POINT_FIVE_
# VOLTS)

# Apply these settings to input A of the instrument
my_model_335.set_input_sensor("A", sensor_settings)

# Set diode excitation current on channel A to 10uA
my_model_335.set_diode_excitation_current("A", my_model_335.DiodeCurrent.TEN_MICROAMPS)

# Collect instrument data
heater_output_1 = my_model_335.get_heater_output(1)
heater_output_2 = my_model_335.get_heater_output(2)
temperature_reading = my_model_335.get_all_kelvin_reading()

# Open a csv file to write
file = open("335_record_data.csv", "w")

# Write the data to the file
file.write("Data retrieved from the Lake Shore Model 335\n")
file.write("Temperature Reading A: " + str(temperature_reading[0]) + "\n")
file.write("Temperature Reading B: " + str(temperature_reading[1]) + "\n")
file.write("Heater Output 1: " + str(heater_output_1) + "\n")
file.write("Heater Output 2: " + str(heater_output_2) + "\n")
file.close()

```

Setting up autotune on the Model 335

```

from lakeshore import Model335
from time import sleep

# Connect to the first available Model 335 temperature controller over USB using a baud_
↳rate of 57600
my_model_335 = Model335(57600)

# It is assumed that the instrument is configured properly with a control input sensor_
↳curve
# and heater output, capable of closed loop control

# Configure the display mode
my_model_335.set_display_setup(my_model_335.DisplaySetup.TWO_INPUT_A)

# Configure heater output 1 using the HeaterSetup class and set_heater_setup method
my_model_335.set_heater_setup_one(my_model_335.HeaterResistance.HEATER_50_OHM, 1.0, my_
↳model_335.HeaterOutputDisplay.POWER)

# Configure heater output 1 to a setpoint of 310 kelvin (units correspond to the_
↳configured output units)
set_point = 325
my_model_335.set_control_setpoint(1, set_point)

# Turn on the heater by setting the range
my_model_335.set_heater_range(1, my_model_335.HeaterRange.HIGH)

# Check to see if there are any heater related errors
heater_error = my_model_335.get_heater_status(1)
if heater_error is not my_model_335.HeaterError.NO_ERROR:
    raise Exception(heater_error.name)

# Allow the heater some time to turn on and start maintaining a setpoint
sleep(10)

# Ensure that the temperature is within 5 degrees kelvin of the setpoint
kelvin_reading = my_model_335.get_kelvin_reading(1)
if (kelvin_reading < (set_point - 5)) or (kelvin_reading > (set_point + 5)):
    raise Exception("Temperature reading is not within 5k of the setpoint")

# Initiate autotune in PI mode, initial conditions will not be met if the system is not
# maintaining a temperature within 5 K of the setpoint
my_model_335.set_autotune(1, my_model_335.AutotuneMode.P_I)

# Poll the instrument until the autotune process completes
autotune_status = my_model_335.get_tuning_control_status()
while autotune_status["active_tuning_enable"] and not autotune_status["tuning_error"]:
    autotune_status = my_model_335.get_tuning_control_status()
    # Print the status to the console every 5 seconds
    print("Active tuning: " + str(autotune_status["active_tuning_enable"]))
    print("Stage status: " + str(autotune_status["stage_status"]) + "/10")
    sleep(5)

```

(continues on next page)

(continued from previous page)

```

if autotune_status["tuning_error"]:
    raise Exception("An error occurred while running autotune")

```

Instrument class methods

```

class lakeshore.model_335.Model335(baud_rate, serial_number=None, com_port=None, timeout=2.0,
                                   ip_address=None, tcp_port=None, **kwargs)

```

A class object representing the Lake Shore Model 335 cryogenic temperature controller.

get_analog_output_percentage(*output*)

Returns the output percentage of the analog voltage output.

Args:

output (int):

Specifies which analog voltage output to query.

Returns:

(float):

Analog voltage heater output percentage.

set_autotune(*output, mode*)

Initiates auto-tuning of the heater control loop.

Args:

output (int):

Specifies the output associated with the loop to be Auto-tuned.

mode (TemperatureControllerEnums.AutotuneMode):

Specifies the Autotune mode. Member of instrument's AutotuneMode IntEnum class.

set_brightness(*brightness*)

Method to set the front display brightness.

Args:

brightness (TemperatureControllerEnums.BrightnessLevel):

A member of the instrument's BrightnessLevel IntEnum class.

get_brightness()

Method to query the front display brightness.

Returns:

(TemperatureControllerEnums.BrightnessLevel):

A member of the instrument's BrightnessLevel IntEnum class.

get_operation_condition()

Returns the names of the operation condition register bits and their values.

Returns:

OperationEvent:

The object with the names of the operation condition register bits and their values.

get_operation_event_enable()

Returns the names of the operation event enable register and their values.

These values determine which bits propagate to the operation condition register.

Returns:

OperationEvent:

The object with the names of the operation event enable register and their values.

set_operation_event_enable(*register_mask*)

Configures values of the operation event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (OperationEvent):

An OperationEvent class object with all bits configured true or false.

get_operation_event()

Returns the names of the operation event register bits and their values.

Returns:

OperationEvent:

The object with the names of the operation event register bits and their values.

get_thermocouple_junction_temp()

Returns the temperature of the ceramic thermocouple block from the room temperature compensation calculation.

Returns:

(float):

Temperature of the ceramic thermocouple block (kelvin).

set_soft_cal_curve_dt_470(*curve_number*, *serial_number*, *calibration_point_1*=(4.2, 1.62622), *calibration_point_2*=(77.35, 1.02032), *calibration_point_3*=(305, 0.50691))

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured DT-470 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_soft_cal_curve_pt_100(*curve_number, serial_number, calibration_point_1=(77.35, 20.234), calibration_point_2=(305, 112.384), calibration_point_3=(480, 178.353)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured PT-100 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_soft_cal_curve_pt_1000(*curve_number, serial_number, calibration_point_1=(77.35, 202.34), calibration_point_2=(305, 1123.84), calibration_point_3=(480, 1783.53)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured PT-1000 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_diode_excitation_current(*channel, excitation_current*)

Sets the excitation current of a specific channel.

The 10 uA excitation current is the only calibrated excitation current, and is used in almost all applications. The Model 336 will default the 10 uA current setting any time the input sensor type is changed.

Args:

channel (str):

Specifies which sensor input to configure: "A" - "D".

excitation_current (TemperatureControllerEnums.DiodeCurrent):

A member of the instrument's DiodeCurrent IntEnum class.

get_diode_excitation_current(*channel*)

Returns the diode excitation current setting as a string.

Args:

channel (str):

Specifies which input to return: "A" - "D".

Returns:

(TemperatureControllerEnums.DiodeCurrent):

A member of the instrument's DiodeCurrent IntEnum class. Diode excitation current.

get_tuning_control_status()

Returns dictionary of tuning control status values.

If initial conditions are not met when starting the autotune procedure, causing the auto-tuning process to never actually begin, then the error status will be set to 1 and the stage status will be stage 00.

Returns:

(dict[str, bool | int]):

{"active_tuning_enable": bool, "output": int, "tuning_error": bool, "stage_status": int}

active_tuning_enable: False = no active tuning, True = active tuning. output: Heater output of the control loop being tuned. tuning_error: False = no tuning error, True = tuning error. stage_status: Specifies the current stage in the Autotune process. If tuning error occurred, stage status represents stage that failed.

set_filter(input_channel, filter_enable, data_points, reset_threshold)

Configures the input_channel filter parameter.

Args:

input_channel (int or str):

Specifies which input channel to configure.

filter_enable (bool):

Specified whether the filtering function is enabled or not.

data_points (int):

Specifies how many points the filter function uses: 2 - 64.

reset_threshold (int):

Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets. Options are: 1% - 10%.

get_filter(input_channel)

Returns the input_channel filter configuration.

Args:

input_channel (int or str):

Specifies which input channel to configure.

Returns:

(dict[str, bool | int])

{"filter_enable": bool, "data_points": int, "reset_threshold": int}

filter_enable: Specified whether the filtering function is enabled or not. data_points: Specifies how many points the filter function uses. reset_threshold: Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets (1% - 10%).

set_monitor_output_heater(*channel, high_value, low_value, units=MonitorOutUnits.KELVIN, polarity=Polarity.UNIPOLAR*)

Configures output 2. Use the set_heater_output_mode command to set the output mode to Monitor Out.

Args:

channel (self.InputSensor):

Specifies which sensor input to monitor.

high_value (float):

Represents the data at which the Monitor Out reaches +100% output. Entered in the units designated by the <units> argument.

low_value (float):

Represents the data at which the analog output reaches -100% output if bipolar, or 0% output if unipolar. Entered in the units designated by the <units> argument.

units (self.MonitorOutUnits):

Specifies the units on which to base the output voltage.

polarity (self.Polarity):

Specifies output voltage is unipolar or bipolar.

get_monitor_output_heater()

Used to obtain all monitor out parameters for output 2.

Returns:

(dict):

```
{“channel”: self.InputSensor, “units”: self.MonitorOutUnits, “high_value”: float, “low_value”: float, “polarity”: self.Polarity}
```

See set_monitor_output_heater method arguments

get_celsius_reading(*channel*)

Returns the temperature value in Celsius of either channel.

Args:

channel (str):

Selects the sensor input to query (“A” or “B”),

set_display_setup(*mode*)

Sets the display mode.

Args:

mode (self.DisplaySetup):

Specifies the front panel display mode. See self.DisplaySetup IntEnum class.

get_display_setup()

Returns the display mode.

Return:

(self.DisplaySetup):

Specifies the front panel display mode. See self.DisplaySetup IntEnum class.

set_heater_setup_one(*heater_resistance, max_current, output_display_mode*)

Method to configure heater output one.

Args:

heater_resistance (self.HeaterResistance):

See self.HeaterResistance IntEnum class.

max_current (float):

Specifies the maximum current for the heater.

output_display_mode (self.HeaterOutputDisplay):

Specifies how the heater output is displayed. See self.HeaterOutType IntEnum class.

set_heater_setup_two(*output_type, heater_resistance, max_current, display_mode*)

Method to configure the heater output 2.

Args:

output_type (self.HeaterOutType):

Specifies whether the heater output is in constant current or voltage mode. See self.HeaterOutType IntEnum class.

heater_resistance (self.HeaterResistance):

See self.HeaterResistance IntEnum class.

max_current (float):

Specifies the maximum current for the heater.

display_mode (self.HeaterOutType):

Specifies how the heater output is displayed. Required only if output_type is set to CURRENT. See self.HeaterOutType IntEnum class.

get_heater_setup(*heater_output*)

Returns the heater configuration status.

Args:

heater_output (int):

Selects which heater output to query:

Return:

(dict):

See set_heater_setup_one/set_heater_setup_two method arguments. {"output_type": self.HeaterOutType, "heater_resistance": self.HeaterResistance, "max_current": float, "output_display_mode": self.HeaterOutputDisplay}

set_input_sensor(*channel, sensor_parameters*)

Sets the sensor type and associated parameters.

Args:

channel (str):

Specifies input to configure ("A" or "B").

sensor_parameters (self.InputSensorSettings):

See self.InputSensorSettings class.

get_input_sensor(*channel*)

Returns the sensor type and associated parameters.

Args:

channel (str):

Specifies sensor input to configure ("A" or "B").

Return:

(self.InputSensorSettings):

See self.InputSensor IntEnum class.

get_all_kelvin_reading()

Returns the temperature value in kelvin of all channels.

Return:

(list: float)

- [channel_A, channel_B]

set_heater_output_mode(output, mode, channel, powerup_enable=False)

Configures the heater output mode.

Args:**output (int):**

Specifies which output to configure (1 or 2).

mode (self.HeaterOutputMode):

Member of self.HeaterOutputMode IntEnum class. Specifies the control mode.

channel (self.InputSensor):

Specifies which input to use for control.

powerup_enable (bool):

Specifies whether the output remains on (True) or shuts off after power cycle (False).

get_heater_output_mode(output)

Returns the heater output mode for a given output and whether powerup is enabled.

Args:**output (int):**

Specifies which output to query (1 or 2).

Return:

(dict): {"mode": self.HeaterOutputMode, "channel": self.InputSensor, "powerup_enable": bool}

set_output_two_polarity(output_polarity)

Sets polarity of output 2 to either unipolar or bipolar.

Only applicable when output 2 is in voltage mode.

Args:**output_polarity (self.Polarity):**

Specifies whether output voltage is UNIPOLAR or BIPOLAR.

get_output_2_polarity()

Returns the polarity of output 2.

Return:**(self.Polarity):**

Specifies whether output is UNIPOLAR or BIPOLAR.

set_heater_range(output, heater_range)

Sets the heater range for a particular output.

The range setting has no effect if an output is in the off mode, and does not apply to an output in Monitor Out mode. An output in Monitor Out mode is always on.

Args:

output (int):

Specifies which output to configure (1 or 2).

heater_range (IntEnum):

For Outputs 1 and 2 in Current mode: self.HeaterRange IntEnum member. For Output 2 in Voltage mode: self.HeaterVoltageRange IntEnum member.

get_heater_range(*output*)

Returns the heater range for a particular output.

Args:

output (int):

Specifies which output to configure (1 or 2).

Return:

heater_range (IntEnum):

For Outputs 1 and 2 in Current mode: self.HeaterRange IntEnum member. For Output 2 in Voltage mode: self.HeaterVoltageRange IntEnum member.

all_heaters_off()

Recreates the front panel safety feature of shutting off all heaters.

get_input_reading_status(*channel*)

Returns the state of the input status flag bits.

Args:

channel (str):

Specifies which channel to query (“A” or “B”).

Return:

(InputReadingStatus):

Boolean representation of each bit of the input status flag register.

set_warmup_supply(*control*, *percentage*)

Warmup mode applies only to Output 2 in Voltage mode.

The Output Type parameter must be configured using the set_heater_setup() method, and the Output mode and Control Input parameters must be configured using the set_monitor_out_parameters() method.

Args:

control (self.WarmupControl):

Specifies the type of control used.

percentage (float):

Specifies the percentage of full scale (10 V) Monitor Out voltage to apply.

get_warmup_supply()

Returns the output 2 warmup supply configuration.

Return:

(dict):

{“control”: self.WarmupControl, “percentage”: float }

set_control_loop_zone_table(*output*, *zone*, *control_loop_zone*)

Configures the output zone parameters.

Args:

output (int):

Specifies which heater output to configure (1 or 2).

zone (int):

Specifies which zone in the table to configure (1 to 10).

control_loop_zone (ControlLoopZone):

See ControlLoopZone class.

get_control_loop_zone_table(*output*, *zone*)

Returns a list of zone control parameters for a selected output and zone.

Args:**output (int):**

Specifies which heater output to query (1 or 2).

zone (int):

Specifies which zone in the table to query (1 to 10).

Return:**(self.ControlLoopZone):**

See self.ControlLoopZone class.

class HeaterResistance(values*)**

Enumerator used to represent the different heater resistances.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class Polarity(*values)

Enumerator for unipolar or bipolar output operation.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class RelayControlAlarm(*values)

Enumeration of the setting determining which alarm(s) cause a relay to close in alarm mode.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

clear_interface_command()

Clears the bits in the SBR, SESR, OER, and terminates all operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register. Terminates all pending operations. Clears the interface, but not the controller.

command(*commands, check_errors=True)

Sends an SCPI command or multiple commands to the instrument.

Args:

commands (str):

A serial command.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

delete_curve(curve)

Deletes the user curve.

Args:

curve (int):

Specifies a user curve to delete.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

get_alarm_parameters(input_channel)

Returns the present state of all alarm parameters.

Args:

input_channel (str):

Specifies which input to configure.

Returns:

alarm_settings (AlarmSettings):

See AlarmSettings class.

get_alarm_status(channel)

Returns the high state and low state of the alarm for the specified channel.

Args:

channel (str or int)

Specifies which input channel to read from.

Returns:**(dict[str, bool]):**

{"high_state_enabled": bool, "low_state_enabled" bool}

get_control_setpoint(*output*)

Returns the value for a given control output.

Args:**output (int):**

Specifies which output's control loop to query (1 or 2).

Returns:**(float):**

The value for the set-point (in the preferred units of the control loop sensor).

get_curve(*curve*)

Returns a list of all the data points in a particular curve.

Args:**curve (int):**

Specifies which curve to set.

Returns:**list[tuple[float, float] | tuple[float, float, float]]:**

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_data_point(*curve, index*)

Returns a standard or user curve data point.

Args:**curve (int):**

Specifies which curve to query.

index (int):

Specifies the points index in the curve.

Returns:**tuple[float, float] | tuple[float, float, float]:**

(sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_header(*curve_number*)

Returns parameters set on a particular user curve header.

Args:**curve_number (int):**

Specifies a curve to retrieve.

Returns:**(CurveHeader):**

A CurveHeader class object containing the curve information.

get_display_field_settings(*field*)

Returns the settings of the specified display field when display is in Custom mode.

Args:

field (int)

Defines which field of the display to retrieve settings from.

Returns:

(dict[str, LegacyTemperatureControllerEnums.InputChannel | LegacyTemperatureControllerEnums.DisplayFieldUnits]):

See set_display_field_settings method. {"input_channel": InputChannel, "display_units": DisplayFieldUnits}

get_heater_output(output)

Sample heater output in percent, scale is dependent upon the instrument used and heater configuration.

Args:

output (int):

Heater output to query.

Returns:

(float):

percent of full scale current/voltage/power.

get_heater_pid(output)

Returns the closed loop control parameters of the heater output.

Args:

output (int):

Specifies which output's control loop to query.

Returns:

(dict):

{"gain": float, "integral": float, "derivative": float}

gain: Proportional term in PID control. integral: Integral term in PID control. derivative: Derivative term in PID control.

get_heater_status(output)

Returns the heater error code state, error is cleared upon querying the heater status.

Args:

output (int):

Specifies which heater output to query (1 or 2).

Returns:

(TemperatureControllerEnums.HeaterError):

Object of instrument's HeaterError type.

get_ieee_488()

Returns the IEEE address set.

Returns:

(int):

Address 1-30 (0 and 31 reserved).

get_input_curve(input_channel)

Returns the curve number being used for a given input.

Args:

input_channel (str or int):

Specifies which input to query.

Returns:**(int):**

Curve number (0-59).

get_kelvin_reading(*input_channel*)

Returns the temperature value in kelvin of the given channel.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:**float:**

The temperature value.

get_keypad_lock()

Returns the state of the keypad lock and the lock-out code.

Returns:**(dict[str, bool | int]):**

{“state”: bool, “code”: int}

get_led_state()

Returns whether front panel LEDs are enabled.

Returns:**(bool):**

Specifies whether front panel LEDs are functional. False if disabled, True enabled.

get_manual_output(*output*)

Returns the manual output value in percent.

Args:**output (int):**

Specifies output to query.

Returns:**(float):**

Manual output percent.

get_min_max_data(*input_channel*)

Returns the minimum and maximum data from an input.

Args:**input_channel (str):**

Specifies which input to query.

Returns:**(dict[str, float]):**

{“minimum”: float, “maximum”: float}

get_relay_alarm_control_parameters(*relay_number*)

Returns the relay alarm configuration for either of the two configurable relays.

Relay must be configured for alarm mode to retrieve parameters.

Args:

relay_number (int)

Specifies which relay to query. Options are: 1 or 2.

Returns:

(dict[str, str | LegacyTemperatureControllerEnums.RelayControlAlarm]):

{“activating_input_channel”: str, “alarm_relay_trigger_type”: RelayControlAlarm}

get_relay_control_mode(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):

Specifies which relay to query. Options are: 1 or 2.

Returns:

(LegacyTemperatureControllerEnums.RelayControlMode):

The configured mode of the relay. Represented as a member of the instrument’s RelayControlMode IntEnum class.

get_relay_status(*relay_channel*)

Returns whether the relay at the specified channel is On or Off.

Args:

relay_channel (int):

The relay channel to query.

Returns:

(bool):

True if relay is on, False if relay is off.

get_remote_interface_mode()

Returns the state of the interface mode.

Returns:

(TemperatureControllerEnums.InterfaceMode):

A member of the instrument’s InterfaceMode IntEnum class.

get_self_test()

Instrument self test result completed at power up.

Returns:

(bool):

True = errors found. False = no errors found.

get_sensor_name(*input_channel*)

Returns the name of the sensor on the specified channel.

Args:

input_channel (str or int):

Specifies which input_channel channel to read from.

Returns:**name (str):**

Name associated with the sensor.

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor's units.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:**(float):**

The raw sensor reading in the units of the connected sensor.

get_service_request()

Returns the status byte register bits and their values as a class instance.

get_setpoint_ramp_parameter(*output*)

Returns the control loop parameters of a particular output.

Args:**output (int):**

Specifies which output's control loop to return.

Returns:**(dict[str, bool | float]):**

{“ramp_enable”: bool, “rate_value”: float}

get_setpoint_ramp_status(*output*)

Returns whether the set-point is ramping.

Args:**output (int):**

Specifies which output's control loop to query.

Returns:**(bool):**

Ramp status. False = Not ramping, True = Ramping.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

Returns:**StandardEventRegister:**

The object with the names of the standard event enable register bits and their values.

get_status_byte()

Returns the status flag bits as a class instance without resetting the register.

get_temperature_limit(*input_channel*)

Returns the value of the temperature limit in kelvin.

Args:

input_channel (str or int):

Specifies which input to query.

Returns:

float:

The value of the temperature limit.

query(*queries, check_errors=True)

Send a query to the instrument and return the response.

Args:

queries (str):

A serial query ending in a question mark.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

str:

The instrument query response as a string.

reset_alarm_status()

Clears the high and low status of all alarms.

reset_instrument()

Sets controller parameters to power-up settings.

reset_min_max_data()

Resets the minimum and maximum input data.

set_alarm_parameters(input_channel, alarm_enable, alarm_settings=None)

Configures the alarm parameters for an input.

Args:

input_channel (str):

Specifies which input to configure.

alarm_enable (bool):

Specifies whether to turn on the alarm for the input, or turn the alarm off.

alarm_settings (AlarmSettings):

See AlarmSettings class. Required only if alarm_enable is set to True.

set_control_setpoint(output, value)

Set set-point for specific output's control loop.

Control settings, that is, P, I, D, and Set-point, are assigned to outputs, which results in the settings being applied to the control loop formed by the output and its control input.

Args:

output (int):

Specifies which output's control loop to configure.

value (float):

The value for the set-point (in the preferred units of the control loop sensor).

set_curve(*curve, data_points*)

Method to define a user curve using a list of data points.

Args:

curve (int):

Specifies which curve to set.

data_points (list[tuple[float, float] | tuple[float, float, float]]):

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

set_curve_data_point(*curve, index, sensor_units, temperature, curvature=None*)

Configures a user curve point.

Args:

curve (int or str):

Specifies which curve to configure.

index (int):

Specifies the points index in the curve.

sensor_units (float):

Specifies sensor units for this point to 6 digits.

temperature (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

curvature (float):

Specify only if the point is part of a cubic spindle curve. The curvature value scale used to calculate spindle coefficients to 6 digits. Optional parameter.

set_curve_header(*curve_number, curve_header*)

Configures the user curve header.

Args:

curve_number (int):

Specifies which curve to configure.

curve_header (CurveHeader):

Instrument's CurveHeader class object containing the desired curve information.

set_display_field_settings(*field, input_channel, display_units*)

Configures a display field when the display is in custom mode.

Args:

field (int):

Defines which field of the display is being configured.

input_channel (LegacyTemperatureControllerEnums.InputChannel):

Defines which input to display. A member of the instrument's InputChannel IntEnum class.

display_units (LegacyTemperatureControllerEnums.DisplayFieldUnits):

Defines which units to display reading in. A member of the instrument's DisplayFieldUnits IntEnum class.

set_heater_pid(*output, gain, integral, derivative*)

Configure the closed loop control parameters of the heater output.

Args:

output (int):

Specifies which output's control loop to configure.

gain (float):

Proportional term in PID control. This controls how strongly the control output reacts to the present error.

integral (float):

Integral term in PID control. This controls how strongly the control output reacts to the past error history.

derivative (float):

Derivative term in PID control. This value controls how quickly the present field set point will transition to a new set-point.

set_ieee_488(*address*)

Specifies the IEEE address.

Args:

address (int):

1-30 (0 and 31 reserved).

set_input_curve(*input_channel*, *curve_number*)

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str or int):

Specifies which input to configure.

curve_number (int):

0 = none, 1-20 = standard curves, 21-59 = user curves.

set_keypad_lock(*state*, *code*)

Locks or unlocks front panel keypad (except for alarms and disabling heaters).

Args:

state (bool):

Sets the keypad to locked or unlocked. Options are: False for unlocked or True for locked.

code (int):

Specifies 3 digit lock-out code. Options are: 000 - 999.

set_led_state(*state*)

Sets the front panel LEDs to on or off.

Args:

state (bool):

Sets the LEDs to functional or nonfunctional. False if disabled, True enabled.

set_manual_output(*output*, *value*)

When instrument is in closed loop PID, Zone, or Open Loop modes a manual output may be set.

Args:

output (int):

Specifies output to configure.

value (float):

Specifies value for manual output in percent.

set_relay_alarms(*relay_number, activating_input_channel, alarm_relay_trigger_type*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:

relay_number (int):

The relay to configure. Options are: 1 or 2.

activating_input_channel (str or int):

Specifies which input alarm activates the relay.

alarm_relay_trigger_type (LegacyTemperatureControllerEnums.RelayControlAlarm):

Specifies the type of alarm that triggers the relay.

set_remote_interface_mode(*mode*)

Places the instrument in one of three interface modes.

Args:

mode (TemperatureControllerEnums.InterfaceMode):

A member of the instrument's InterfaceMode IntEnum class.

set_sensor_name(*input_channel, sensor_name*)

Sets a given name to a sensor on the specified channel.

Args:

input_channel (str or int):

Specifies which input_channel channel to read from.

sensor_name (str):

Name user wants to give to the sensor on the specified channel.

set_service_request(*register_mask*)

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:

register_mask (service_request_enable):

A service_request_enable class object with all bits configured.

set_setpoint_ramp_parameter(*output, ramp_enable, rate_value*)

Sets the control loop of a particular output.

Args:

output (int):

Specifies which output's control loop to configure.

ramp_enable (bool):

Specifies whether ramping is off or on (False = Off or True = On).

rate_value (float):

Specifies set-point ramp rate in kelvin per minute. The rate is always positive but will respond to ramps up or down. A rate of 0 is interpreted as infinite, and will respond as if set-point ramping were off. (0.1 to 100)

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits set to a value.

set_temperature_limit(input_channel, limit)

After a set temperature limit is exceeded, all control outputs will shut down.

Args:

input_channel (str or int):

Specifies which input to configure.

limit (float):

The temperature limit in kelvin for which to shut down all control outputs when exceeded. A limit of zero will turn the feature off.

turn_relay_off(relay_number)

Turns the specified relay off.

Args:

relay_number (int):

The relay to turn off. Options are: 1 or 2.

turn_relay_on(relay_number)

Turns the specified relay on.

Args:

relay_number (int):

The relay to turn on. Options are: 1 or 2.

write(command_string)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Settings classes

This section outlines the classes used to interact with methods which return or accept an argument of a class object, specific to the Lake Shore model 335.

class lakeshore.model_335.**Model335InputSensorSettings**(*sensor_type, autorange_enable, compensation, units, input_range=None*)

Class object used in the get/set_input_sensor methods.

Special-members

`__init__`

class lakeshore.model_335.**Model335ControlLoopZoneSettings**(*upper_bound, proportional, integral, derivative, manual_output_value, heater_range, channel, ramp_rate*)

Control loop configuration for a particular heater output and zone.

Special-members

`__init__`

Status register classes

This section describes the register objects. Each bit in the register is represented as a member of the register's class

```
class lakeshore.model_335.Model335StatusByteRegister(message_available_summary_bit,
                                                    event_status_summary_bit, service_request,
                                                    operation_summary_bit)
```

Class object representing the status byte register LSB to MSB.

```
class lakeshore.model_335.Model335ServiceRequestEnable(message_available_summary_bit,
                                                       event_status_summary_bit,
                                                       operation_summary_bit)
```

Class object representing the service request enable register LSB to MSB.

```
lakeshore.model_335.Model335StandardEventRegister
    alias of StandardEventRegister
```

```
class lakeshore.temperature_controllers.StandardEventRegister(operation_complete, query_error,
                                                                execution_error, command_error,
                                                                power_on)
```

Class object representing the standard event register.

```
lakeshore.model_335.Model335OperationEvent
    alias of OperationEvent
```

```
class lakeshore.temperature_controllers.OperationEvent(alarm, sensor_overload, loop_2_ramp_done,
                                                         loop_1_ramp_done, new_sensor_reading,
                                                         autotune_process_completed,
                                                         calibration_error,
                                                         processor_communication_error)
```

Class object representing the status byte register LSB to MSB.

```
class lakeshore.model_335.Model335InputReadingStatus(invalid_reading, temp_underrange,
                                                       temp_ouerrange, sensor_units_zero,
                                                       sensor_units_ouerrange)
```

Class object representing the input status flag bits.

Enumeration objects

This section describes the Enum type objects that have been created to name various settings of the Model 335 series that are represented as an int or single character to the instrument. The purpose of these enum types is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

```
class lakeshore.temperature_controllers_enums.TemperatureControllerEnums
```

Class containing the enums relevant to temperature controllers.

```
class InterfaceMode(*values)
```

Enumeration for the mode of the remote interface.

```
LOCAL = 0
```

```
REMOTE = 1
```

```
REMOTE_LOCAL_LOCK = 2
```

```
class HeaterError(*values)
```

Enumeration for possible errors flagged by the heater.

NO_ERROR = 0

HEATER_OPEN_LOAD = 1

HEATER_SHORT = 2

class CurveFormat(*values)

Enumerations specify formats for temperature sensor curves.

MILLIVOLT_PER_KELVIN = 1

VOLTS_PER_KELVIN = 2

OHMS_PER_KELVIN = 3

LOG_OHMS_PER_KELVIN = 4

class CurveTemperatureCoefficient(*values)

Enumerations specify positive/negative temperature sensor curve coefficients.

NEGATIVE = 1

POSITIVE = 2

class BrightnessLevel(*values)

Enumerator to specify the brightness level of an instrument display.

QUARTER = 0

HALF = 1

THREE_QUARTERS = 2

FULL = 3

class AutotuneMode(*values)

Enumerator used to represent the different autotune control modes.

P_ONLY = 0

P_I = 1

P_I_D = 2

class DiodeCurrent(*values)

Enumerator used to represent diode current ranges.

TEN_MICROAMPS = 0

ONE_MILLIAMP = 1

class HeaterOutputUnits(*values)

Enumerator used to represent heater output unit settings.

CURRENT = 1

POWER = 2

class Interface(*values)

Enumerator used to represent remote interface communication methods.

USB = 0

ETHERNET = 1

IEEE488 = 2

class InputSensorUnits(*values)

Enumerator used to represent temperature sensor unit options.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

class ControlTypes(*values)

Enumerator used to represent the control type settings.

AUTO_OFF = 0

CONTINUOUS = 1

class LanStatus(*values)

Represents the different status states for the lan connection.

STATIC_IP = 0

DHCP = 1

AUTO_IP = 2

ADDRESS_NOT_ACQUIRED_ERROR = 3

DUPLICATE_INITIAL_IP_ERROR = 4

DUPLICATE_ONGOING_IP_ERROR = 5

CABLE_UNPLUGGED = 6

MODULE_ERROR = 7

ACQUIRING_ADDRESS = 8

ETHERNET_DISABLED = 9

class DisplayFields(*values)

Enumeration of the possible number of fields to include in a custom display mode.

LARGE_2 = 0

LARGE_4 = 1

SMALL_8 = 2

class DisplayFieldsSize(*values)

Enumeration of the display fields when mode is set to all inputs.

SMALL = 0

LARGE = 1

```
class RelayControlMode(*values)
```

Relay operating mode enumeration.

```
RELAY_OFF = 0
```

```
RELAY_ON = 1
```

```
ALARMS = 2
```

```
class lakeshore.model_335.Model335Enums
```

Class containing the enums relevant to the Model 335.

```
class InputSensor(*values)
```

Enumeration when “NONE” is an option for sensor input.

```
NONE = 0
```

```
CHANNEL_A = 1
```

```
CHANNEL_B = 2
```

```
class MonitorOutUnits(*values)
```

Units associated with a sensor channel.

```
KELVIN = 1
```

```
CELSIUS = 2
```

```
SENSOR = 3
```

```
class InputSensorType(*values)
```

Sensor type enumeration.

```
DISABLED = 0
```

```
DIODE = 1
```

```
PLATINUM_RTD = 2
```

```
NTC_RTD = 3
```

```
THERMOCOUPLE = 4
```

```
class DiodeRange(*values)
```

Diode voltage range enumeration.

```
TWO_POINT_FIVE_VOLTS = 0
```

```
TEN_VOLTS = 1
```

```
class RTDRange(*values)
```

RTD resistance range enumeration.

```
TEN_OHM = 0
```

```
THIRTY_OHM = 1
```

```
HUNDRED_OHM = 2
```

```
THREE_HUNDRED_OHM = 3
```

```
ONE_THOUSAND_OHM = 4
THREE_THOUSAND_OHM = 5
TEN_THOUSAND_OHM = 6
THIRTY_THOUSAND_OHM = 7
ONE_HUNDRED_THOUSAND_OHM = 8
```

```
class ThermocoupleRange(*values)
    Thermocouple range enumeration.
```

```
FIFTY_MILLIVOLT = 0
```

```
class HeaterOutType(*values)
    Heater output 2 enumeration.
```

```
CURRENT = 0
```

```
VOLTAGE = 1
```

```
class HeaterOutputDisplay(*values)
    Heater output display units enumeration.
```

```
CURRENT = 1
```

```
POWER = 2
```

```
class HeaterOutputMode(*values)
    Control loop enumeration.
```

```
OFF = 0
```

```
CLOSED_LOOP = 1
```

```
ZONE = 2
```

```
OPEN_LOOP = 3
```

```
MONITOR_OUT = 4
```

```
WARMUP_SUPPLY = 5
```

```
class WarmupControl(*values)
    Heater output 2 voltage mode warmup enumerations.
```

```
AUTO_OFF = 0
```

```
CONTINUOUS = 1
```

```
class HeaterRange(*values)
    Control loop heater range enumeration.
```

```
OFF = 0
```

```
LOW = 1
```

```
MEDIUM = 2
```

HIGH = 3

class DisplaySetup(*values)

Panel display setup enumeration.

INPUT_A = 0

INPUT_A_MAX_MIN = 1

TWO_INPUT_A = 2

INPUT_B = 3

INPUT_B_MAX_MIN = 4

TWO_INPUT_B = 5

CUSTOM = 6

TWO_LOOP = 7

class HeaterVoltageRange(*values)

Voltage mode heater enumerations.

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class InputChannel(*values)

Panel display information enumeration.

NONE = 0

INPUT_A = 1

INPUT_B = 2

SETPOINT_1 = 3

SETPOINT_2 = 4

OUTPUT_1 = 5

OUTPUT_2 = 6

class DisplayFieldUnits(*values)

Panel display units enumeration.

KELVIN = 1

CELSIUS = 2

SENSOR_UNITS = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

SENSOR_NAME = 6

Model 336 Cryogenic Temperature Controller

The Model 336 measures and controls cryogenic temperature environments.

More information about the instrument can be found on our [website](#) including the manual which has a list of all commands and queries.

Example Scripts

Below are a few example scripts for the Model 336 that use the Lake Shore Python driver.

Using calibration curves with a temperature instrument

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and
↳serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a
↳temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↳VOLTS_PER_KELVIN, 300.0,
                                     myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set
↳to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.
↳The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,
↳30, "SN123", (276, 10),
                                                    (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can
↳then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that
↳curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Setting up heater outputs on the Model 336

```

from lakeshore import Model336

# Connect to the first available Model 336 temperature controller over USB with a baud_
↪rate of 57600
my_model_336 = Model336()

# Set the control loop values for outputs 1 and 3
my_model_336.set_heater_pid(1, 40, 27, 0)
my_model_336.set_heater_pid(3, 35, 20, 0)

# Configure heater output 1 with a 50 ohm load, 0.75 amp max current, and screen display_
↪mode to power
my_model_336.set_heater_setup(1, my_model_336.HeaterResistance.HEATER_50_OHM, 0.75, my_
↪model_336.HeaterOutputUnits.POWER)

# Configure analog heater output 3 to monitor sensor channel A, a high and low value of_
↪3.65 and 1.02 kelvin
# respectively as a unipolar output
my_model_336.set_monitor_output_heater(3, my_model_336.InputChannel.CHANNEL_A, my_model_
↪336.InputSensorUnits.KELVIN, 3.65, 1.02,
                                     my_model_336.Polarity.UNIPOLAR)

# Set closed loop output mode for heater 1
my_model_336.set_heater_output_mode(1, my_model_336.HeaterOutputMode.CLOSED_LOOP, my_
↪model_336.InputChannel.CHANNEL_A)

# Set closed loop output mode for heater 3
my_model_336.set_heater_output_mode(3, my_model_336.HeaterOutputMode.CLOSED_LOOP, my_
↪model_336.InputChannel.CHANNEL_B)

# Set a control setpoint for outputs 1 and 3 to 1.5 kelvin
my_model_336.set_control_setpoint(1, 1.5)
my_model_336.set_control_setpoint(3, 2.5)

# Turn the heaters on by setting the heater range
my_model_336.set_heater_range(1, my_model_336.HeaterRange.MEDIUM)
my_model_336.set_heater_range(3, my_model_336.HeaterVoltageRange.VOLTAGE_ON)

# Obtain the output percentage of output 1 and print it to the console
heater_one_output = my_model_336.get_heater_output(1)
print("Output 1: " + str(heater_one_output))

# Obtain the output percentage of output 3 and print it to the console
heater_three_output = my_model_336.get_analog_output_percentage(3)
print("Output 3: " + str(heater_three_output))

```

Instrument class methods

```

class lakeshore.model_336.Model336(serial_number=None, com_port=None, timeout=2.0,
                                   ip_address=None, tcp_port=7777, **kwargs)

```

A class object representing the Lake Shore Model 336 cryogenic temperature controller.

status_byte_register

alias of *Model336StatusByteRegister*

service_request_enable

alias of *Model336ServiceRequestEnable*

get_analog_output_percentage(*output*)

Returns the output percentage of the analog voltage output.

Args:**output (int):**

Specifies which analog voltage output to query.

Returns:**(float):**

Analog voltage heater output percentage.

set_autotune(*output, mode*)

Initiates auto-tuning of the heater control loop.

Args:**output (int):**

Specifies the output associated with the loop to be Auto-tuned.

mode (TemperatureControllerEnums.AutotuneMode):

Specifies the Autotune mode. Member of instrument's AutotuneMode IntEnum class.

set_contrast_level(*contrast_level*)

Sets the display contrast level on the front panel.

Args:**contrast_level (int):**

Contrast value: 1 - 32.

get_contrast_level()

Returns the contrast level of front display.

Returns:**int:**

The contrast level of front display.

get_operation_condition()

Returns the names of the operation condition register bits and their values.

Returns:**OperationEvent:**

The object with the names of the operation condition register bits and their values.

get_operation_event_enable()

Returns the names of the operation event enable register and their values.

These values determine which bits propagate to the operation condition register.

Returns:**OperationEvent:**

The object with the names of the operation event enable register and their values.

set_operation_event_enable(*register_mask*)

Configures values of the operation event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (OperationEvent):

An OperationEvent class object with all bits configured true or false.

get_operation_event()

Returns the names of the operation event register bits and their values.

Returns:

OperationEvent:

The object with the names of the operation event register bits and their values.

get_thermocouple_junction_temp()

Returns the temperature of the ceramic thermocouple block from the room temperature compensation calculation.

Returns:

(float):

Temperature of the ceramic thermocouple block (kelvin).

set_soft_cal_curve_dt_470(*curve_number, serial_number, calibration_point_1=(4.2, 1.62622), calibration_point_2=(77.35, 1.02032), calibration_point_3=(305, 0.50691)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured DT-470 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_soft_cal_curve_pt_100(*curve_number, serial_number, calibration_point_1=(77.35, 20.234), calibration_point_2=(305, 112.384), calibration_point_3=(480, 178.353)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured PT-100 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_soft_cal_curve_pt_1000(*curve_number, serial_number, calibration_point_1=(77.35, 202.34), calibration_point_2=(305, 1123.84), calibration_point_3=(480, 1783.53)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured PT-1000 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:**curve_number (int):**

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_2 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple[float, float]):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

set_filter(*input_channel, filter_enable, data_points, reset_threshold*)

Configures the input_channel filter parameter.

Args:**input_channel (int or str):**

Specifies which input channel to configure.

filter_enable (bool):

Specified whether the filtering function is enabled or not.

data_points (int):

Specifies how many points the filter function uses: 2 - 64.

reset_threshold (int):

Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets. Options are: 1% - 10%.

get_filter(*input_channel*)

Returns the input_channel filter configuration.

Args:**input_channel (int or str):**

Specifies which input channel to configure.

Returns:

(dict[str, bool | int])

{“filter_enable”: bool, “data_points”: int, “reset_threshold”: int}

filter_enable: Specified whether the filtering function is enabled or not. data_points: Specifies how many points the filter function uses. reset_threshold: Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets (1% - 10%).

set_network_settings(*dhcp_enable, auto_ip_enable, ip_address, sub_mask, gateway, primary_dns, secondary_dns, pref_host, pref_domain, description*)

Network class constructor.

Args:

dhcp_enable (bool):

Enable or disable DHCP.

auto_ip_enable (bool):

Enable or disable dynamically configured link-local addressing (Auto IP).

ip_address (str):

IP address for static configuration.

sub_mask (str):

Subnet mask for static configuration.

gateway (str):

Gateway address for static configuration.

primary_dns (str):

Primary DNS address for static configuration.

secondary_dns (str):

Secondary DNS address for static configuration.

pref_host (str):

Preferred Hostname (15 character maximum).

pref_domain (str):

Preferred Domain name (64 character maximum).

description (str):

Instrument description (32 character maximum).

get_network_settings()

Method to retrieve the IP settings.

Returns:

(dict[str, bool | str]):

See set_network_settings arguments.

get_network_configuration()

Method to return the configured ethernet parameters.

Return:

(dict[str, TemperatureControllerEnums.LanStatus | str]):

{“lan_status”: LanStatus, “ip_address”: str, “sub_mask”: str, “gateway”: str,

“primary_dns”: str,

“secondary_dns”: str, “hostname” str, “domain”: str, “mac_address”: str} lan_status: Current status of the ethernet connection. Member of the instrument’s LanStatus IntEnum class.

ip_address: Configured IP address. sub_mask: Configured subnet mask. gateway: Configured gateway address. primary_dns: Configured primary DNS address. secondary_dns: Configured secondary DNS address. hostname: Assigned hostname. domain: Assigned domain. mac_address: Module MAC address.

set_website_login(*username, password*)

Sets the username and password for the web interface.

Args:

username (str):

15 character string representing the website username.

password (str):

15 character string representing the website password.

get_website_login()

Method to return the username and password for the web interface.

Returns:

(dict[str, str]): A dictionary containing 15 character string type items.

{“username”: str, “password”: str}

get_celsius_reading(*channel*)

Returns the temperature in Celsius of any channel.

Args:

channel (str):

Specifies which channel to read from: “A” - “H4”.

Returns:

float:

The temperature in Celsius.

set_interface(*interface*)

Selects the remote interface for the instrument,

Args:

interface (TemperatureControllerEnums.Interface):

Member of instrument’s Interface IntEnum class,

get_interface()

Returns the remote interface for the instrument.

Returns:

(TemperatureControllerEnums.Interface):

Member of instrument’s Interface IntEnum class.

get_tuning_control_status()

Returns dictionary of tuning control status values.

If initial conditions are not met when starting the autotune procedure, causing the auto-tuning process to never actually begin, then the error status will be set to 1 and the stage status will be stage 00.

Returns:

(dict[str, bool | int]):

```
{“active_tuning_enable”: bool, “output”: int, “tuning_error”: bool, “stage_status”: int}
  active_tuning_enable: False = no active tuning, True = active tuning.
  output: Heater output of the control loop being tuned.
  tuning_error: False = no tuning error, True = tuning error.
  stage_status: Specifies the current stage in the Autotune process. If tuning error occurred, stage status represents stage that failed.
```

set_diode_excitation_current(*channel*, *excitation_current*)

Sets the excitation current of a specific channel.

The 10 uA excitation current is the only calibrated excitation current, and is used in almost all applications. The Model 336 will default the 10 uA current setting any time the input sensor type is changed.

Args:

channel (str):

Specifies which sensor input to configure: “A” - “D”.

excitation_current (TemperatureControllerEnums.DiodeCurrent):

A member of the instrument’s DiodeCurrent IntEnum class.

get_diode_excitation_current(*channel*)

Returns the diode excitation current setting as a string.

Args:

channel (str):

Specifies which input to return: “A” - “D”.

Returns:

(TemperatureControllerEnums.DiodeCurrent):

A member of the instrument’s DiodeCurrent IntEnum class. Diode excitation current.

set_monitor_output_heater(*output*, *channel*, *units*, *high_value*, *low_value*, *polarity*)

Configures a voltage-controlled output.

Use the set_heater_output_mode command to set the output mode to Monitor Out.

Args:

output (int):

Voltage-controlled output to configure (3 or 4)

channel (InputChannel):

Specifies which sensor input to monitor. A member of the InputChannel IntEnum class.

units (self.InputSensorUnits):

Specifies the units on which to base the output voltage. A member of the self.InputSensorUnits IntEnum class.

high_value (float):

Represents the data at which the Monitor Out reaches +100% output. Entered in the units designated by the <units> argument.

low_value (float):

Represents the data at which the analog output reaches -100% output if bipolar, or 0% output if unipolar. Entered in the units designated by the <units> argument.

polarity (self.Polarity):

Specifies whether the output voltage is unipolar or bipolar. Member of the self.Polarity IntEnum class.

get_monitor_output_heater(*output*)

Used to obtain all monitor out parameters for a specific output.

Args:**output (int):**

Voltage-controlled output to configure (3 or 4).

Returns:**(dict):**

See set_monitor_output_heater arguments

set_display_setup(*mode*, *num_fields=""*, *displayed_output=""*)

Sets the display mode.

Args:**mode (self.DisplaySetupMode):**

Member of self.DisplaySetupMode IntEnum class Specifies display mode for default and 3062 options

num_fields (IntEnum)

When mode is set to custom, specifies the number of fields (Member of self.DisplayFields). When mode is set to all inputs, specifies size of readings (Member of self.DisplayFieldsSize).

displayed_output (int):

Configures the bottom half of the custom display screen. Only required if mode is set to CUSTOM.
Output: (1 - 4)

get_display_setup()

Returns the display mode.

Returns:**(dict):**

See set_display_setup method arguments. Keys: "mode", "num_fields", "displayed_output"

set_heater_setup(*output*, *heater_resistance*, *max_current*, *heater_output*)

Method to configure the heaters.

Args:**output (int):**

Specifies which heater output to configure (1 or 2).

heater_resistance (self.HeaterResistance):

Member of self.HeaterResistance IntEnum class.

max_current (float):

User defined maximum output current (see table 4-11 for max current and resistance relationships).

heater_output (self.HeaterOutputUnits):

Specifies whether the heater output displays in current or power. Member of self.HeaterOutputUnits IntEnum class.

get_heater_setup(*heater_output*)

Returns the heater configuration status.

Args:**heater_output (int):**

Specifies which heater output to configure (1 or 2)

Returns:

(dict):

See set_heater_setup arguments Keys: heater_resistance, max_current, output_display_mode.

set_input_sensor(*channel, sensor_parameters*)

Sets the sensor type and associated parameters.

Args:

channel (str):

Specifies input to configure (“A” - “D”): 3062 option (“D1” - “D5”)

sensor_parameters (Model336InputSensorSettings):

See Model336InputSensorSettings class.

get_input_sensor(*channel*)

Returns the sensor type and associated parameters.

Args:

channel (str):

Specifies sensor input to configure (“A” or “B”)

Returns:

(Model336InputSensorSettings):

See Model336InputSensorSettings class.

get_all_kelvin_reading()

Returns the temperature value in kelvin of all channels.

Returns:

(list: float):

[channel_A, channel_B, channel_C, channel_D]

set_heater_output_mode(*output, mode, channel, powerup_enable=False*)

Configures the heater output mode.

Args:

output (int):

Specifies which output to configure (1 - 4)

mode (self.HeaterOutputMode):

Member of self.HeaterOutputMode IntEnum class. Specifies the control mode.

channel (InputChannel):

InputChannel IntEnum class. Specifies which input to use for control.

powerup_enable (bool):

Specifies whether the output remains on (True) or shuts off after power cycle (False).

get_heater_output_mode(*output*)

Returns the heater output mode for a given output and whether powerup is enabled.

Args:

output (int):

Specifies which output to retrieve (1 - 4).

Returns:

(dict):

See `set_heater_output_mode` method arguments. Keys: `mode`, `channel`, `powerup_enable`.

set_heater_range(*output*, *heater_range*)

Sets the heater range for a particular output.

The range setting has no effect if an output is in the Off mode, and does not apply to an output in Monitor Out mode. An output in Monitor Out mode is always on.

Args:

output (int):

Specifies which output to configure (1 - 4).

heater_range (IntEnum):

For Outputs 1 and 2: Member of `self.HeaterRange` IntEnum class. For Outputs 3 and 4: `self.HeaterVoltageRange` IntEnum class.

get_heater_range(*output*)

Returns the heater range for a particular output.

Args:

output (int):

Specifies which output to query (1 or 2).

Returns:

(IntEnum):

For Outputs 1 and 2: Member of `self.HeaterRange` IntEnum class. For Outputs 3 and 4: Member of `self.HeaterVoltageRange` IntEnum class.

all_heaters_off()

Recreates the front panel safety feature of shutting off all heaters.

get_input_reading_status(*channel*)

Reruns the state of the input status flag bits.

Args:

channel (str):

Specifies which channel to query ("A" - "D"). Use "D1" - "D5" for 3062 option.

Returns:

(Model336InputReadingStatus):

Boolean representation of each bit in the input status flag register.

get_all_sensor_reading()

Returns the sensor unit reading of all channels.

Returns:

(list: float):

[`channel_A`, `channel_B`, `channel_C`, `channel_D`]

set_warmup_supply_parameter(*output*, *control*, *percentage*)

Warmup mode applies only to voltage heater outputs 3 and 4.

The Output mode and Control Input parameters must be configured using the `set_monitor_out_parameters()` method.

Args:

output (int):

Specifies which output to configure (3 or 4).

control (self.ControlTypes):

Member of the self.ControlTypes IntEnum class.

percentage (float):

Specifies the percentage of full scale (10 V) Monitor Out voltage to apply to turn on the external power supply. (A value of 50.5 translates to a 50.5 percent output voltage).

get_warmup_supply_parameter(*output*)

Returns the warmup supply configuration for a particular output.

Args:

output (int):

Specifies which analog voltage heater output to retrieve (3 or 4).

Returns:

(dict):

See set_warmup_supply_parameter method arguments

set_control_loop_zone_table(*output, zone, control_loop_zone*)

Configures the output zone parameters.

Args:

output (int):

Specifies which analog voltage heater output to configure (1 or 2).

zone (int):

Specifies which zone in the table to configure (1 to 10).

control_loop_zone (Model336ControlLoopZoneSettings):

See Model336ControlLoopZoneSettings class.

get_control_loop_zone_table(*output, zone*)

Returns a list of zone control parameters for a selected output and zone.

Args:

output (int):

Specifies which heater output to query (1 or 2).

zone (int):

Specifies which zone in the table to query (1 to 10).

Returns:

(Model336ControlLoopZoneSettings):

See Model336ControlLoopZoneSettings class.

class HeaterResistance(values*)**

Enumerator used to represent the different heater resistances.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class Polarity(*values)

Enumerator for unipolar or bipolar output operation.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class RelayControlAlarm(*values)

Enumeration of the setting determining which alarm(s) cause a relay to close in alarm mode.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

clear_interface_command()

Clears the bits in the SBR, SESR, OER, and terminates all operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register. Terminates all pending operations. Clears the interface, but not the controller.

command(*commands, check_errors=True)

Sends an SCPI command or multiple commands to the instrument.

Args:**commands (str):**

A serial command.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

delete_curve(*curve*)

Deletes the user curve.

Args:

curve (int):
Specifies a user curve to delete.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

get_alarm_parameters(*input_channel*)

Returns the present state of all alarm parameters.

Args:

input_channel (str):
Specifies which input to configure.

Returns:

alarm_settings (AlarmSettings):
See AlarmSettings class.

get_alarm_status(*channel*)

Returns the high state and low state of the alarm for the specified channel.

Args:

channel (str or int)
Specifies which input channel to read from.

Returns:

(dict[str, bool]):
{“high_state_enabled”: bool, “low_state_enabled” bool}

get_control_setpoint(*output*)

Returns the value for a given control output.

Args:

output (int):
Specifies which output’s control loop to query (1 or 2).

Returns:

(float):
The value for the set-point (in the preferred units of the control loop sensor).

get_curve(*curve*)

Returns a list of all the data points in a particular curve.

Args:

curve (int):
Specifies which curve to set.

Returns:

list[tuple[float, float] | tuple[float, float, float]]:

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_data_point(*curve*, *index*)

Returns a standard or user curve data point.

Args:

curve (int):

Specifies which curve to query.

index (int):

Specifies the points index in the curve.

Returns:

tuple[float, float] | tuple[float, float, float]:

(sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_header(*curve_number*)

Returns parameters set on a particular user curve header.

Args:

curve_number (int):

Specifies a curve to retrieve.

Returns:

(CurveHeader):

A CurveHeader class object containing the curve information.

get_display_field_settings(*field*)

Returns the settings of the specified display field when display is in Custom mode.

Args:

field (int)

Defines which field of the display to retrieve settings from.

Returns:

(dict[str, LegacyTemperatureControllerEnums.InputChannel |

LegacyTemperatureControllerEnums.DisplayFieldUnits]):

See set_display_field_settings method. {"input_channel": InputChannel, "display_units": DisplayFieldUnits}

get_heater_output(*output*)

Sample heater output in percent, scale is dependent upon the instrument used and heater configuration.

Args:

output (int):

Heater output to query.

Returns:

(float):

percent of full scale current/voltage/power.

get_heater_pid(*output*)

Returns the closed loop control parameters of the heater output.

Args:

output (int):
Specifies which output's control loop to query.

Returns:

(dict):
{**"gain": float, "integral": float, "derivative": float**}
gain: Proportional term in PID control. integral: Integral term in PID control. derivative:
Derivative term in PID control.

get_heater_status(*output*)

Returns the heater error code state, error is cleared upon querying the heater status.

Args:

output (int):
Specifies which heater output to query (1 or 2).

Returns:

(TemperatureControllerEnums.HeaterError):
Object of instrument's HeaterError type.

get_ieee_488()

Returns the IEEE address set.

Returns:

(int):
Address 1-30 (0 and 31 reserved).

get_input_curve(*input_channel*)

Returns the curve number being used for a given input.

Args:

input_channel (str or int):
Specifies which input to query.

Returns:

(int):
Curve number (0-59).

get_kelvin_reading(*input_channel*)

Returns the temperature value in kelvin of the given channel.

Args:

input_channel (str | int):
Selects the channel to retrieve reading.

Returns:

float:
The temperature value.

get_keypad_lock()

Returns the state of the keypad lock and the lock-out code.

Returns:

(dict[str, bool | int]):
 {"state": bool, "code": int}

get_led_state()

Returns whether front panel LEDs are enabled.

Returns:

(bool):
 Specifies whether front panel LEDs are functional. False if disabled, True enabled.

get_manual_output(*output*)

Returns the manual output value in percent.

Args:

output (int):
 Specifies output to query.

Returns:

(float):
 Manual output percent.

get_min_max_data(*input_channel*)

Returns the minimum and maximum data from an input.

Args:

input_channel (str):
 Specifies which input to query.

Returns:

(dict[str, float]):
 {"minimum": float, "maximum": float}

get_relay_alarm_control_parameters(*relay_number*)

Returns the relay alarm configuration for either of the two configurable relays.

Relay must be configured for alarm mode to retrieve parameters.

Args:

relay_number (int)
 Specifies which relay to query. Options are: 1 or 2.

Returns:

(dict[str, str | LegacyTemperatureControllerEnums.RelayControlAlarm]):
 {"activating_input_channel": str, "alarm_relay_trigger_type": RelayControlAlarm}

get_relay_control_mode(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):
 Specifies which relay to query. Options are: 1 or 2.

Returns:

(LegacyTemperatureControllerEnums.RelayControlMode):
 The configured mode of the relay. Represented as a member of the instrument's RelayControlMode IntEnum class.

get_relay_status(*relay_channel*)

Returns whether the relay at the specified channel is On or Off.

Args:

relay_channel (int):
The relay channel to query.

Returns:

(bool):
True if relay is on, False if relay is off.

get_remote_interface_mode()

Returns the state of the interface mode.

Returns:

(TemperatureControllerEnums.InterfaceMode):
A member of the instrument's InterfaceMode IntEnum class.

get_self_test()

Instrument self test result completed at power up.

Returns:

(bool):
True = errors found. False = no errors found.

get_sensor_name(*input_channel*)

Returns the name of the sensor on the specified channel.

Args:

input_channel (str or int):
Specifies which input_channel channel to read from.

Returns:

name (str):
Name associated with the sensor.

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor's units.

Args:

input_channel (str | int):
Selects the channel to retrieve reading.

Returns:

(float):
The raw sensor reading in the units of the connected sensor.

get_service_request()

Returns the status byte register bits and their values as a class instance.

get_setpoint_ramp_parameter(*output*)

Returns the control loop parameters of a particular output.

Args:

output (int):

Specifies which output's control loop to return.

Returns:**(dict[str, bool | float]):**

{“ramp_enable”: bool, “rate_value”: float}

get_setpoint_ramp_status(output)

Returns whether the set-point is ramping.

Args:**output (int):**

Specifies which output's control loop to query.

Returns:**(bool):**

Ramp status. False = Not ramping, True = Ramping.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

Returns:**StandardEventRegister:**

The object with the names of the standard event enable register bits and their values.

get_status_byte()

Returns the status flag bits as a class instance without resetting the register.

get_temperature_limit(input_channel)

Returns the value of the temperature limit in kelvin.

Args:**input_channel (str or int):**

Specifies which input to query.

Returns:**float:**

The value of the temperature limit.

query(*queries, check_errors=True)

Send a query to the instrument and return the response.

Args:**queries (str):**

A serial query ending in a question mark.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:**str:**

The instrument query response as a string.

reset_alarm_status()

Clears the high and low status of all alarms.

reset_instrument()

Sets controller parameters to power-up settings.

reset_min_max_data()

Resets the minimum and maximum input data.

set_alarm_parameters(*input_channel*, *alarm_enable*, *alarm_settings=None*)

Configures the alarm parameters for an input.

Args:

input_channel (str):

Specifies which input to configure.

alarm_enable (bool):

Specifies whether to turn on the alarm for the input, or turn the alarm off.

alarm_settings (AlarmSettings):

See AlarmSettings class. Required only if alarm_enable is set to True.

set_control_setpoint(*output*, *value*)

Set set-point for specific output's control loop.

Control settings, that is, P, I, D, and Set-point, are assigned to outputs, which results in the settings being applied to the control loop formed by the output and its control input.

Args:

output (int):

Specifies which output's control loop to configure.

value (float):

The value for the set-point (in the preferred units of the control loop sensor).

set_curve(*curve*, *data_points*)

Method to define a user curve using a list of data points.

Args:

curve (int):

Specifies which curve to set.

data_points (list[tuple[float, float] | tuple[float, float, float]]):

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

set_curve_data_point(*curve*, *index*, *sensor_units*, *temperature*, *curvature=None*)

Configures a user curve point.

Args:

curve (int or str):

Specifies which curve to configure.

index (int):

Specifies the points index in the curve.

sensor_units (float):

Specifies sensor units for this point to 6 digits.

temperature (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

curvature (float):

Specify only if the point is part of a cubic spindle curve. The curvature value scale used to calculate spindle coefficients to 6 digits. Optional parameter.

set_curve_header(*curve_number*, *curve_header*)

Configures the user curve header.

Args:**curve_number (int):**

Specifies which curve to configure.

curve_header (CurveHeader):

Instrument's CurveHeader class object containing the desired curve information.

set_display_field_settings(*field*, *input_channel*, *display_units*)

Configures a display field when the display is in custom mode.

Args:**field (int):**

Defines which field of the display is being configured.

input_channel (LegacyTemperatureControllerEnums.InputChannel):

Defines which input to display. A member of the instrument's InputChannel IntEnum class.

display_units (LegacyTemperatureControllerEnums.DisplayFieldUnits):

Defines which units to display reading in. A member of the instrument's DisplayFieldUnits IntEnum class.

set_heater_pid(*output*, *gain*, *integral*, *derivative*)

Configure the closed loop control parameters of the heater output.

Args:**output (int):**

Specifies which output's control loop to configure.

gain (float):

Proportional term in PID control. This controls how strongly the control output reacts to the present error.

integral (float):

Integral term in PID control. This controls how strongly the control output reacts to the past error history.

derivative (float):

Derivative term in PID control. This value controls how quickly the present field set point will transition to a new set-point.

set_ieee_488(*address*)

Specifies the IEEE address.

Args:**address (int):**

1-30 (0 and 31 reserved).

set_input_curve(*input_channel*, *curve_number*)

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str or int):

Specifies which input to configure.

curve_number (int):

0 = none, 1-20 = standard curves, 21-59 = user curves.

set_keypad_lock(*state*, *code*)

Locks or unlocks front panel keypad (except for alarms and disabling heaters).

Args:

state (bool):

Sets the keypad to locked or unlocked. Options are: False for unlocked or True for locked.

code (int):

Specifies 3 digit lock-out code. Options are: 000 - 999.

set_led_state(*state*)

Sets the front panel LEDs to on or off.

Args:

state (bool):

Sets the LEDs to functional or nonfunctional. False if disabled, True enabled.

set_manual_output(*output*, *value*)

When instrument is in closed loop PID, Zone, or Open Loop modes a manual output may be set.

Args:

output (int):

Specifies output to configure.

value (float):

Specifies value for manual output in percent.

set_relay_alarms(*relay_number*, *activating_input_channel*, *alarm_relay_trigger_type*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:

relay_number (int):

The relay to configure. Options are: 1 or 2.

activating_input_channel (str or int):

Specifies which input alarm activates the relay.

alarm_relay_trigger_type (LegacyTemperatureControllerEnums.RelayControlAlarm):

Specifies the type of alarm that triggers the relay.

set_remote_interface_mode(*mode*)

Places the instrument in one of three interface modes.

Args:

mode (TemperatureControllerEnums.InterfaceMode):

A member of the instrument's InterfaceMode IntEnum class.

set_sensor_name(*input_channel*, *sensor_name*)

Sets a given name to a sensor on the specified channel.

Args:

input_channel (str or int):

Specifies which input_channel channel to read from.

sensor_name (str):

Name user wants to give to the sensor on the specified channel.

set_service_request(*register_mask*)

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:

register_mask (service_request_enable):

A service_request_enable class object with all bits configured.

set_setpoint_ramp_parameter(*output*, *ramp_enable*, *rate_value*)

Sets the control loop of a particular output.

Args:

output (int):

Specifies which output's control loop to configure.

ramp_enable (bool):

Specifies whether ramping is off or on (False = Off or True = On).

rate_value (float):

Specifies set-point ramp rate in kelvin per minute. The rate is always positive but will respond to ramps up or down. A rate of 0 is interpreted as infinite, and will respond as if set-point ramping were off. (0.1 to 100)

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits set to a value.

set_temperature_limit(*input_channel*, *limit*)

After a set temperature limit is exceeded, all control outputs will shut down.

Args:

input_channel (str or int):

Specifies which input to configure.

limit (float):

The temperature limit in kelvin for which to shut down all control outputs when exceeded. A limit of zero will turn the feature off.

turn_relay_off(*relay_number*)

Turns the specified relay off.

Args:

relay_number (int):

The relay to turn off. Options are: 1 or 2.

turn_relay_on(*relay_number*)

Turns the specified relay on.

Args:

relay_number (int):

The relay to turn on. Options are: 1 or 2.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Settings classes

This section outlines the classes used to interact with methods which return or accept an argument of a class object, specific to the Lake Shore model 336.

```
class lakeshore.model_336.Model336InputSensorSettings(sensor_type, autorange_enable,  
compensation, units, input_range=None)
```

Class object used in the get/set_input_sensor methods.

```
__init__(sensor_type, autorange_enable, compensation, units, input_range=None)
```

Constructor for the InputSensorSettings class.

Args:

sensor_type (self.InputSensorType):

Specifies input sensor type

autorange_enable (bool):

Specifies auto-ranging (False = off, True = on)

compensation (bool):

Specifies input compensation (False = off, True = on)

units (self.InputSensorUnits):

Specifies the preferred units parameter for sensor readings and for the control set-point.

input_range (IntEnum)

Specifies input range if autorange_enable is false. See IntEnum classes: self.DiodeRange, self.RTDRange, and self.ThermocoupleRange.

```
class lakeshore.model_336.Model336ControlLoopZoneSettings(upper_bound, proportional, integral,  
derivative, manual_out_value,  
heater_range, channel, rate)
```

Control loop configuration for a particular heater output and zone.

```
__init__(upper_bound, proportional, integral, derivative, manual_out_value, heater_range, channel, rate)
```

Constructor.

Args:

upper_bound (float):

Specifies the upper set-point boundary of this zone in kelvin.

proportional (float):

Specifies the proportional gain for this zone (0.1 to 1000).

integral (float):

Specifies the integral gain for this zone (0.1 to 1000).

derivative (float):

Specifies the derivative gain for this zone (0 to 200 %).

manual_out_value (float):

Specifies the manual output for this zone (0 to 100 %).

heater_range (self.HeaterRange):

Specifies the heater range for this zone. See self.HeaterRange IntEnum class.

channel (InputChannel):

See InputChannel IntEnum class. Passing the NONE member will use the previously assigned sensor.

rate (float):

Specifies the ramp rate for this zone (0 - 100 K/min).

`lakeshore.model_336.Model336AlarmSettings`

alias of *AlarmSettings*

class `lakeshore.temperature_controllers.AlarmSettings`(*high_value, low_value, deadband, latch_enable, audible=None, visible=None, alarm_enable=None*)

Class used to disable or configure an alarm in conjunction with the `set/get_alarm_parameters()` method.

__init__(*high_value, low_value, deadband, latch_enable, audible=None, visible=None, alarm_enable=None*)

Constructor for AlarmSettings class.

Args:**high_value (float):**

Sets the value the source is checked against to activate the high alarm.

low_value (float):

Sets the value the source is checked against to activate low alarm.

deadband (float):

Sets the value that the source must change outside an alarm. condition to deactivate an unlatched alarm.

latch_enable (bool):

Specifies a latched alarm. False = off, True = on

audible (bool):

Specifies if the internal speaker will beep when an alarm condition occurs. False = off, True = on

visible (bool):

Specifies if the Alarm LED on the instrument front panel will blink when an alarm condition occurs. False = off, True = on

`lakeshore.model_336.Model336CurveHeader`

alias of *CurveHeader*

class `lakeshore.temperature_controllers.CurveHeader`(*curve_name, serial_number, curve_data_format, temperature_limit, coefficient*)

A class to configure the temperature sensor curve header parameters.

`__init__(curve_name, serial_number, curve_data_format, temperature_limit, coefficient)`

Constructor for CurveHeader class.

Args:

curve_name (str):

Specifies curve name (limit of 15 characters).

serial_number (str):

Specifies curve serial number (limit of 10 characters).

curve_data_format (TemperatureControllerEnums.CurveFormat):

Member of the instrument's CurveFormat IntEnum class. Specifies the curve data format.

temperature_limit (float):

Specifies the curve temperature limit in Kelvin.

coefficient (TemperatureControllerEnums.CurveTemperatureCoefficient):

Member of instrument's CurveTemperatureCoefficient IntEnum class. Specifies the curve temperature coefficient.

Status Register Classes

This page describes the register objects. Each bit in the register is represented as a member of the register's class

`lakeshore.model_336.Model336StandardEventRegister`

alias of `StandardEventRegister`

```
class lakeshore.model_336.Model336StatusByteRegister(message_available_summary_bit,
                                                    event_status_summary_bit, service_request,
                                                    operation_summary_bit)
```

Class object representing the status byte register LSB to MSB.

```
bit_names = ['', '', '', '', 'message_available_summary_bit',
             'event_status_summary_bit', 'service_request', 'operation_summary_bit']
```

```
class lakeshore.model_336.Model336ServiceRequestEnable(message_available_summary_bit,
                                                       event_status_summary_bit,
                                                       operation_summary_bit)
```

Class object representing the service request enable register LSB to MSB.

```
bit_names = ['', '', '', '', 'message_available_summary_bit',
             'event_status_summary_bit', '', 'operation_summary_bit']
```

`lakeshore.model_336.Model336OperationEvent`

alias of `OperationEvent`

```
class lakeshore.model_336.Model336InputReadingStatus(invalid_reading, temp_underrange,
                                                    temp_ouerrange, sensor_units_zero,
                                                    sensor_units_ouerrange)
```

Class object representing the input status flag bits.

```
bit_names = ['invalid_reading', '', '', '', 'temp_underrange', 'temp_ouerrange',
             'sensor_units_zero', 'sensor_units_ouerrange']
```

Enumeration objects

This section describes the Enum type objects that have been created to name various settings of the Model 336 series that are represented as an int or single character to the instrument. The purpose of these enum types is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

class lakeshore.temperature_controllers_enums.**TemperatureControllerEnums**

Class containing the enums relevant to temperature controllers.

class **InterfaceMode**(*values)

Enumeration for the mode of the remote interface.

LOCAL = 0

REMOTE = 1

REMOTE_LOCAL_LOCK = 2

class **HeaterError**(*values)

Enumeration for possible errors flagged by the heater.

NO_ERROR = 0

HEATER_OPEN_LOAD = 1

HEATER_SHORT = 2

class **CurveFormat**(*values)

Enumerations specify formats for temperature sensor curves.

MILLIVOLT_PER_KELVIN = 1

VOLTS_PER_KELVIN = 2

OHMS_PER_KELVIN = 3

LOG_OHMS_PER_KELVIN = 4

class **CurveTemperatureCoefficient**(*values)

Enumerations specify positive/negative temperature sensor curve coefficients.

NEGATIVE = 1

POSITIVE = 2

class **BrightnessLevel**(*values)

Enumerator to specify the brightness level of an instrument display.

QUARTER = 0

HALF = 1

THREE_QUARTERS = 2

FULL = 3

class **AutotuneMode**(*values)

Enumerator used to represent the different autotune control modes.

P_ONLY = 0

P_I = 1

P_I_D = 2

class DiodeCurrent(*values)

Enumerator used to represent diode current ranges.

TEN_MICROAMPS = 0

ONE_MILLIAMP = 1

class HeaterOutputUnits(*values)

Enumerator used to represent heater output unit settings.

CURRENT = 1

POWER = 2

class Interface(*values)

Enumerator used to represent remote interface communication methods.

USB = 0

ETHERNET = 1

IEEE488 = 2

class InputSensorUnits(*values)

Enumerator used to represent temperature sensor unit options.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

class ControlTypes(*values)

Enumerator used to represent the control type settings.

AUTO_OFF = 0

CONTINUOUS = 1

class LanStatus(*values)

Represents the different status states for the lan connection.

STATIC_IP = 0

DHCP = 1

AUTO_IP = 2

ADDRESS_NOT_ACQUIRED_ERROR = 3

DUPLICATE_INITIAL_IP_ERROR = 4

DUPLICATE_ONGOING_IP_ERROR = 5

CABLE_UNPLUGGED = 6

```
MODULE_ERROR = 7
```

```
ACQUIRING_ADDRESS = 8
```

```
ETHERNET_DISABLED = 9
```

```
class DisplayFields(*values)
```

Enumeration of the possible number of fields to include in a custom display mode.

```
LARGE_2 = 0
```

```
LARGE_4 = 1
```

```
SMALL_8 = 2
```

```
class DisplayFieldsSize(*values)
```

Enumeration of the display fields when mode is set to all inputs.

```
SMALL = 0
```

```
LARGE = 1
```

```
class RelayControlMode(*values)
```

Relay operating mode enumeration.

```
RELAY_OFF = 0
```

```
RELAY_ON = 1
```

```
ALARMS = 2
```

```
class lakeshore.model_336.Model336Enums
```

Class containing the enums relevant to the Model 336.

```
class InputChannel(*values)
```

Enumeration where “NONE” is an option for sensor input.

```
NONE = 0
```

```
CHANNEL_A = 1
```

```
CHANNEL_B = 2
```

```
CHANNEL_C = 3
```

```
CHANNEL_D = 4
```

```
CHANNEL_D2 = 5
```

```
CHANNEL_D3 = 6
```

```
CHANNEL_D4 = 7
```

```
CHANNEL_D5 = 8
```

```
class DisplaySetupMode(*values)
```

Front panel display setup enum.

```
INPUT_A = 0
```

```
INPUT_B = 1
INPUT_C = 2
INPUT_D = 3
CUSTOM = 4
FOUR_LOOP = 5
ALL_INPUTS = 6
INPUT_D2 = 7
INPUT_D3 = 8
INPUT_D4 = 9
INPUT_D5 = 10
```

```
class InputSensorType(*values)
```

Sensor type enumeration.

THERMOCOUPLE is only valid with the 3060 option, CAPACITANCE is only valid with the 3061 option.

```
DISABLED = 0
DIODE = 1
PLATINUM_RTD = 2
NTC_RTD = 3
THERMOCOUPLE = 4
CAPACITANCE = 5
```

```
class DiodeRange(*values)
```

Diode voltage range enumeration

```
TWO_POINT_FIVE_VOLTS = 0
TEN_VOLTS = 1
```

```
class RTDRange(*values)
```

RTD resistance range enumeration.

THIRTY_THOUSAND_OHM and ONE_HUNDRED_THOUSAND_OHM are only valid for NTC RTDs.

```
TEN_OHM = 0
THIRTY_OHM = 1
HUNDRED_OHM = 2
THREE_HUNDRED_OHM = 3
ONE_THOUSAND_OHM = 4
```

THREE_THOUSAND_OHM = 5

TEN_THOUSAND_OHM = 6

THIRTY_THOUSAND_OHM = 7

ONE_HUNDRED_THOUSAND_OHM = 8

class ThermocoupleRange(*values)

Thermocouple range enumeration.

FIFTY_MILLIVOLT = 0

class HeaterOutputMode(*values)

Control loop enumeration.

OFF = 0

CLOSED_LOOP = 1

ZONE = 2

OPEN_LOOP = 3

MONITOR_OUT = 4

WARMUP_SUPPLY = 5

class HeaterRange(*values)

Current mode heater enumerations.

OFF = 0

LOW = 1

MEDIUM = 2

HIGH = 3

class HeaterVoltageRange(*values)

Voltage mode heater enumerations.

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class DisplayFieldUnits(*values)

Panel display units enumeration.

KELVIN = 1

CELSIUS = 2

SENSOR_UNITS = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

SENSOR_NAME = 6

Model 346 Cryogenic Temperature Controller

The Model 346 measures and controls cryogenic temperature environments.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example Scripts

Below are a few example scripts for the Model 346 that use the Lake Shore Python driver.

Using calibration curves with a temperature instrument

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and
↳ serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a
↳ temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↳ VOLTS_PER_KELVIN, 300.0,
                                     myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set
↳ to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.
↳ The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,
↳ 30, "SN123", (276, 10),
                                     (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can
↳ then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that
↳ curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Setting up outputs on the Model 346

```

from lakeshore import Model346

# Connect to the first available Model 346 temperature controller over USB
my_model_346 = Model346()

# Set the control loop values for outputs 1 and 5
my_model_346.set_heater_pid(1, 40, 27, 0)
my_model_346.set_heater_pid(5, 42, 25, 10)

# Configure heater output 1 with a 50 ohm load, 0.75 amp max current, and screen display.
↳mode to power
my_model_346.set_heater_setup(1, 50.0, 0.75, my_model_346.HeaterOutputUnits.POWER)

# Configure settings of analog output 5 to have a high and low value of 3.65 and 1.02.
↳(preferred units) respectively
my_model_346.set_analog_output_settings(5, 3.65, 1.02)

# Set closed loop output mode for outputs 1 and 5
my_model_346.set_output_mode(1, my_model_346.OutputMode.PID, "A")
my_model_346.set_output_mode(5, my_model_346.OutputMode.PID, "A")

# Set a control setpoint for outputs 1 and 5 to 1.5 and 2.5 (preferred units).
↳respectively
my_model_346.set_control_setpoint(1, 1.5)
my_model_346.set_control_setpoint(5, 2.5)

# Turn the heater on by setting the heater range
my_model_346.set_heater_range(1, my_model_346.HeaterRange.LOW)
# Output 5 "turned on" when its output mode was set.

# Obtain the output percentage of output 1 and print it to the console
output_one = my_model_346.get_heater_output(1)
print("Output 1: " + str(output_one))

# Obtain the output percentage of output 3 and print it to the console
output_five = my_model_346.get_analog_output_settings(5)
print("Output 5: " + str(output_five))

```

Setting up options cards on the Model 346

```

from lakeshore import Model346, Model346InputSensorSettings

# Initialize the Model346 temperature controller
my_model_346 = Model346()

# Retrieve connected option cards as a dictionary of slot identifiers and OptionCard.
↳objects
option_cards = my_model_346.get_installed_option_cards()

# Print the details of each option card

```

(continues on next page)

(continued from previous page)

```

for slot, card in option_cards.items():
    print(f"Slot {slot}: {card}")

# Define NTC RTD sensor settings
ntc_rtd_settings = Model346InputSensorSettings(
    sensor_type=my_model_346.InputSensorType.NTC_RTD,
    autorange_enable=True,
    compensation=True,
    units=my_model_346.InputSensorUnits.KELVIN,
    input_range=my_model_346.NTCRTDRange.ONE_THOUSAND_OHM
)

# Define Thermocouple settings for type E
thermocouple_settings = Model346InputSensorSettings(
    sensor_type=my_model_346.InputSensorType.THERMOCOUPLE,
    autorange_enable=False, # Thermocouples have a fixed range of 50mV
    compensation=True,
    units=my_model_346.InputSensorUnits.KELVIN,
    input_range=my_model_346.ThermocoupleRange.FIFTY_MILLIVOLTS
)

# Loop through option card slots and configure channels for SCANNER and THERMOCOUPLE_
↳cards
for slot, option_card in option_cards.items():

    # Configure all scanner input sensor channels as NTC RTDs
    if option_card.card_type == my_model_346.OptionCardType.SCANNER:
        for channel in option_card.channels: # Loop through channels of the SCANNER card
            my_model_346.set_input_sensor(input_channel=channel, sensor_params=ntc_rtd_
↳settings)

        # Get the scanner input sensor settings and print them in a readable format
        for channel in option_card.channels:
            sensor_settings = my_model_346.get_input_sensor(input_channel=channel)
            print(
                f"Channel {channel} on slot {slot} has sensor settings: "
                f"SensorType={sensor_settings.sensor_type.name}, "
                f"AutoRange={sensor_settings.autorange_enable}, "
                f"Compensation={sensor_settings.compensation}, "
                f"Units={sensor_settings.units.name}, "
                f"InputRange={sensor_settings.input_range.name}"
            )

        # Read and print the current temperature reading from each scanner channel for_
↳channel in option_card.channels:
            temperature_reading = my_model_346.get_kelvin_reading(input_channel=channel)
            print(f"Current temperature reading from channel {channel} on slot {slot}:
↳{temperature_reading} K")

    # Thermocouple setup for channels on a Thermocouple card
    if option_card.card_type == my_model_346.OptionCardType.THERMOCOUPLE:
        for channel in option_card.channels:

```

(continues on next page)

(continued from previous page)

```

    # Set the sensor parameters for the thermocouple
    my_model_346.set_input_sensor(input_channel=channel, sensor_
↳params=thermocouple_settings)

    my_model_346.set_input_curve(input_channel=channel, curve_number=my_model_
↳346.StandardCurveType.THERMOCOUPLE_TYPE_E)

    # Get the curve settings and print them
    curve_settings = my_model_346.get_input_curve(input_channel=channel)
    print(f"Channel {channel} on slot {slot} has curve settings: {curve_settings}
↳")

    # Get the thermocouple sensor settings and print them
    sensor_settings = my_model_346.get_input_sensor(input_channel=channel)
    print(
        f"Channel {channel} on slot {slot} has sensor settings: "
        f"SensorType={sensor_settings.sensor_type.name}, "
        f"AutoRange={sensor_settings.autorange_enable}, "
        f"Compensation={sensor_settings.compensation}, "
        f"Units={sensor_settings.units.name}, "
        f"InputRange={sensor_settings.input_range.name}"
    )

    # Read and print the current temperature reading from the thermocouple
↳channel
    temperature_reading = my_model_346.get_kelvin_reading(input_channel=channel)
    print(f"Current temperature reading from channel {channel} on slot {slot}:
↳{temperature_reading} K")

```

Instrument class methods

```
class lakeshore.model_346.Model346(serial_number=None, com_port=None, baud_rate=921600,
    flow_control=False, timeout=5.0, ip_address=None, tcp_port=7777,
    **kwargs)
```

Class for interaction with the Lake Shore Model 346 cryogenic temperature controller.

set_filter(input_channel, filter_settings)

Configures the input filter for the specified channel.

The reading filter applies exponential smoothing to the sensor input readings. If the filter is turned on for a sensor input, all reading values for that input are filtered. The filter is a running average, so it does not change the update rate of an input. The filtered reading is used for everything except closed loop control loop calculations, which use the unfiltered reading.

Args:

input_channel (str):

Specifies input to configure: A, B, C1, C2, C3, C4, D1, D2, D3, D4, E1, E2, E3, E4, F1, F2, F3, F4, G1, G2, G3, G4, H1, H2, H3, H4

filter_settings (Model346FilterSettings):

The filter configuration to apply to the input.

get_filter(*input_channel*)

Returns the input filter configuration for the specified channel.

Args:

input_channel (str):

Specifies input to query: A, B, C1, C2, C3, C4, D1, D2, D3, D4, E1, E2, E3, E4, F1, F2, F3, F4, G1, G2, G3, G4, H1, H2, H3, H4

Returns:

Model346FilterSettings:

An Model346FilterSettings object populated with the filter parameters.

get_option_card_slot()

Returns the available option card slots.

get_digital_input_status()

Returns the status of the digital inputs.

Returns:

dict[str, int]:

{“input_one”: int, “input_two”: int}

set_threshold(*input_channel, threshold_id, threshold_settings*)

Configures a reading threshold for the specified input.

The threshold feature allows monitoring of up to four independent thresholds per input.

Args:

input_channel (str):

Specifies input to configure: A, B, C1-C4, D1-D4, E1-E4, F1-F4, G1-G4, H1-H4.

threshold_id (int):

Threshold identifier (1 to 4).

threshold_settings (Model346ThresholdSettings):

Threshold configuration to apply.

get_threshold(*input_channel, threshold_id*)

Returns the reading threshold configuration for the specified input and threshold ID.

Args:

input_channel (str):

Specifies input to query: A, B, C1-C4, D1-D4, E1-E4, F1-F4, G1-G4, H1-H4.

threshold_id (int):

Threshold identifier (1 to 4).

Returns:

Model346ThresholdSettings:

An object populated with the threshold value and comparison operator.

set_alarm_settings(*input_channel, alarm_settings*)

Sets the configuration of a particular alarm.

Args:

input_channel (str):

Specifies the input channel to configure (e.g., A, B, C1, C2, etc.).

alarm_settings (Model346AlarmSettings):

The configuration that should be applied to the alarm.

get_alarm_settings(*input_channel*)

Returns present configuration of a particular alarm.

Args:**input_channel (str):**

Specifies the input channel to query (e.g., A, B, C1, C2, etc.).

Returns:**Model346AlarmSettings:**

An Model346AlarmSettings object populated with the respective alarm parameters.

set_stability_settings(*output_channel*, *stability_settings*)

Sets the stability detection configuration for a particular output.

Args:**output_channel (int):**

Specifies the output channel to configure (1 to 10).

stability_settings (Model346StabilitySettings):

The configuration that should be applied to the output.

get_stability_settings(*output_channel*)

Returns present configuration of a particular stability setting.

Args:**output_channel (int):**

Specifies the output of interest.

Returns:**Model346StabilitySettings:**

An Model346StabilitySettings object populated with the respective stability parameters.

turn_relay_on(*relay_number*)

Turns the specified relay on.

Args:**relay_number (int):**

The relay to turn on. Options are: 1 or 2.

turn_relay_off(*relay_number*)

Turns the specified relay off.

Args:**relay_number (int):**

The relay to turn off. Options are: 1 or 2.

set_input_sensor(*input_channel*, *sensor_params*)

Sets the sensor type and associated parameters.

Args:**input_channel (str):**

Specifies input to configure (A - H4).

sensor_params (Model346InputSensorSettings):

See Model346InputSensorSettings class.

get_input_sensor(*input_channel*)

Returns the sensor type and associated parameters.

Args:

input_channel (str):

Specifies input to query (A - H4).

Returns:

Model346InputSensorSettings:

The configured sensor settings for a particular input sensor.

set_heater_range(*output, heater_range*)

Sets the heater range for a particular output.

The range setting has no effect if an output is in the Off mode, and does not apply to an output in Monitor Out mode. An output in Monitor Out mode is always on.

Args:

output (int):

Specifies which output to configure (1 to 4, 9, or 10).

heater_range (Model346Enums.HeaterRange):

The heater range to apply to a particular output.

get_heater_range(*output*)

Returns the heater range for a particular output.

Args:

output (int):

Specifies which output to query (1 to 4, 9, or 10).

Returns:

(Model346Enums.HeaterRange):

The heater range being applied to a particular output.

set_heater_autorange(*output, enabled*)

Sets the heater autorange setting for a particular output.

Args:

output (int):

Specifies which output to configure (1 - 4).

enabled (int):

Whether autorange should be enabled or disabled.

get_heater_autorange(*output*)

Returns the heater autorange setting for a particular output.

Args:

output (int):

Specifies which output to query (1 - 4).

Returns:

(Model346Enums.HeaterAutorange):

Whether autorange is enabled or disabled.

set_analog_out_state(*output*, *analog_out_state*)

Sets the analog out state to enable or disable

Args:**output (int):**

Specifies which output to configure (5 - 8).

analog_out_state (int):

OFF = 0, ON = 1.

get_analog_out_state(*output*)

Returns the analog state for a particular output.

Args:**output (int):**

Specifies which output to query (5 - 8).

Returns:**(int):**

The analog output state being applied to a particular output. OFF = 0, ON = 1.

set_output_percent_limit(*output*, *percent_limit*)

Sets the output percent limit for a particular output.

Args:**output (int):**

Specifies which output to configure (1 - 10).

percent_limit (float):

Sets a percentage limit for a particular output (0 to 100).

get_output_percent_limit(*output*)

Returns the output percent limit for a particular output.

Args:**output (int):**

Specifies which output to query (1 - 10).

Returns:**float:**

The set percentage limit for a particular output.

all_heaters_off()

Recreates the front panel safety feature of shutting off all heaters.

set_heater_setup(*output*, *heater_resistance*, *max_output*, *output_units*)

Method to configure the heaters.

Args:**output (int):**

Specifies which heater output to configure (1 to 4, 9 or 10)

heater_resistance (float):

10 to 100 ohms.

max_output (float):

User defined maximum output. Represents 100% output as either power (watts) or current (amps) depending on the output_units parameter.

output_units (Model346Enums.HeaterOutputUnits):

Specifies whether the heater output displays in power or current.

get_heater_setup(*output*)

Returns the heater configuration status.

Args:

output (int):

Specifies which heater output to query (1 to 4, 9 or 10).

Returns:

dict[str, float | Model346Enums.HeaterOutputUnits]:

{“heater_resistance”: float, “max_output”: float, “output_units”: HeaterOutputUnits}

get_heater_output_measurements(*output*)

Returns the heater output measurements.

Args:

output (int):

Specifies which heater output to query (1 - 8).

Returns:

dict[str, float]:

{“heater_voltage”: float, “heater_current”: float, “heater_resistance”: float, “heater_power”: float}

get_calculated_heater_output(*output*)

Returns the calculated output value and power for the specified output.

Args:

output (int):

Specifies which output to query (1 - 10).

Returns:

dict[str, float]:

{“output_value”: float, “output_power”: float}

For outputs 1-4: output_value is current in Amperes For outputs 5-8: output_value is voltage in Volts, output_power is not calculated and will return as 0 For outputs 9-10: output_value is sum of currents, output_power is sum of powers

set_output_mode(*output, mode, input_channel, powerup_enable=False, warmup=False*)

Configures the output mode of a particular output.

Args:

output (int):

Specifies which output to configure (1 - 10).

mode (Model346Enums.OutputMode):

Specifies the control mode.

input_channel (str):

Specifies which input to use for control (A - H4, NONE). Unused if not in PID mode.

powerup_enable (bool):

Specifies whether the output remains on (True) or shuts off after power cycle (False).

warmup (bool):

Specifies whether the output is enabled and set when a WARM is issued (True) or not (False).

get_output_mode(*output*)

Returns the output mode for a particular output.

Args:**output (int):**

Specifies which output to retrieve (1 - 10).

Returns:**Model346OutputModeSettings:**

A Model346OutputModeSettings object populated with the output mode parameters.

set_output_group(*output, group*)

Assigns a heater output to a virtual heater group for coordinated control.

Args:**output (int):**

Specifies which heater output to configure (1 - 4).

group (int):

The virtual heater group to assign the output to: 9 or 10. Set to 0 to remove the output from a group.

get_output_group(*output*)

Returns the virtual heater group assignment for a particular output.

Args:**output (int):**

Specifies which heater output to query (1 - 4).

Returns:**int:**

The virtual heater group (9 or 10), or 0 if not assigned to a group.

start_warmup(*room_temperature=293.15, percent_out=10*)

Starts the warmup process for all heaters configured with warmup set to True in OUTMODE.

Args:**room_temperature (float):**

Target temperature in Kelvin (275 to 325). Defaults to 293.15 K.

percent_out (float):

The acceptable output percentage for the system to be considered WARM (0 to 100). Defaults to 10.

set_analog_output_settings(*output, high_value, low_value*)

Configures a voltage-controlled output.

Use the set_output_mode command to set the output mode to Monitor Out.

Args:

output (int):

Analog output channel to configure (5 - 8).

high_value (float):

Represents the input reading at which the Monitor Out reaches +100% output (+10 V).

low_value (float):

Represents the input reading at which the analog output reaches 0% output (0 V).

get_analog_output_settings(*output*)

Used to obtain the monitor out parameters for a specific output.

Args:

output (int):

Analog output of interest (5 - 8).

Returns:

dict[str, float]:

{“high_value”: float, “low_value”: float}

reset_alarm_status()

Resets all alarm statuses including latched alarms.

set_keypad_lock(*state*)

Sets the state of the front panel settings lock.

Args:

state (bool): The desired state of the settings lock (True for locked, False for unlocked).

get_keypad_lock()

Returns the current state of the front panel settings lock.

Returns:

bool: True if settings are locked, False if unlocked.

reset_min_max_data(*channel*)

Resets the minimum and maximum reading data for the give input, or all inputs if ALL is specified.

Args:

channel (str):

Specifies the input channel to reset (A - H4) or ALL to reset all inputs.

get_input_reading_status(*input_channel*)

Returns the status of the specified input channel.

Args:

input_channel (str):

Specifies the input channel to query (A - H4).

Returns:

(Model346InputReadingStatus):

Boolean representation of each bit of the input status flag register.

get_input_reading_operation_status(*input_channel*)

Returns the operation status of the specified input channel.

Args:

input_channel (str):

Specifies the input channel to query (A - H4).

Returns:**(Model346InputReadingOperationStatus):**

Boolean representation of each bit of the input operation status flag register. Includes: `curve_applied`, `reading_in_celsius`, `low_alarm_active`, `high_alarm_active`.

get_output_operation_status(output)

Returns the operation status of the specified output.

Args:**output (int):**

Specifies the output to query (1 - 10).

Returns:**(Model346OutputOperationStatus):**

Boolean representation of each bit of the output operation status flag register. Includes: `ramp_active`, `stabilizing`, `stable`, `warmup_mode`, `warm`.

get_output_status(output)

Returns the status of the specified output.

Args:**output (int):**

Specifies the output to query (1 - 10).

Returns:**(Model346OutputStatus):**

Boolean representation of each bit of the output status flag register. Includes: `- heater_error`: HTRST? is non-zero - `percentage_limited`: Output is percentage limited - `current_source_zeroed`: Output current source is zeroed to prevent transience while changing ranges

get_thermocouple_junction_temp(input_channel)

Returns the thermocouple junction temperature for the specified input channel.

Args:**input_channel (str):**

Specifies the thermocouple input to query (E1, E2, F1, F2, G1, G2, H1, H2).

Returns:**float:**

The thermocouple junction temperature in Kelvin.

set_heater_limit(output, enabled, short_circuit_resistance, open_circuit_resistance)

Enables or disables the heater limit for a particular output and sets the bounds.

Args:**output (int):**

Specifies which output to configure (1 - 4).

enabled (int):

Specifies whether the heater limit is enabled.

short_circuit_resistance (float):

Specifies the short circuit limit in ohms.

open_circuit_resistance (float):

Specifies the open circuit limit in ohms.

get_heater_limit(*output*)

Returns the heater limit settings for a particular output.

Args:

output (int):

Specifies which output to query (1 - 4).

Returns:

dict[str, float | bool]:

{“enabled”: bool, “short_circuit_resistance”: float, “open_circuit_resistance”: float}

reset_control_setpoint(*output*)

Resets the control setpoint for a particular output.

Args:

output (int):

Specifies which output to reset (1 - 10).

get_ramp_setpoint(*output*)

Returns the current setpoint for a particular output.

Args:

output (int):

Specifies which output to query (1 - 10).

Returns:

float:

The current control setpoint for the specified output.

set_control_loop_zone_table(*output, zone_number, zone_settings*)

Sets the control loop zone table for a particular output.

Args:

output (int):

Specifies which output to configure (1 to 4, 9, or 10).

zone_number (int):

Specifies which zone to configure (1 - 10).

zone_settings (Model346ControlLoopZoneSettings):

The settings to apply to the specified zone.

get_control_loop_zone_table(*output, zone_number*)

Returns the control loop zone table for a particular output and zone.

Args:

output (int):

Specifies which output to query (1 to 4, 9, or 10).

zone_number (int):

Specifies which zone to query (1 - 10).

Returns:

Model346ControlLoopZoneSettings:

The settings for the specified control loop zone.

get_curve_number_of_points(*curve_number*)

Returns the number of data points in a curve.

Args:**curve_number (int):**

Specifies which curve to query (1 - 60).

Returns:**int:**

The number of points in the specified curve. Returns 0 if the curve is deleted/empty

set_relay_alarms(*relay_number, mode, input_channel, condition*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:**relay_number (int):**

The relay to configure (1 or 2).

mode (Model346Enums.RelayControlMode):

Specifies the relay control feature/mode.

input_channel (str | int):

Input instance for the specified mode. Type depends on mode: - For THERMOMETRY_INPUT: input channel (A, B, C1-C4, D1-D4, E1-E4, F1-F4, G1-G4, H1-H4, NONE) - For OUTPUT_STATUS: numeric output (1-10) - For FUNCTION_OUTPUT: function ID (1-16) - For DIGITAL_INPUT: digital input number (1-2) - For SYSTEM_STATUS: 0 (warmup all) - For OFF and ON: ignored (use 0)

condition (int | IntEnum):

Condition to evaluate. Type depends on mode: - For THERMOMETRY_INPUT: RelayThermometryCondition enum or int (0-10) - For OUTPUT_STATUS: RelayOutputStatusCondition enum or int (0-4) - For DIGITAL_INPUT: RelayDigitalInputCondition enum or int (0-1) - For FUNCTION_OUTPUT: RelayFunctionOutputCondition enum or int (0-1) - For OFF, ON, and SYSTEM_STATUS: ignored (use 0)

get_relay_alarm_control_parameters(*relay_number*)

Returns the relay control parameters for a specified relay.

Args:**relay_number (int):**

The relay to query (1 or 2).

Returns:**Model346RelayControlSettings:**

A Model346RelayControlSettings object populated with relay control parameters.

get_thermocouple_compensation_offset(*input_channel*)

Returns the thermal block compensation offset for a thermocouple input.

Args:**input_channel (str):**

Specifies the thermocouple input to query. Valid channels: E1, E2, F1, F2, G1, G2, H1, H2

Returns:

float:

Offset value in Kelvin currently applied to the thermal block reading. Returns 0.0 if no offset has been configured.

get_installed_option_cards()

Queries the installed option cards and returns detailed information.

Returns:

dict[str, OptionCard]: A dictionary mapping slots (E, F, G, H) to OptionCard objects.

get_option_card_type(slot)

Returns the type of option card connected to the specified slot.

Args:

slot (str): Specifies the slot to query (E, F, G, H).

Returns:

Model346Enums.OptionCardType: The type of option card connected to the slot.

Raises:

ValueError: If the slot is invalid.

get_option_card_channels(slot)

Returns the list of channels available on the specified option card slot.

Args:

slot (str): Specifies the slot to query (E, F, G, H).

Returns:

list[str]: The list of channels available on the option card in the specified slot.

Raises:

ValueError: If the option card type is unsupported.

class AutotuneMode(*values)

Enumerator used to represent the different autotune control modes.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class BrightnessLevel(*values)

Enumerator to specify the brightness level of an instrument display.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class ControlTypes(*values)

Enumerator used to represent the control type settings.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
```

(continues on next page)

(continued from previous page)

```
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class CurveFormat(*values)

Enumerations specify formats for temperature sensor curves.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class CurveTemperatureCoefficient(*values)

Enumerations specify positive/negative temperature sensor curve coefficients.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class DiodeCurrent(*values)

Enumerator used to represent diode current ranges.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class DisplayFields(*values)

Enumeration of the possible number of fields to include in a custom display mode.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class DisplayFieldsSize(*values)

Enumeration of the display fields when mode is set to all inputs.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class HeaterError(*values)

Enumeration for possible errors flagged by the heater.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class Interface(*values)

Enumerator used to represent remote interface communication methods.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class InterfaceMode(*values)

Enumeration for the mode of the remote interface.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class LanStatus(*values)

Represents the different status states for the lan connection.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

clear_interface_command()

Clears the bits in the SBR, SESR, OER, and terminates all operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register. Terminates all pending operations. Clears the interface, but not the controller.

command(*commands, check_errors=True)

Send an SCPI command or multiple commands to the instrument.

Args:**commands (str):**

Any number of SCPI commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

delete_curve(curve)

Deletes the user curve.

Args:**curve (int):**

Specifies a user curve to delete.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

Resets all system information such as settings, wi-fi connections, date and time, etc.

get_control_setpoint(*output*)

Returns the value for a given control output.

Args:

output (int):

Specifies which output's control loop to query (1 or 2).

Returns:

(float):

The value for the set-point (in the preferred units of the control loop sensor).

get_curve(*curve*)

Returns a list of all the data points in a particular curve.

Args:

curve (int):

Specifies which curve to set.

Returns:

list[tuple[float, float] | tuple[float, float, float]]:

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_data_point(*curve, index*)

Returns a standard or user curve data point.

Args:

curve (int):

Specifies which curve to query.

index (int):

Specifies the points index in the curve.

Returns:

tuple[float, float] | tuple[float, float, float]:

(sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_header(*curve_number*)

Returns parameters set on a particular user curve header.

Args:

curve_number (int):

Specifies a curve to retrieve.

Returns:

(CurveHeader):

A CurveHeader class object containing the curve information.

get_heater_output(*output*)

Sample heater output in percent, scale is dependent upon the instrument used and heater configuration.

Args:**output (int):**

Heater output to query.

Returns:**(float):**

percent of full scale current/voltage/power.

get_heater_pid(*output*)

Returns the closed loop control parameters of the heater output.

Args:**output (int):**

Specifies which output's control loop to query.

Returns:**(dict):**

{"gain": float, "integral": float, "derivative": float}

gain: Proportional term in PID control. integral: Integral term in PID control. derivative: Derivative term in PID control.

get_heater_status(*output*)

Returns the heater error code state, error is cleared upon querying the heater status.

Args:**output (int):**

Specifies which heater output to query (1 or 2).

Returns:**(TemperatureControllerEnums.HeaterError):**

Object of instrument's HeaterError type.

get_input_curve(*input_channel*)

Returns the curve number being used for a given input.

Args:**input_channel (str or int):**

Specifies which input to query.

Returns:**(int):**

Curve number (0-59).

get_kelvin_reading(*input_channel*)

Returns the temperature value in kelvin of the given channel.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:

float:

The temperature value.

get_manual_output(*output*)

Returns the manual output value in percent.

Args:

output (int):

Specifies output to query.

Returns:

(float):

Manual output percent.

get_min_max_data(*input_channel*)

Returns the minimum and maximum data from an input.

Args:

input_channel (str):

Specifies which input to query.

Returns:

(dict[str, float]):

{“minimum”: float, “maximum”: float}

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_relay_control_mode(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):

Specifies which relay to query. Options are: 1 or 2.

Returns:

(LegacyTemperatureControllerEnums.RelayControlMode):

The configured mode of the relay. Represented as a member of the instrument's RelayControlMode IntEnum class.

get_relay_status(*relay_channel*)

Returns whether the relay at the specified channel is On or Off.

Args:**relay_channel (int):**

The relay channel to query.

Returns:**(bool):**

True if relay is on, False if relay is off.

get_self_test()

Instrument self test result completed at power up.

Returns:**(bool):**

True = errors found. False = no errors found.

get_sensor_name(*input_channel*)

Returns the name of the sensor on the specified channel.

Args:**input_channel (str or int):**

Specifies which input_channel channel to read from.

Returns:**name (str):**

Name associated with the sensor.

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor's units.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:**(float):**

The raw sensor reading in the units of the connected sensor.

get_service_request()

Returns the status byte register bits and their values as a class instance.

get_service_request_enable_mask()

Returns the named bits of the status byte service request enable register.

This register determines which bits propagate to the master summary status bit.

get_setpoint_ramp_parameter(*output*)

Returns the control loop parameters of a particular output.

Args:

output (int):

Specifies which output's control loop to return.

Returns:

(dict[str, bool | float]):

{“ramp_enable”: bool, “rate_value”: float}

get_setpoint_ramp_status(*output*)

Returns whether the set-point is ramping.

Args:

output (int):

Specifies which output's control loop to query.

Returns:

(bool):

Ramp status. False = Not ramping, True = Ramping.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

get_standard_events()

Returns the names of the standard event register bits and their values.

get_status_byte()

Returns named bits of the status byte register and their values.

get_temperature_limit(*input_channel*)

Returns the value of the temperature limit in kelvin.

Args:

input_channel (str or int):

Specifies which input to query.

Returns:

float:

The value of the temperature limit.

modify_operation_register_mask(*bit_name, value*)

Gets the operation condition register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_questionable_register_mask(*bit_name, value*)

Gets the questionable condition register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_service_request_mask(*bit_name*, *value*)

Gets the service request enable mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_standard_event_register_mask(*bit_name*, *value*)

Gets the standard event register mask, changes a bit, and sets the register.

Args:**bit_name (str):**

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

query(**queries*, *check_errors=True*)

Sends an SCPI query or multiple queries to the instrument and return the response(s).

Args:**queries (str):**

Any number of SCPI queries or commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

The instrument query response as a string.

reset_instrument()

Sets controller parameters to power-up settings.

reset_measurement_settings()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_control_setpoint(*output*, *value*)

Set set-point for specific output's control loop.

Control settings, that is, P, I, D, and Set-point, are assigned to outputs, which results in the settings being applied to the control loop formed by the output and its control input.

Args:**output (int):**

Specifies which output's control loop to configure.

value (float):

The value for the set-point (in the preferred units of the control loop sensor).

set_curve(*curve, data_points*)

Method to define a user curve using a list of data points.

Args:

curve (int):

Specifies which curve to set.

data_points (list[tuple[float, float] | tuple[float, float, float]]):

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

set_curve_data_point(*curve, index, sensor_units, temperature, curvature=None*)

Configures a user curve point.

Args:

curve (int or str):

Specifies which curve to configure.

index (int):

Specifies the points index in the curve.

sensor_units (float):

Specifies sensor units for this point to 6 digits.

temperature (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

curvature (float):

Specify only if the point is part of a cubic spindle curve. The curvature value scale used to calculate spindle coefficients to 6 digits. Optional parameter.

set_curve_header(*curve_number, curve_header*)

Configures the user curve header.

Args:

curve_number (int):

Specifies which curve to configure.

curve_header (CurveHeader):

Instrument's CurveHeader class object containing the desired curve information.

set_heater_pid(*output, gain, integral, derivative*)

Configure the closed loop control parameters of the heater output.

Args:

output (int):

Specifies which output's control loop to configure.

gain (float):

Proportional term in PID control. This controls how strongly the control output reacts to the present error.

integral (float):

Integral term in PID control. This controls how strongly the control output reacts to the past error history.

derivative (float):

Derivative term in PID control. This value controls how quickly the present field set point will transition to a new set-point.

set_input_curve(*input_channel*, *curve_number*)

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str or int):

Specifies which input to configure.

curve_number (int):

0 = none, 1-20 = standard curves, 21-59 = user curves.

set_manual_output(*output*, *value*)

When instrument is in closed loop PID, Zone, or Open Loop modes a manual output may be set.

Args:

output (int):

Specifies output to configure.

value (float):

Specifies value for manual output in percent.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:

register_mask ([Instrument]OperationRegister):

An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:

register_mask ([Instrument]QuestionableRegister):

An instrument specific QuestionableRegister class object with all bits configured true or false.

set_sensor_name(*input_channel*, *sensor_name*)

Sets a given name to a sensor on the specified channel.

Args:

input_channel (str or int):

Specifies which input_channel channel to read from.

sensor_name (str):

Name user wants to give to the sensor on the specified channel.

set_service_request(*register_mask*)

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:

register_mask (service_request_enable):

A service_request_enable class object with all bits configured.

set_service_request_enable_mask(*register_mask*)

Configures values of the service request enable register bits.

This register determines which bits propagate to the master summary bit.

Args:

register_mask (StatusByteRegister):

A StatusByteRegister class object with all bits configured true or false.

set_setpoint_ramp_parameter(*output, ramp_enable, rate_value*)

Sets the control loop of a particular output.

Args:

output (int):

Specifies which output's control loop to configure.

ramp_enable (bool):

Specifies whether ramping is off or on (False = Off or True = On).

rate_value (float):

Specifies set-point ramp rate in kelvin per minute. The rate is always positive but will respond to ramps up or down. A rate of 0 is interpreted as infinite, and will respond as if set-point ramping were off. (0.1 to 100)

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits configured true or false.

set_temperature_limit(*input_channel, limit*)

After a set temperature limit is exceeded, all control outputs will shut down.

Args:

input_channel (str or int):

Specifies which input to configure.

limit (float):

The temperature limit in kelvin for which to shut down all control outputs when exceeded. A limit of zero will turn the feature off.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Settings classes

This section outlines the classes used to interact with methods which return or accept an argument of a class object, specific to the Lake Shore model 346.

```
class lakeshore.model_346.Model346InputSensorSettings(sensor_type, autorange_enable,  
                                                    compensation, units, input_range=None)
```

Class object used in the get/set_input_sensor methods.

```
__init__(sensor_type, autorange_enable, compensation, units, input_range=None)
```

Constructor for the InputSensor class.

Args:

sensor_type (Model346Enums.InputSensorType):

Specifies input sensor type.

autorange_enable (bool):

Specifies autoranging (False = off, True = on).

compensation (bool):

Specifies input compensation. (False = off, True = on).

units (Model346Enums.InputSensorUnits):

Specifies the preferred units parameter for sensor readings and for the control set-point.

input_range (IntEnum):

Specifies input range if autorange_enable is False. See IntEnum classes: Model346Enums.DiodeRange, Model346Enums.PTCRTDRange, and Model346Enums.NTCRTDRange.

```
class lakeshore.model_346.Model346AlarmSettings(enabled, high_limit, low_limit, deadband, latch,  
                                                    audible, visible)
```

Class used to configure an alarm in conjunction with the set/get_alarm_settings() method for the Model346.

```
__init__(enabled, high_limit, low_limit, deadband, latch, audible, visible)
```

Constructor for AlarmSettings class.

Args:

enabled (bool):

Enables or Disables the alarm.

high_limit (float):

Specifies the high limit for the alarm. Units depend on whether a curve is assigned to the input (sensor units if no curve, K or C per INTYPE if a curve is assigned).

low_limit (float):

Specifies the low limit for the alarm.

deadband (float):

Specifies the deadband for the alarm.

latch (bool):

Specifies whether the alarm latches when triggered.

audible (bool):

Specifies whether the alarm is audible when triggered.

visible (bool):

Specifies whether the alarm is visible when triggered.

```
lakeshore.model_346.Model346CurveHeader
```

alias of *CurveHeader*

Enumeration objects

This section describes the Enum type objects that have been created to name various settings of the Model 346 series that are represented as an int or single character to the instrument. The purpose of these enum types is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

`class lakeshore.model_346.Model346Enums`

Class containing the enums relevant to the Model 346.

`class InputSensorType(*values)`

Sensor type enumeration.

`DISABLED = 0`

`DIODE = 1`

`PLATINUM_RTD = 2`

`NTC_RTD = 3`

`THERMOCOUPLE = 4`

`class InputSensorUnits(*values)`

Enumerator used to represent temperature sensor unit options.

`KELVIN = 0`

`CELSIUS = 1`

`class DiodeRange(*values)`

Diode voltage range enumeration

`TWO_POINT_FIVE_VOLTS = 0`

`class PTCRTDRange(*values)`

PTC RTD resistance range enumeration.

`TEN_OHM = 0`

`ONE_HUNDRED_OHM = 1`

`ONE_THOUSAND_OHM = 2`

`class NTCRTDRange(*values)`

NTC RTD resistance range enumeration.

`ONE_HUNDRED_OHM = 0`

`THREE_HUNDRED_OHM = 1`

`ONE_THOUSAND_OHM = 2`

`THREE_THOUSAND_OHM = 3`

`TEN_THOUSAND_OHM = 4`

`THIRTY_THOUSAND_OHM = 5`

`HUNDRED_THOUSAND_OHM = 6`

```
class ThermocoupleRange(*values)
    Thermocouple voltage range enumeration.
    FIFTY_MILLIVOLTS = 0

class HeaterRange(*values)
    Heater range enumerations for heater outputs(1-4) amd heater groups 9 and 10.
    OFF = 0
    LOW = 1
    HIGH = 2

class HeaterAutorange(*values)
    Heater autorange enumeration for heater outputs 1-4.
    DISABLED = 0
    ENABLED = 1

class HeaterRangeAnalogOutput(*values)
    Heater range enumeration for analog outputs 5 and 8.
    OFF = 0
    ON = 1

class HeaterOutputUnits(*values)
    Enumerator used to represent heater output unit settings.
    POWER = 0
    CURRENT = 1

class OutputMode(*values)
    Output control enumeration.
    OFF = 0
    PID = 1
    ZONE = 2
    OPEN_LOOP = 3
    MONITOR_OUT = 4

class StandardCurveType(*values)
    Enumeration for standard curve types.
    DIODE_DT670 = 2
    PTC_PT100 = 6
    NTC_RX102A_AA = 8
    NTC_RX202A_AA = 9
    NTC_RX103A_AA = 10
```

THERMOCOUPLE_TYPE_K = 12

THERMOCOUPLE_TYPE_E = 13

class RelayControlAlarm(*values)

Enumeration of the setting determining which alarm(s) cause a relay to close in alarm mode.

LOW_ALARM = 0

HIGH_ALARM = 1

LOW_OR_HIGH_ALARM = 2

LOW_AND_HIGH_ALARM = 3

class RelayControlMode(*values)

Enumeration for relay control feature/mode.

OFF = 0

ON = 1

ALARMS = 2

THERMOMETRY_INPUT = 2

OUTPUT_STATUS = 3

DIGITAL_INPUT = 4

SYSTEM_STATUS = 5

FUNCTION_OUTPUT = 6

class RelayThermometryCondition(*values)

Enumeration for relay thermometry input conditions.

LOW_ALARM_ACTIVE = 0

HIGH_ALARM_ACTIVE = 1

EITHER_ALARM_ACTIVE = 2

BOTH_ALARMS_ACTIVE = 3

THRESHOLD_1 = 4

THRESHOLD_2 = 5

THRESHOLD_3 = 6

THRESHOLD_4 = 7

SENSOR_FAULT = 8

TEMPERATURE_FAULT = 9

TEMPERATURE_EXTRAPOLATED = 10

class RelayOutputStatusCondition(*values)

Enumeration for relay output status conditions.

```

OUTPUT_LOAD_FAULT = 0

OUTPUT_STABILIZING = 1

OUTPUT_STABLE = 2

OUTPUT_WARM = 3

OUTPUT_SETPOINT_RAMPING_COMPLETE = 4

class RelayDigitalInputCondition(*values)
    Enumeration for relay digital input conditions.

    LOW = 0

    HIGH = 1

class RelayFunctionOutputCondition(*values)
    Enumeration for relay function output conditions.

    FALSE = 0

    TRUE = 1

class ThresholdComparison(*values)
    Enumeration for threshold comparison operator.

    LESS_THAN = 0

    GREATER_THAN = 1

class OptionCardType(*values)
    Enumeration for option card types.

    NONE = 0

    SCANNER = 1

    LEVEL_SENSE = 2

    MOTOR_CONTROL = 3

    THERMOCOUPLE = 4

```

Register Classes

This page describes the register objects. Each bit in the register is represented as a member of the register's class

`lakeshore.model_346.Model346StandardEventRegister`

alias of `StandardEventRegister`

```

class lakeshore.xip_instrument.StandardEventRegister(operation_complete, query_error,
                                                    device_specific_error, execution_error,
                                                    command_error, power_on)

```

Class object representing the standard event register.

```

bit_names = ['operation_complete', 'query_error', 'device_specific_error',
            'execution_error', 'command_error', '', 'power_on']

```

lakeshore.model_346.**Model346StatusByteRegister**

alias of *StatusByteRegister*

```
class lakeshore.xip_instrument.StatusByteRegister(error_available, questionable_summary,
message_available_summary,
event_status_summary, master_summary,
operation_summary)
```

Class object representing the status byte register.

```
bit_names = ['', '', 'error_available', 'questionable_summary',
'message_available_summary', 'event_status_summary', 'master_summary',
'operation_summary']
```

```
class lakeshore.model_346.Model346OperationRegister(core_summary, option_summary,
output_summary, all_warm,
external_24v_missing)
```

Class object representing the operation status register.

Represents the top-level STATUS:OPERation:CONDition? register.

Bit 0 (weight 1): CORE Summary Bit (A, B, C1-4, D1-4) Bit 1 (weight 2): OPTion Summary Bit (E, F, G, H) Bit 2 (weight 4): OUTput Summary Bit (1-10) Bit 4 (weight 16): ALL WARM bit Bit 15 (weight 32768): External 24V not present

All other bits (3, 5-14) are not used per the manual.

```
bit_names = ['core_summary', 'option_summary', 'output_summary', '', 'all_warm', '',
'', '', '', '', '', '', '', '', 'external_24v_missing']
```

```
class lakeshore.model_346.Model346QuestionableRegister(core_summary, option_summary,
output_summary, function_summary)
```

Class object representing the questionable status register.

Represents the top-level STATUS:QUEStionable:CONDition? register.

The Model 346 status system organizes questionable status into:

- CORE: Inputs A, B, C1-C4, D1-D4 (RDGST? status)
- OPTion: Option cards E, F, G, H (channels 1-4 each)
- OUTput: Outputs 1-10 (OUTST? status)
- FUNction: Functions 1-16

```
bit_names = ['core_summary', 'option_summary', 'output_summary', 'function_summary']
```

Model 350 Cryogenic Temperature Controller

The Model 350 measures and controls cryogenic temperature environments.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Instrument methods

```
class lakeshore.model_350.Model350(serial_number=None, com_port=None, baud_rate=57600,
data_bits=7, stop_bits=1, parity='O', flow_control=False,
handshaking=False, timeout=2.0, ip_address=None, tcp_port=7777,
**kwargs)
```

A class object representing the Lake Shore Model 350 cryogenic temperature controller.

command(*command_string*)

Send a command to the instrument.

Args:

command_string (str):

A serial command.

connect_tcp(*ip_address, tcp_port, timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

query(*query_string*)

Send a query to the instrument and return the response.

Args:

query_string (str):

A serial query ending in a question mark.

Returns:

The instrument query response as a string.

write(*command_string*)

Alias of `command`. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Model 372 AC Resistance Bridge

The Model 372 is both an AC resistance bridge and temperature controller designed for measurements below 100 milliKelvin.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example scripts

Setting a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and
```

(continues on next page)

(continued from previous page)

```

↳serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a
↳temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↳VOLTS_PER_KELVIN, 300.0,
                                myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set
↳to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.
↳The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,
↳30, "SN123", (276, 10),
                                (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can
↳then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that
↳curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)

```

Using enums to configure an input sensor

```

from lakeshore import Model372, Model372InputSetupSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure a sensor
# Create Model372InputSetupSettings object with current excitation mode, 31.6 uA
↳excitation current, autoranging on
# (tracking current), current source not shunted, preferred units of Kelvin, and a
↳resistance range of 20.0 kOhms
sensor_settings = Model372InputSetupSettings(my_model_372.SensorExcitationMode.CURRENT,
                                my_model_372.MeasurementInputCurrentRange.
↳RANGE_31_POINT_6_MICRO_AMPS,
                                my_model_372.AutoRangeMode.CURRENT, False,

```

(continues on next page)

(continued from previous page)

```

my_model_372.InputSensorUnits.KELVIN,
my_model_372.MeasurementInputResistance.
↪RANGE_20_KIL_OHMS)

# Pass settings into method along with desired input channel
my_model_372.configure_input(1, sensor_settings)

# Get all readings (temperature, resistance, excitation power, quadrature) from sensor
sensor_1_readings = my_model_372.get_all_input_readings(1)

# Record readings to a file
file = open("372_sensor_1_data.csv", "w")
file.write("Header information\n")
# Call readings using the keys of the returned dictionary
file.write("Temperature Reading," + str(sensor_1_readings['kelvin']) + "\n")
file.write("Resistance Reading," + str(sensor_1_readings['resistance']) + "\n")
file.write("Excitation Power," + str(sensor_1_readings['power']) + "\n")
file.write("Imaginary Part of Resistance," + str(sensor_1_readings['quadrature']) + "\n")
file.close()

```

Setting up a control loop with the model 372

```

from lakeshore import Model372, Model372HeaterOutputSettings, ↪
↪Model372ControlLoopZoneSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure output for zone mode, controlled by control input, with power up enabled ↪
↪filter enabled and a reading
# delay of 10 seconds
# Note; it's assumed that the control input is enabled and configured
heater_settings = Model372HeaterOutputSettings(my_model_372.OutputMode.ZONE, my_model_
↪372.InputChannel.CONTROL, True, True, 10)
my_model_372.configure_heater(1, heater_settings)

# Configure a relay for Warmup Heater Zone
my_model_372.set_relay_for_warmup_heater_control_zone(1)

# Set up control loop with an upper bound of 15K, a gain of 50, an integral value of ↪
↪5000, a derivative of 2000, and a
# manual output of 50%. Range is set to true, setpoint ramp rate is set to 10 seconds, ↪
↪and relay 1 is configured for
# the zone and relay 2 is not configured
control_loop_settings = Model372ControlLoopZoneSettings(15, 50.0, 5000, 2000, 50, True, ↪
↪10, True, False)
# Set control loop on output 1 (Warmup Heater) in zone 4
my_model_372.set_control_loop_parameters(1, 4, control_loop_settings)

# Create a setpoint for 5 K
my_model_372.set_setpoint_kelvin(1, 5.0)

```

(continues on next page)

(continued from previous page)

```
# Enable ramping to setpoint for output 1 at a rate of 10 Kelvin/minute
my_model_372.set_setpoint_ramp_parameter(1, True, 10)
```

Instrument class methods

```
class lakeshore.model_372.Model372(baud_rate, serial_number=None, com_port=None, timeout=2.0,
                                   ip_address=None, tcp_port=7777, **kwargs)
```

A class object representing the Lake Shore Model 372 AC bridge and temperature controller.

clear_interface()

Clears the interface.

Clears all bits in the status byte register and the standard event status register. Does not clear the instrument.

reset_instrument()

Resets the instrument to power-up settings and parameters.

```
set_display_settings(mode, number_of_fields="", displayed_info="")
```

Sets which parameters to display and how to display them.

Args:

mode (self.DisplayMode):

Sets the input to monitor on the display, or configures display for custom.

number_of_fields (self.DisplayFields):

Configures the number of display fields to include in a custom display.

displayed_info (self.DisplayInfo):

Determines whether to display information about the loop of the active scan channel or a specific heater in the bottom left of the display in custom mode.

get_display_mode()

Returns the current mode of the display.

Returns:

(self.DisplayMode):

Enumerated object representing the current mode of the display.

get_custom_display_settings()

Returns the settings of the display in custom mode.

Returns:

(dict):

mode: self.DisplayMode, number_of_fields: self.DisplayFields, displayed_info: self.DisplayInfo

get_resistance_reading(input_channel)

Returns the input reading in Ohms.

Args:

input_channel (str or int):

Specifies which input channel to read from. Options are: 1-16, or "A" (for control input).

Returns:

(float):

Sensor reading in Ohms.

get_quadrature_reading(*input_channel*)

Returns the imaginary part of the reading in Ohms. Only valid for measurement inputs.

Args:**input_channel (int):**

Specifies which input channel to read from. Options are: 1-16.

Returns:**(float):**

The imaginary part of the sensor reading, in Ohms.

get_all_input_readings(*input_channel*)

Returns the kelvin reading, resistance reading, and, if a measurement input, the quadrature reading.

Args:**input_channel (str or int):**

Specifies which input channel to read from. Options are: 1-16, or "A" (for control input).

Returns:**(dict):**

- **If measurement input:**

- {kelvin: float, resistance: float, power: float, quadrature: float}

- **If control input:**

- {kelvin: float, resistance: float, power: float}

get_input_setup_parameters(*input_channel*)

Returns the settings on the specified input.

Args:**input_channel (str or int):**

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:**input_sensor_settings (Model372InputSetupSettings):**

object of Model372InputSetupSettings representing the parameters of the excitation of the sensor on the specified channel

configure_input(*input_channel*, *settings*)

Sets the desired setup settings on the specified input.

Args:**input_channel (str or int):**

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

settings (Model372InputSetupSettings):

Object of Model372InputSetupSettings representing the parameters of the excitation of the sensor on the specified channel.

disable_input(*input_channel*)

Disables the desired input channel.

Args:

input_channel (str or int):

Specifies which input channel to disable. Options are: 1-16, or "A" (control input).

get_input_channel_parameters(*input_channel*)

Returns the settings on the specified input channel.

Args:

input_channel (str or int):

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

input_channel_settings (Model372InputChannelSettings):

Contains variables representing the different channel settings parameters.

set_input_channel_parameters(*input_channel, settings*)

Sets the desired channel settings on the specified input channel.

Args:

input_channel (str or int):

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

settings (Model372InputChannelSettings):

Defines how to set the various parameters.

get_analog_heater_output(*output_channel*)

Returns the output of the warm-up or analog/still heater.

Args:

output_channel (int):

Specifies which heater to read from. Options:

1 output 1 (warm up heater), or 2 output 2 (analog heater).

Returns:

reading (float):

Output of the analog heater being queried.

all_off()

Recreates the front panel safety feature of shutting off all heaters.

set_heater_output_range(*output_channel, heater_range*)

Sets the output range.

Args:

output_channel (int):

Specifies which heater to set. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

heater_range (Enum or bool):

Specifies the range of the output. Options: Sample Heater (Enum) - Object of type self.SampleHeaterOutputRange. Warmup Heater/Still Heater (bool) - False: output off, True: output on.

get_heater_output_range(*output_channel*)

Return's the range of the output on a given channel.

Args:

output_channel (int):

Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

Returns:**heater_range (bool or Enum):**

If channel 1 or 2, returns bool for if output is on or off. If channel 0, an object of enum type `SampleHeaterOutputRange`.

set_filter(*input_channel, state, settle_time, window*)

Sets a filter for the specified input channel.

Args:**input_channel (str or int):**

Specifies which input channel to read from. Options are: 0 (all channels/measurement inputs), 1-16, or "A" (control input).

state (bool):

Specifies whether to turn filter on or off. Options are: False for off, or True for on.

settle_time (float):

Specifies filter settle time. Options are: 1 - 200 s.

window (float):

Specifies what percent of full scale reading limits the filtering function. Options are: 1 - 80.

get_filter(*input_channel*)

Returns information about the filter set on the specified channel.

Args:**input_channel (str or int):**

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:**state (bool):**

Specifies whether to turn filter on or off.

settle_time (int):

Specifies filter settle time.

window (int):

Specifies what percent of full scale reading limits the filtering function.

set_ieee_interface_parameter(*address*)

Sets the IEEE address of the instrument.

Args:**address (int):**

Specifies the IEEE address. Options are: 1 - 30.

get_ieee_interface_parameter()

Returns the IEEE address of the instrument.

Returns:**address (int):**

The IEEE address.

get_excitation_power(*input_channel*)

Returns the most recent power calculation for the selected input channel.

Args:

input_channel (str or int):

Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

power (float):

Most recent power calculation for the input being queried.

get_heater_output_settings(*output_channel*)

Returns the mode and settings of the given output channel.

Args:

output_channel (int):

Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

Returns:

outputmode_settings (Model372HeaterOutputSettings):

Object of class Model372HeaterOutputSettings whose variables are set to reflect the current output settings of the queried heater.

configure_heater(*output_channel, settings*)

Sets up a heater output.

Analog heaters (outputs 1 and 2) might need to configure further settings in `configure_analog_heater`.

Args:

output_channel (int):

Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

settings (Model372HeaterOutputSettings):

Defines how to set the output mode settings.

set_common_mode_reduction(*state*)

Sets common mode reduction to given state for all measurement channels.

Args:

state (bool):

Sets CMR to enabled or disable. Options are: False (for disable), or True (for enable).

get_common_mode_reduction()

Returns whether CMR is set for measurement channels.

Returns:

False (boolean) if CMR is disabled, or True (boolean) if CMR is enabled.

set_scanner_status(*input_channel, status*)

Sets the scanner to the specified channel, and enables or disables auto scan.

Args:

input_channel (int):

Specifies which measurement input to set the scanner to. Options are: 1 - 16.

status (bool):

Specifies whether to turn auto scan feature on. Options are: False (disable), True (enable).

get_scanner_status()

Returns which channel the scanner is on and whether the auto scan feature is enabled.

Returns:

input_channel (int):

The measurement channel the scanner is currently on.

status (bool):

True if auto-scan in on, or False if auto-scan is off.

set_alarm_beep(status)

Enables or disables a beep for alarms.

Args:

status (bool):

False (for disable), or True (for enable).

get_alarm_beep_status()

Returns whether beep for alarms is enabled or disabled.

Returns

status (bool):

True (beep is enabled), or False (beep is disabled).

set_still_output(power)

Sets the still output of the still/analog heater to power% of full power.

Heater gets configured for still mode if not currently configured.

Args:

power (float):

Specifies the percent of full power for still output. Options are: 0 - 100.

get_still_output()

Returns the percent of full power being outputted by still heater in still mode.

Returns:

power (float):

Percent of full power being outputted by heater.

set_warmup_output(auto_control, current)

Sets up the warmup output to continuous control at the percent current specified.

Configures the warmup heater for continuous control mode from the control input.

Args:

auto_control (bool):

Specifies whether to turn on auto control. Options are: False for auto off, or True for continuous.

current (float):

Specifies percent of full current to apply to external output. Options are: 0 - 100

get_warmup_output()

Returns the control setting and percent current outputted in the warmup heater in warmup mode.

Returns:

auto_control (bool):

Specifies whether to turn on auto control. Returns: False for auto off, or True for continuous

current (float):

Specifies percent of full current to apply to external output.

set_setpoint_kelvin(*output_channel*, *setpoint*)

Sets the control set-point in Kelvin. Changes input parameters so preferred units are Kelvin.

Args:

output_channel (int):

Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

setpoint (float):

Specifies the set-point the heater ramps to, in Kelvin.

set_setpoint_ohms(*output_channel*, *setpoint*)

Sets the control set-point in Ohms. Changes input parameters so preferred units are Ohms.

Args:

output_channel (int):

Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

setpoint (float):

Specifies the set-point the heater ramps to, in Kelvin.

get_setpoint_kelvin(*output_channel*)

Returns the set-point for the given output channel in kelvin.

Changes the control input's preferred units to Kelvin as a result.

Args:

output_channel (int):

Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

Returns:

setpoint (float):

Set-point of the output in Kelvin.

get_setpoint_ohms(*output_channel*)

Returns the set-point for the given output channel in kelvin.

Changes the control input's preferred units to Kelvin as a result.

Args:

output_channel (int):

Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

Returns:

setpoint (float):

Set-point of the output in Ohms.

get_excitation_frequency(*input_channel*)

Returns the excitation frequency in Hz for either the measurement or control inputs.

Args:**input_channel (int or str):**

Specifies which input to get frequency from. Options are: 0 : measurement inputs, or "A" : control input.

Returns:**frequency (Enum):**

The excitation frequency in Hz, returned as an object of self.InputFrequency Enum type.

set_excitation_frequency(*input_channel, frequency*)

Sets the excitation frequency (in Hz) for either the measurement or control inputs.

Args:**input_channel (int or str):**

Specifies which input to get frequency from. Options are: 0 : measurement inputs, or "A" : control input.

frequency (Enum):

The excitation frequency in Hz (if float), represented as an object of type self.InputFrequency.

set_digital_output(*bit_weight*)

Sets the status of the 5 digital output lines to high or low.

Args:**bit_weight (DigitalOutputRegister):**

Determines which bits to set or reset.

get_digital_output()

Returns which digital output bits are set or reset by representing them in a binary number.

Returns:**bit_weight (DigitalOutputRegister):**

Determines which bits to set or reset.

set_interface(*interface*)

Sets the interface for the instrument to communicate over.

Args:**interface (self.Interface):**

Selects the interface based on the values as defined in the self.Interface enum class.

get_interface()

Returns the interface connected to the instrument.

Returns:**interface (self.Interface):**

Returns the interface as an object of the self.Interface enum class.

set_alarm_parameters(*input_channel*, *alarm_enable*, *alarm_settings=None*)

Sets an alarm on the specified channel as defined by parameters.

Args:

input_channel (int or str):

Defines which channel to configure an alarm on. Options are: 0 for all measurement inputs, 1 - 16, or "A" for control input.

alarm_enable (bool)

Defines whether to turn alarm on or off.

alarm_settings (Model372AlarmParameters)

Model372AlarmParameters object containing desired alarm settings. Optional if alarm is disabled.

get_alarm_parameters(*input_channel*)

Returns the parameters for the alarm set for the input at the specified channel.

Args:

input_channel (int or str):

Defines which channel to configure an alarm on. Options are: 1 - 16, or "A" for control input.

Returns:

(dict):

{*"alarm_enable"*: bool, *"alarm_settings"*: Model372AlarmParameters}

set_relay_for_sample_heater_control_zone(*relay_number*)

Configures a relay to follow the sample heater output as part of a control zone.

Settings can be further configured in `set_control_loop_zone_parameters` method.

Args:

relay_number (int):

The relay to configure. Options are: 1 or 2.

set_relay_for_warmup_heater_control_zone(*relay_number*)

Configures a relay to follow the warm-up heater output as part of a control zone.

Settings can be further configured in `set_control_loop_zone_parameters` method.

Args:

relay_number (int):

The relay to configure. Options are: 1 or 2.

get_ieee_interface_mode()

Returns the IEEE interface mode of the instrument.

Returns:

mode (self.InterfaceMode):

Returns the mode as an enum type of class `self.InterfaceMode`.

set_ieee_interface_mode(*mode*)

Sets the IEEE interface mode of the instrument.

Args:

mode (self.InterfaceMode):

Defines the mode of the instrument as an object of the enum type `Model372IEEEInterfaceMode`.

set_monitor_output_source(*source*)

Sets the source of the monitor output. Also affects the reference output.

Args:**source (self.MonitorOutputSource):**

Defines the source to run the monitor output off of.

get_monitor_output_source()

Returns the source for the monitor output.

Returns:**source (MonitorOutputSource):**

Returns the source as an object of the MonitorOutputSource class.

get_warmup_heater_setup()

Returns the settings regarding the resistance, current and units of the warmup heater (output channel 1).

Returns:**(dict):**

```
{“resistance”: float, “max_current”: float, “units”: self.HeaterOutputUnits}
```

get_sample_heater_setup()

Returns the setup of the sample heater (channel 0).

Returns:**(dict):**

```
{“resistance”: float, “units”: self.HeaterOutputUnits}
```

setup_warmup_heater(*resistance, max_current, units*)

Configures the current and power of the warmup heater (output channel 1).

The max current must not cause the heater to exceed it’s max power (calculated by $I = \sqrt{P/R}$) or it’s max voltage (calculated by $I = V/R$). Check your heater’s specifications before setting the max current, and use the lower current produced from the two calculations.

Args:**resistance (self.HeaterResistance):**

Heater load in ohms, as an object of the enum type self.HeaterResistance.

max_current (float):

User specified max current in A.

units (self.HeaterOutputUnits):

Defines which units the output is displayed in (Current (A) or Power (W)).

setup_sample_heater(*resistance, units*)

Configures the current and power of the sample heater (output channel 0.)

Args:**resistance (float):**

Heater load in ohms. Options are: 1 - 2000.

units (self.HeaterOutputUnits):

Defines which units the output is displayed in (Current (A) or Power (W)).

configure_analog_monitor_output_heater(*source, high_value, low_value, settings=None*)

Configures the still heater's analog settings for Monitor Out mode.

Can fully configure the heater by including the settings parameter, but it is recommended to configure non-analog properties of the heater through the `configure_heater` method.

Args:

source (self.InputSensorUnits):

The units to use for channel data.

high_value (float):

The data at which the output reaches +100% output.

low_value (float):

The data at which the outputs reach 0% output for unipolar output, or -100% for bipolar. output.

settings (Model372HeaterOutputSettings):

Optional if heater is already configured using `configure_heater`. Gives non-analog configurations for heater.

get_analog_monitor_output_settings()

Retrieves the analog monitor settings of output 2 configured in monitor output mode.

Returns:

(dict):

{“source”: self.InputSensorUnits, “high_value”: float, “low_value”: float}

configure_analog_heater(*output_channel, manual_value, settings=None*)

Configures the analog settings of a heater for modes other than Monitor Out.

(Use `configure_analog_monitor_out_heater` for Monitor Out mode). Can fully configure the heater by including the settings parameter, but it is recommended to first configure the heater using the `configure_heater` method before using this method.

Args:

output_channel (Model372HeaterOutput):

The output to configure.

manual_value (float):

The value of the analog output as it applies to the set analog mode.

settings (Model372HeaterOutputSettings):

Optional if heater is already configured using `configure_heater`. Gives non-analog configurations for heater.

get_analog_manual_value(*output_channel*)

Returns the manual value of an analog heater.

The manual value is the analog value used for Open Loop, Closed Loop, Warm Up, or Still mode.

Args:

output_channel (int):

The analog output to query. Options are: 1 (Warm up heater), or 2 (Still heater).

Returns:

(float):

The manual analog value for the heater.

set_website_login(*username, password*)

Sets the username and password to connect instrument to website.

Args:

username (str):

Username to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

password (str):

Password to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

get_website_login()

Returns the set username and password for web login for the instrument.

Returns:

username (str):

The current set username for the web login

password (str):

The current set password for the web login

get_control_loop_zone_parameters(*output_channel, zone*)

Returns the settings parameters of the control loop on the specified output channel and zone.

Args:

output_channel (int):

Channel of the heater being queried. Options are: 0 for sample heater, or 1 for warm-up heater.

zone (int):

Control loop zone to configure. Options are: 1 - 10.

Returns:

settings (Model372ControlLoopZoneSettings):

An object of the Model372ControlLoopZoneSettings class containing information of the settings in the values of its variables.

set_control_loop_parameters(*output_channel, zone, settings*)

Returns the parameters of the control loop set in the specified zone for the specified heater output.

Args:

output_channel (int):

Channel of the heater being queried. Options are: 0 for sample heater, or 1 for warm-up heater.

zone (int):

Control loop zone to configure. Options are: 1 - 10.

settings (Model372ControlLoopZoneSettings):

An object of the Model372ControlLoopZoneSettings with the variable set to configure the desired settings.

get_reading_status(*input_channel*)

Returns any flags raised during a measurement reading.

Args:

input_channel (str or int):

The input whose reading status is being queried. Options are: 1 - 16, or "A" (control input).

Returns:

bit_states (dict):

Dictionary containing the names of the flag and a boolean value corresponding to if the flag is raised or not.

class HeaterResistance(*values)

Enumerator used to represent the different heater resistances.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class Polarity(*values)

Enumerator for unipolar or bipolar output operation.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

class RelayControlAlarm(*values)

Enumeration of the setting determining which alarm(s) cause a relay to close in alarm mode.

conjugate()

Returns self, the complex conjugate of any int.

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

as_integer_ratio()

Return a pair of integers, whose ratio is equal to the original int.

The ratio is in lowest terms and has a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

is_integer()

Returns True. Exists for duck type compatibility with float.is_integer.

real

the real part of a complex number

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

denominator

the denominator of a rational number in lowest terms

clear_interface_command()

Clears the bits in the SBR, SESR, OER, and terminates all operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register. Terminates all pending operations. Clears the interface, but not the controller.

command(*commands, check_errors=True)

Sends an SCPI command or multiple commands to the instrument.

Args:

commands (str):

A serial command.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.
Optional Parameter.

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None*, *com_port=None*, *baud_rate=None*, *data_bits=None*, *stop_bits=None*, *parity=None*, *timeout=None*, *handshaking=None*, *flow_control=None*)

Establish a serial USB connection.

delete_curve(*curve*)

Deletes the user curve.

Args:**curve (int):**

Specifies a user curve to delete.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

get_alarm_status(*channel*)

Returns the high state and low state of the alarm for the specified channel.

Args:**channel (str or int)**

Specifies which input channel to read from.

Returns:**(dict[str, bool]):**

{“high_state_enabled”: bool, “low_state_enabled” bool}

get_control_setpoint(*output*)

Returns the value for a given control output.

Args:**output (int):**

Specifies which output’s control loop to query (1 or 2).

Returns:**(float):**

The value for the set-point (in the preferred units of the control loop sensor).

get_curve(*curve*)

Returns a list of all the data points in a particular curve.

Args:**curve (int):**

Specifies which curve to set.

Returns:

list[tuple[float, float] | tuple[float, float, float]]:

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_data_point(*curve*, *index*)

Returns a standard or user curve data point.

Args:

curve (int):

Specifies which curve to query.

index (int):

Specifies the points index in the curve.

Returns:

tuple[float, float] | tuple[float, float, float]:

(sensor_units: float, temp_value: float, curvature_value: float (optional)).

get_curve_header(*curve_number*)

Returns parameters set on a particular user curve header.

Args:

curve_number (int):

Specifies a curve to retrieve.

Returns:

(CurveHeader):

A CurveHeader class object containing the curve information.

get_display_field_settings(*field*)

Returns the settings of the specified display field when display is in Custom mode.

Args:

field (int)

Defines which field of the display to retrieve settings from.

Returns:

(dict[str, LegacyTemperatureControllerEnums.InputChannel |

LegacyTemperatureControllerEnums.DisplayFieldUnits]):

See set_display_field_settings method. {"input_channel": InputChannel, "display_units": DisplayFieldUnits}

get_heater_output(*output*)

Sample heater output in percent, scale is dependent upon the instrument used and heater configuration.

Args:

output (int):

Heater output to query.

Returns:

(float):

percent of full scale current/voltage/power.

get_heater_pid(*output*)

Returns the closed loop control parameters of the heater output.

Args:**output (int):**

Specifies which output's control loop to query.

Returns:**(dict):**

{"gain": float, "integral": float, "derivative": float}

gain: Proportional term in PID control. integral: Integral term in PID control. derivative: Derivative term in PID control.

get_heater_status(*output*)

Returns the heater error code state, error is cleared upon querying the heater status.

Args:**output (int):**

Specifies which heater output to query (1 or 2).

Returns:

(TemperatureControllerEnums.HeaterError):

Object of instrument's HeaterError type.

get_ieee_488()

Returns the IEEE address set.

Returns:**(int):**

Address 1-30 (0 and 31 reserved).

get_input_curve(*input_channel*)

Returns the curve number being used for a given input.

Args:**input_channel (str or int):**

Specifies which input to query.

Returns:**(int):**

Curve number (0-59).

get_kelvin_reading(*input_channel*)

Returns the temperature value in kelvin of the given channel.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:**float:**

The temperature value.

get_keypad_lock()

Returns the state of the keypad lock and the lock-out code.

Returns:

(dict[str, bool | int]):
{"state": bool, "code": int}

get_led_state()

Returns whether front panel LEDs are enabled.

Returns:

(bool):
Specifies whether front panel LEDs are functional. False if disabled, True enabled.

get_manual_output(*output*)

Returns the manual output value in percent.

Args:

output (int):
Specifies output to query.

Returns:

(float):
Manual output percent.

get_min_max_data(*input_channel*)

Returns the minimum and maximum data from an input.

Args:

input_channel (str):
Specifies which input to query.

Returns:

(dict[str, float]):
{"minimum": float, "maximum": float}

get_relay_alarm_control_parameters(*relay_number*)

Returns the relay alarm configuration for either of the two configurable relays.

Relay must be configured for alarm mode to retrieve parameters.

Args:

relay_number (int)
Specifies which relay to query. Options are: 1 or 2.

Returns:

(dict[str, str | LegacyTemperatureControllerEnums.RelayControlAlarm]):
{"activating_input_channel": str, "alarm_relay_trigger_type": RelayControlAlarm}

get_relay_control_mode(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):
Specifies which relay to query. Options are: 1 or 2.

Returns:

(LegacyTemperatureControllerEnums.RelayControlMode):
The configured mode of the relay. Represented as a member of the instrument's RelayControlMode IntEnum class.

get_relay_status(*relay_channel*)

Returns whether the relay at the specified channel is On or Off.

Args:**relay_channel (int):**

The relay channel to query.

Returns:**(bool):**

True if relay is on, False if relay is off.

get_remote_interface_mode()

Returns the state of the interface mode.

Returns:**(TemperatureControllerEnums.InterfaceMode):**

A member of the instrument's InterfaceMode IntEnum class.

get_self_test()

Instrument self test result completed at power up.

Returns:**(bool):**

True = errors found. False = no errors found.

get_sensor_name(*input_channel*)

Returns the name of the sensor on the specified channel.

Args:**input_channel (str or int):**

Specifies which input_channel channel to read from.

Returns:**name (str):**

Name associated with the sensor.

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor's units.

Args:**input_channel (str | int):**

Selects the channel to retrieve reading.

Returns:**(float):**

The raw sensor reading in the units of the connected sensor.

get_service_request()

Returns the status byte register bits and their values as a class instance.

get_setpoint_ramp_parameter(*output*)

Returns the control loop parameters of a particular output.

Args:

output (int):

Specifies which output's control loop to return.

Returns:

(dict[str, bool | float]):

{“ramp_enable”: bool, “rate_value”: float}

get_setpoint_ramp_status(*output*)

Returns whether the set-point is ramping.

Args:

output (int):

Specifies which output's control loop to query.

Returns:

(bool):

Ramp status. False = Not ramping, True = Ramping.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

Returns:

StandardEventRegister:

The object with the names of the standard event enable register bits and their values.

get_status_byte()

Returns the status flag bits as a class instance without resetting the register.

get_temperature_limit(*input_channel*)

Returns the value of the temperature limit in kelvin.

Args:

input_channel (str or int):

Specifies which input to query.

Returns:

float:

The value of the temperature limit.

query(queries*, *check_errors*=True)**

Send a query to the instrument and return the response.

Args:

queries (str):

A serial query ending in a question mark.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

str:

The instrument query response as a string.

reset_alarm_status()

Clears the high and low status of all alarms.

reset_min_max_data()

Resets the minimum and maximum input data.

set_control_setpoint(*output, value*)

Set set-point for specific output's control loop.

Control settings, that is, P, I, D, and Set-point, are assigned to outputs, which results in the settings being applied to the control loop formed by the output and its control input.

Args:**output (int):**

Specifies which output's control loop to configure.

value (float):

The value for the set-point (in the preferred units of the control loop sensor).

set_curve(*curve, data_points*)

Method to define a user curve using a list of data points.

Args:**curve (int):**

Specifies which curve to set.

data_points (list[tuple[float, float] | tuple[float, float, float]]):

A list containing every point in the curve represented as a tuple. (sensor_units: float, temp_value: float, curvature_value: float (optional)).

set_curve_data_point(*curve, index, sensor_units, temperature, curvature=None*)

Configures a user curve point.

Args:**curve (int or str):**

Specifies which curve to configure.

index (int):

Specifies the points index in the curve.

sensor_units (float):

Specifies sensor units for this point to 6 digits.

temperature (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

curvature (float):

Specify only if the point is part of a cubic spindle curve. The curvature value scale used to calculate spindle coefficients to 6 digits. Optional parameter.

set_curve_header(*curve_number, curve_header*)

Configures the user curve header.

Args:**curve_number (int):**

Specifies which curve to configure.

curve_header (CurveHeader):

Instrument's CurveHeader class object containing the desired curve information.

set_display_field_settings(*field*, *input_channel*, *display_units*)

Configures a display field when the display is in custom mode.

Args:

field (int):

Defines which field of the display is being configured.

input_channel (LegacyTemperatureControllerEnums.InputChannel):

Defines which input to display. A member of the instrument's InputChannel IntEnum class.

display_units (LegacyTemperatureControllerEnums.DisplayFieldUnits):

Defines which units to display reading in. A member of the instrument's DisplayFieldUnits IntEnum class.

set_heater_pid(*output*, *gain*, *integral*, *derivative*)

Configure the closed loop control parameters of the heater output.

Args:

output (int):

Specifies which output's control loop to configure.

gain (float):

Proportional term in PID control. This controls how strongly the control output reacts to the present error.

integral (float):

Integral term in PID control. This controls how strongly the control output reacts to the past error history.

derivative (float):

Derivative term in PID control. This value controls how quickly the present field set point will transition to a new set-point.

set_ieee_488(*address*)

Specifies the IEEE address.

Args:

address (int):

1-30 (0 and 31 reserved).

set_input_curve(*input_channel*, *curve_number*)

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str or int):

Specifies which input to configure.

curve_number (int):

0 = none, 1-20 = standard curves, 21-59 = user curves.

set_keypad_lock(*state*, *code*)

Locks or unlocks front panel keypad (except for alarms and disabling heaters).

Args:

state (bool):

Sets the keypad to locked or unlocked. Options are: False for unlocked or True for locked.

code (int):

Specifies 3 digit lock-out code. Options are: 000 - 999.

set_led_state(*state*)

Sets the front panel LEDs to on or off.

Args:**state (bool):**

Sets the LEDs to functional or nonfunctional. False if disabled, True enabled.

set_manual_output(*output*, *value*)

When instrument is in closed loop PID, Zone, or Open Loop modes a manual output may be set.

Args:**output (int):**

Specifies output to configure.

value (float):

Specifies value for manual output in percent.

set_relay_alarms(*relay_number*, *activating_input_channel*, *alarm_relay_trigger_type*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:**relay_number (int):**

The relay to configure. Options are: 1 or 2.

activating_input_channel (str or int):

Specifies which input alarm activates the relay.

alarm_relay_trigger_type (LegacyTemperatureControllerEnums.RelayControlAlarm):

Specifies the type of alarm that triggers the relay.

set_remote_interface_mode(*mode*)

Places the instrument in one of three interface modes.

Args:**mode (TemperatureControllerEnums.InterfaceMode):**

A member of the instrument's InterfaceMode IntEnum class.

set_sensor_name(*input_channel*, *sensor_name*)

Sets a given name to a sensor on the specified channel.

Args:**input_channel (str or int):**

Specifies which input_channel channel to read from.

sensor_name (str):

Name user wants to give to the sensor on the specified channel.

set_service_request(*register_mask*)

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:**register_mask (service_request_enable):**

A service_request_enable class object with all bits configured.

set_setpoint_ramp_parameter(*output*, *ramp_enable*, *rate_value*)

Sets the control loop of a particular output.

Args:

output (int):

Specifies which output's control loop to configure.

ramp_enable (bool):

Specifies whether ramping is off or on (False = Off or True = On).

rate_value (float):

Specifies set-point ramp rate in kelvin per minute. The rate is always positive but will respond to ramps up or down. A rate of 0 is interpreted as infinite, and will respond as if set-point ramping were off. (0.1 to 100)

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (StandardEventRegister):

A StandardEventRegister class object with all bits set to a value.

set_temperature_limit(*input_channel*, *limit*)

After a set temperature limit is exceeded, all control outputs will shut down.

Args:

input_channel (str or int):

Specifies which input to configure.

limit (float):

The temperature limit in kelvin for which to shut down all control outputs when exceeded. A limit of zero will turn the feature off.

turn_relay_off(*relay_number*)

Turns the specified relay off.

Args:

relay_number (int):

The relay to turn off. Options are: 1 or 2.

turn_relay_on(*relay_number*)

Turns the specified relay on.

Args:

relay_number (int):

The relay to turn on. Options are: 1 or 2.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Instrument settings classes and registers

```
class lakeshore.model_372.Model372InputChannelSettings(enable, dwell_time, pause_time,
                                                    curve_number,
                                                    temperature_coefficient=None)
```

Class object representing parameters for the channel settings of an self.InputChannel.

```
__init__(enable, dwell_time, pause_time, curve_number, temperature_coefficient=None)
```

The constructor for Model372InputChannelSettings class.

Args:

enable (bool):

Whether to enable or disable input.

dwell_time (int):

Specifies a value for the auto-scanning dwell time in seconds. Not applicable to control input. Options are: 1 to 200 s.

pause_time (int):

Specifies a value for the change pause time in seconds. Options are: 3 to 200 s.

curve_number (int):

Specifies which calibration curve to use on input sensor. Options are: 0 (none), or 1 - 59.

temperature_coefficient (self.CurveTemperatureCoefficient):

Sets coefficient for temperature control if no curve is selected.

```
class lakeshore.model_372.Model372InputSetupSettings(mode, excitation_range, auto_range,
                                                    current_source_shunted, units,
                                                    resistance_range=None)
```

Class object representing parameters for the sensor and measurement settings of an self.InputChannel.

```
__init__(mode, excitation_range, auto_range, current_source_shunted, units, resistance_range=None)
```

The constructor for Model372InputSetupSettings class.

Args:

mode (self.SensorExcitationMode):

Determines whether to use current or voltage for sensor excitation.

excitation_range (IntEnum):

The voltage or current (depending on mode) excitation range.

auto_range (Model372AutoRangeMode):

Specifies whether auto range is Off, Auto-ranging Current, or in ROX 102B mode.

current_source_shunted (bool):

Specifies whether the current source is shunted. If current source is shunted, excitation is off. If current source is not shunted, excitation is on.

units (self.InputSensorUnits):

Specifies the preferred units, Kelvin or Ohms, for the sensor.

resistance_range (Model372MeasurementInputResistance):

For measurement inputs only, specifies the measurement input resistance range.

```
class lakeshore.model_372.Model372HeaterOutputSettings(output_mode, input_channel,
                                                    powerup_enable, reading_filter, delay,
                                                    polarity=None)
```

Class object representing parameters to configure Heater Output Settings.

`__init__(output_mode, input_channel, powerup_enable, reading_filter, delay, polarity=None)`

The constructor for Model372HeaterOutputSettings class.

Args:

output_mode (self.OutputMode):

The control or output mode to configure the heater for. Defines how the output is controlled.

input_channel (self.InputChannel):

Which input to control output from in a control loop.

powerup_enable (bool):

Specifies whether output stays on after powerup cycle. True if enabled, False if disabled.

reading_filter (bool):

Specifies whether readings are filtered on unfiltered. True if filtered, False if unfiltered.

delay (int):

Specifies delay in seconds for set-point during AutoScanning. Options are: 1 - 255.

polarity (self.Polarity):

Specifies output polarity. Not applicable to warmup heater.

`class lakeshore.model_372.Model372ControlLoopZoneSettings(upper_bound, p_value, i_value, d_value, manual_output, heater_range, ramp_rate, relay_1, relay_2)`

Defines the parameters to set up a Control Loop.

`__init__(upper_bound, p_value, i_value, d_value, manual_output, heater_range, ramp_rate, relay_1, relay_2)`

The constructor for Model372ControlLoopZoneSettings class.

Args:

upper_bound (float):

Upper bound setpoint in Kelvin.

p_value (float):

The gain for a PID system. Options are: 0.0 - 1000.

i_value (float):

The integral value for a PID system. Options are: 0 - 10000.

d_value (float):

The rate for a PID system. Options are: 0 - 2500.

manual_output (float):

Percentage full scale manual output.

heater_range (float or bool):

Heater range for the control zone. Entered as a float for the sample heater. Entered as a bool for the warm-up heater.

ramp_rate (float):

Specifies ramp rate for this zone.

relay_1 (bool):

Specifies if relay 1 is on or off. Only applicable if relay is configured in zone mode and relay's control output matches configured output.

relay_2 (bool):

Specifies if relay 2 is on or off. Only applicable if relay is configured in zone mode and relay's control output matches configured output.

class lakeshore.model_372.**Model372AlarmParameters**(*high_value, low_value, deadband, latch_enable, audible=None, visible=None*)

Sets up an alarm for an input channel.

__init__(*high_value, low_value, deadband, latch_enable, audible=None, visible=None*)

The constructor for Model372AlarmParameters class.

Args:

high_value (int):

Sets value for source to be checked against to set high alarm.

low_value (int):

Sets value for source to be checked against to set low alarm.

deadband (int):

Sets value that source must change outside an alarm condition to deactivate an unlatched alarm.

latch_enable (bool):

Specifies if alarm is latched or not.

audible (bool):

Specifies if an alarm is audible or not.

visible (bool):

Specifies if an alarm is visible via LED on front panel or not.

lakeshore.model_372.Model372CurveHeader

alias of *CurveHeader*

class lakeshore.model_372.**CurveHeader**(*curve_name, serial_number, curve_data_format, temperature_limit, coefficient*)

A class to configure the temperature sensor curve header parameters.

__init__(*curve_name, serial_number, curve_data_format, temperature_limit, coefficient*)

Constructor for CurveHeader class.

Args:

curve_name (str):

Specifies curve name (limit of 15 characters).

serial_number (str):

Specifies curve serial number (limit of 10 characters).

curve_data_format (TemperatureControllerEnums.CurveFormat):

Member of the instrument's CurveFormat IntEnum class. Specifies the curve data format.

temperature_limit (float):

Specifies the curve temperature limit in Kelvin.

coefficient (TemperatureControllerEnums.CurveTemperatureCoefficient):

Member of instrument's CurveTemperatureCoefficient IntEnum class. Specifies the curve temperature coefficient.

lakeshore.model_372.Model372StandardEventRegister

alias of *StandardEventRegister*

class lakeshore.model_372.**StandardEventRegister**(*operation_complete, query_error, execution_error, command_error, power_on*)

Class object representing the standard event register.

```
class lakeshore.model_372.Model372ReadingStatusRegister(current_source_overload,
                                                    voltage_common_mode_stage_overload,
                                                    voltage_mixer_stage_overload,
                                                    voltage_differential_stage_overload,
                                                    resistance_over, resistance_under,
                                                    temperature_over, temperature_under)
```

Class object representing the reading status of an input.

While not a literal register, the return of an int representation of multiple booleans makes it convenient to represent this functionality as a register.

```
class lakeshore.model_372.Model372StatusByteRegister(warmup_heater_ramp_done,
                                                    valid_reading_control_input,
                                                    valid_reading_measurement_input, alarm,
                                                    sensor_overload, event_summary,
                                                    request_service_master_summary_status,
                                                    sample_heater_ramp_done)
```

Class representing the status byte register.

```
class lakeshore.model_372.Model372ServiceRequestEnable(warmup_heater_ramp_done,
                                                    valid_reading_control_input,
                                                    valid_reading_measurement_input, alarm,
                                                    sensor_overload, event_summary,
                                                    sample_heater_ramp_done)
```

Class representing the status byte register.

Enumeration objects

This section describes the Enum type objects that have been created to name various settings of the model 372 that are represented as an int or single character to the instrument. The purpose of these objects is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

```
class lakeshore.temperature_controllers_enums.TemperatureControllerEnums
```

Class containing the enums relevant to temperature controllers.

```
class InterfaceMode(*values)
```

Enumeration for the mode of the remote interface.

```
LOCAL = 0
```

```
REMOTE = 1
```

```
REMOTE_LOCAL_LOCK = 2
```

```
class HeaterError(*values)
```

Enumeration for possible errors flagged by the heater.

```
NO_ERROR = 0
```

```
HEATER_OPEN_LOAD = 1
```

```
HEATER_SHORT = 2
```

```
class CurveFormat(*values)
```

Enumerations specify formats for temperature sensor curves.

```
MILLIVOLT_PER_KELVIN = 1
```

```
VOLTS_PER_KELVIN = 2
```

```
OHMS_PER_KELVIN = 3
```

```
LOG_OHMS_PER_KELVIN = 4
```

```
class CurveTemperatureCoefficient(*values)
```

Enumerations specify positive/negative temperature sensor curve coefficients.

```
NEGATIVE = 1
```

```
POSITIVE = 2
```

```
class BrightnessLevel(*values)
```

Enumerator to specify the brightness level of an instrument display.

```
QUARTER = 0
```

```
HALF = 1
```

```
THREE_QUARTERS = 2
```

```
FULL = 3
```

```
class AutotuneMode(*values)
```

Enumerator used to represent the different autotune control modes.

```
P_ONLY = 0
```

```
P_I = 1
```

```
P_I_D = 2
```

```
class DiodeCurrent(*values)
```

Enumerator used to represent diode current ranges.

```
TEN_MICROAMPS = 0
```

```
ONE_MILLIAMP = 1
```

```
class HeaterOutputUnits(*values)
```

Enumerator used to represent heater output unit settings.

```
CURRENT = 1
```

```
POWER = 2
```

```
class Interface(*values)
```

Enumerator used to represent remote interface communication methods.

```
USB = 0
```

```
ETHERNET = 1
```

```
IEEE488 = 2
```

```
class InputSensorUnits(*values)
```

Enumerator used to represent temperature sensor unit options.

```
KELVIN = 1
```

CELSIUS = 2

SENSOR = 3

class ControlTypes(*values)

Enumerator used to represent the control type settings.

AUTO_OFF = 0

CONTINUOUS = 1

class LanStatus(*values)

Represents the different status states for the lan connection.

STATIC_IP = 0

DHCP = 1

AUTO_IP = 2

ADDRESS_NOT_ACQUIRED_ERROR = 3

DUPLICATE_INITIAL_IP_ERROR = 4

DUPLICATE_ONGOING_IP_ERROR = 5

CABLE_UNPLUGGED = 6

MODULE_ERROR = 7

ACQUIRING_ADDRESS = 8

ETHERNET_DISABLED = 9

class DisplayFields(*values)

Enumeration of the possible number of fields to include in a custom display mode.

LARGE_2 = 0

LARGE_4 = 1

SMALL_8 = 2

class DisplayFieldsSize(*values)

Enumeration of the display fields when mode is set to all inputs.

SMALL = 0

LARGE = 1

class RelayControlMode(*values)

Relay operating mode enumeration.

RELAY_OFF = 0

RELAY_ON = 1

ALARMS = 2

class lakeshore.model_372.Model372Enums

Class containing the enums relevant to the Model 372.

class OutputMode(*values)

Enumeration of the different modes for heater output setup.

OFF = 0

MONITOR_OUT = 1

OPEN_LOOP = 2

ZONE = 3

STILL = 4

CLOSED_LOOP = 5

WARMUP = 6

class InputChannel(*values)

Enumeration of the input channels of the Model 372.

NONE = 0

ONE = 1

TWO = 2

THREE = 3

FOUR = 4

FIVE = 5

SIX = 6

SEVEN = 7

EIGHT = 8

NINE = 9

TEN = 10

ELEVEN = 11

TWELVE = 12

THIRTEEN = 13

FOURTEEN = 14

FIFTEEN = 15

SIXTEEN = 16

CONTROL = 'A'

class SensorExcitationMode(*values)

Enumeration of the possible excitation modes for an input sensor.

VOLTAGE = 0

CURRENT = 1

class AutoRangeMode(*values)

Enumeration for the possible modes of the auto ranging feature.

ROX102B mode is a special auto-ranging mode that applies only to Lake Shore ROX-102B sensor.

OFF = 0

CURRENT = 1

ROX102B = 2

class InputSensorUnits(*values)

Enumeration of the units to handle input readings and display in.

KELVIN = 1

OHMS = 2

class MonitorOutputSource(*values)

Enumeration of the source for an output to monitor.

OFF = 0

CS_NEG = 1

CS_POS = 2

VCM_NEG = 3

VCM_POS = 4

VDIF = 5

VAD_MEASUREMENT = 6

VAD_CONTROL = 7

class RelayControlMode(*values)

Enumeration of the control modes of the configurable relays of the 372.

RELAY_OFF = 0

RELAY_ON = 1

ALARMS = 2

SAMPLE_HEATER_ZONE = 3

WARMUP_HEATER_ZONE = 4

class DisplayMode(*values)

Enumeration of the possible information to display.

MEASUREMENT_INPUT = 0

CONTROL_INPUT = 1

```
CUSTOM = 2
```

class DisplayInfo(*values)
Enumeration of the information to a display in the bottom left of the custom display mode.

```
NONE = 0
SAMPLE_HEATER = 1
WARMUP_HEATER = 2
ACTIVE_SCAN_CHANNEL = 3
```

class CurveFormat(*values)
Enumeration of the units to use in a calibration curve.

```
OHM_PER_KELVIN = 3
LOGOHM_PER_KELVIN = 4
OHM_PER_KELVIN_CUBIC_SPLINE = 7
```

class DisplayFieldUnits(*values)
Enumeration for the possible units to display in a single display field.

```
KELVIN = 1
OHMS = 2
QUADRATURE = 3
MINIMUM_DATA = 4
MAXIMUM_DATA = 5
SENSOR_NAME = 6
```

class SampleHeaterOutputRange(*values)
Enumeration of the output range of the sample heater (output 0).

```
OFF = 0
RANGE_31_POINT_6_MICRO_AMPS = 1
RANGE_100_MICRO_AMPS = 2
RANGE_316_MICRO_AMPS = 3
RANGE_1_MILLI_AMP = 4
RANGE_3_POINT_16_MILLI_AMPS = 5
RANGE_10_MILLI_AMPS = 6
RANGE_31_POINT_6_MILLI_AMPS = 7
RANGE_100_MILLI_AMPS = 8
```

class InputFrequency(*values)
Defines the enumeration of the excitation frequency of an input.

```
FREQUENCY_9_POINT_8_HZ = 1
FREQUENCY_13_POINT_7_HZ = 2
FREQUENCY_16_POINT_2_HZ = 3
FREQUENCY_11_POINT_6_HZ = 4
FREQUENCY_18_POINT_2_HZ = 5
```

```
class MeasurementInputVoltageRange(*values)
```

Enumerates the possible voltage ranges for a measurement input.

```
RANGE_2_MICRO_VOLTS = 1
RANGE_6_POINT_32_MICRO_VOLTS = 2
RANGE_20_MICRO_VOLTS = 3
RANGE_63_POINT_2_MICRO_VOLTS = 4
RANGE_200_MICRO_VOLTS = 5
RANGE_632_MICRO_VOLTS = 6
RANGE_2_MILLI_VOLTS = 7
RANGE_6_POINT_32_MILLI_VOLTS = 8
RANGE_20_MILLI_VOLTS = 9
RANGE_63_POINT_2_MILLI_VOLTS = 10
RANGE_200_MILLI_VOLTS = 11
RANGE_632_MILLI_VOLTS = 12
```

```
class MeasurementInputCurrentRange(*values)
```

Enumeration of the current range of a measurement input.

```
RANGE_1_PICO_AMP = 1
RANGE_3_POINT_16_PICO_AMPS = 2
RANGE_10_PICO_AMPS = 3
RANGE_31_POINT_6_PICO_AMPS = 4
RANGE_100_PICO_AMPS = 5
RANGE_316_PICO_AMPS = 6
RANGE_1_NANO_AMP = 7
RANGE_3_POINT_16_NANO_AMPS = 8
RANGE_10_NANO_AMPS = 9
RANGE_31_POINT_6_NANO_AMPS = 10
RANGE_100_NANO_AMPS = 11
```

```
RANGE_316_NANO_AMPS = 12
RANGE_1_MICRO_AMP = 13
RANGE_3_POINT_16_MICRO_AMPS = 14
RANGE_10_MICRO_AMPS = 15
RANGE_31_POINT_6_MICRO_AMPS = 16
RANGE_100_MICRO_AMPS = 17
RANGE_316_MICRO_AMPS = 18
RANGE_1_MILLI_AMP = 19
RANGE_3_POINT_16_MILLI_AMPS = 20
RANGE_10_MILLI_AMPS = 21
RANGE_31_POINT_6_MILLI_AMPS = 22
```

```
class ControlInputCurrentRange(*values)
```

Enumeration of the current range of the control input.

```
RANGE_316_PICO_AMPS = 1
RANGE_1_NANO_AMP = 2
RANGE_3_POINT_16_NANO_AMPS = 3
RANGE_10_NANO_AMPS = 4
RANGE_31_POINT_6_NANO_AMPS = 5
RANGE_100_NANO_AMPS = 6
```

```
class MeasurementInputResistance(*values)
```

Enumeration of the resistance range of a measurement input.

```
RANGE_2_MILLI_OHMS = 1
RANGE_6_POINT_32_MILLI_OHMS = 2
RANGE_20_MILLI_OHMS = 3
RANGE_63_POINT_2_MILLI_OHMS = 4
RANGE_200_MILLI_OHMS = 5
RANGE_632_MILLI_OHMS = 6
RANGE_2_OHMS = 7
RANGE_6_POINT_32_OHMS = 8
RANGE_20_OHMS = 9
RANGE_63_POINT_2_OHMS = 10
RANGE_200_OHMS = 11
```

```
RANGE_632_OHMS = 12
RANGE_2_KIL_OHMS = 13
RANGE_6_POINT_32_KIL_OHMS = 14
RANGE_20_KIL_OHMS = 15
RANGE_63_POINT_2_KIL_OHMS = 16
RANGE_200_KIL_OHMS = 17
RANGE_632_KIL_OHMS = 18
RANGE_2_MEGA_OHMS = 19
RANGE_6_POINT_32_MEGA_OHMS = 20
RANGE_20_MEGA_OHMS = 21
RANGE_63_POINT_2_MEGA_OHMS = 22
```

```
class HeaterOutput(*values)
    Enumeration of heater output.
    WARM_UP_HEATER = 1
    STILL_HEATER = 2
```

2.4.5 Temperature Monitors

Model 224 Temperature Monitor

The Lake Shore Model 224 measures up to 12 temperature sensor channels.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example Scripts

Configuring the model 224 with a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and
↳ serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a
↳ temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↳ VOLTS_PER_KELVIN, 300.0,
                                     myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)
```

(continues on next page)

(continued from previous page)

```

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set
↳to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.
↳The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,
↳30, "SN123", (276, 10),
                                     (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can
↳then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that
↳curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)

```

Instrument class methods

```

class lakeshore.model_224.Model224(serial_number=None, com_port=None, baud_rate=57600,
                                   data_bits=7, stop_bits=1, parity='O', flow_control=False,
                                   handshaking=False, timeout=2.0, ip_address=None, tcp_port=7777,
                                   **kwargs)

```

A class object representing the Lake Shore Model 224 temperature monitor.

command(*commands, check_errors=True)

Sends an SCPI command or multiple commands to the instrument.

Args:

commands (str):

A serial command.

check_errors (bool):

Chooses whether to check for and raise errors after sending a command. True by default. kwarg.
Optional Parameter

query(*queries, check_errors=True)

Send a query to the instrument and return the response.

Args:

queries (str):

A serial query ending in a question mark.

Returns:

The instrument query response as a string.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:

register_mask (Model224StandardEventRegister):

An StandardEventRegister class object with all bits set to a value.

clear_interface_command()

Clears the bits of the interface and terminates all pending operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register, and terminates all pending operations. Clears the interface, but not the controller.

reset_instrument()

Sets controller parameters to power-up settings.

set_service_request(*register_mask*)

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:

register_mask (Model224ServiceRequestRegister):

A Model224ServiceRequestRegister class object with all bits configured.

get_service_request()

Returns the status byte register bits and their values as a class instance.

get_status_byte()

Returns the status flag bits as a class instance without resetting the register.

get_self_test()

Instrument self test result completed at power up.

Returns:

test_errors (bool):

True means errors found, and False means no errors found.

set_wait_to_continue()

Causes the IEEE-488 interface to hold off until all pending operations have been completed.

This has the same function as the `set_operation_complete()` method, except that it does not set the Operation Complete event bit in the Event Status Register.

set_to_factory_defaults()

Sets all the settings and configurations to their factory default values.

get_reading_status(*input_channel*)

Returns the reading status of any input status flags that may be set.

Args:

input_channel (str):

The input to check for reading status flags. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:**(dict):**

```
{“invalid_reading”: bool, “temperature_under_range”: bool, “temperature_over_range”: bool,
“sensor_units_zero”: bool, “sensor_units_over_range”: bool}
```

get_kelvin_reading(*input_channel*)

Returns the temperature value in kelvin of either channel.

Args:**input_channel:**

Selects the channel to retrieve measurement. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:**(float):**

The reading of the sensor in kelvin.

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor’s units.

Args:**input_channel:**

Selects the channel to retrieve measurement. Options are: Options are: A, B, C(1 - 5), D(1 - 5).

Returns:**reading (float):**

The raw sensor reading in the units of the connected sensor.

get_celsius_reading(*input_channel*)

Returns the given input’s temperature reading in degrees Celsius.

Args:**input_channel (str):**

Selects input to retrieve measurement from. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:**(float):**

Temperature readings in degrees Celsius.

get_all_inputs_celsius_reading()

Returns the temperature reading in degrees Celsius of all the inputs.

Returns:**(dict):**

```
{“input_a_reading”: float, “input_b_reading”: float, “input_c1_reading”: float, “in-
put_c2_reading”: float, “input_c3_reading”: float, “input_c4_reading”: float, “in-
put_c5_reading”: float, “input_d1_reading”: float, “input_d2_reading”: float, “in-
put_d3_reading”: float, “input_d4_reading”: float, “input_d5_reading”: float}
```

set_input_diode_excitation_current(*input_channel*, *diode_current*)

Sets the excitation current of a diode sensor.

Input must be configured for a diode sensor for command to work. Current defaults to 10uA.

Args:

input_channel (str):

The input to configure the diode excitation current for.

diode_current (Model224DiodeExcitationCurrent):

The excitation current for the diode sensor.

get_input_diode_excitation_current(*input_channel*)

Returns the diode excitation current for the given diode sensor.

Args:

input_channel (str):

The diode sensor input to query the current of.

Returns:

diode_current (Model224DiodeExcitationCurrent):

A member of the Model224DiodeExcitationCurrent enum class.

set_sensor_name(*channel, sensor_name*)

Sets a given name to a sensor on the specified channel.

Args:

channel (str):

Specifies which the sensor to name is on. Options are: A, B, C(1 - 5), D(1 - 5).

sensor_name(str):

Name user wants to give to the sensor on the specified channel.

get_sensor_name(*channel*)

Returns the name of the sensor on the specified channel.

Args:

channel (str):

Specifies which input sensor to retrieve name of. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

name (str):

Name associated with the sensor.

set_display_contrast(*contrast_level*)

Sets the contrast level for the front panel display.

Args:

contrast_level (int):

Display contrast for the front panel LCD screen. Options are: 1 - 32.

get_display_contrast()

Returns the contrast level of front panel display.

Returns:

(int):

Contrast level of the front panel LCD screen.

set_ieee_488(*address*)

Specifies the IEEE address.

Args:

address (int):
1-30 (0 and 31 reserved).

get_ieee_488()

Returns the IEEE address set.

Returns:

address (int):
1-30 (0 and 31 reserved).

set_led_state(*state*)

Sets the front panel LEDs to on or off.

Args:

state (bool):
Sets the LEDs to functional or nonfunctional. Options are: False for off, True for on.

get_led_state()

Returns whether front panel LEDs are enabled.

Returns:

state (bool):
Specifies whether front panel LEDs are functional. Returns: False if disabled, True enabled.

set_keypad_lock(*state*, *code*)

Locks or unlocks front panel keypad (except for alarms and disabling heaters).

Args:

state (bool):
Sets the keypad to locked or unlocked. Options are: False for unlocked or True for locked.

code (int):
Specifies 3 digit lock-out code. Options are: 000 - 999.

get_keypad_lock()

Returns the state of the keypad lock and the lock-out code.

Returns:

(dict):
{“state”: bool, “code”: int}

get_min_max_data(*input_channel*)

Returns the minimum and maximum data from an input.

Args:

input_channel (str):
Specifies which input to query.

Returns:

min_max_data (dict):
{“minimum”: float, “maximum”: float}

reset_min_max_data()

Resets the minimum and maximum input data.

set_input_curve(*input_channel*, *curve_number*)

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str):

Specifies which input to configure.

curve_number (int):

0 = none, 1-20 = standard curves, 21-59 = user curves.

get_input_curve(*input_channel*)

Returns the curve number being used for a given input.

Args:

input_channel (str):

Specifies which input to query.

Returns:

curve_number (int):

0-59.

set_website_login(*username*, *password*)

Sets the username and password to connect instrument to website.

Args:

username (str):

Username to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

password (str):

Password to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

get_website_login()

Returns the set username and password for web login for the instrument.

Returns:

(dict):

{“username”: str, “password”: str}

set_alarm_parameters(*input_channel*, *alarm_enable*, *alarm_settings=None*)

Configures the alarm parameters for an input.

Args:

input_channel (str):

Specifies which input to configure.

alarm_enable (bool):

Specifies whether to turn on the alarm for the input, or turn the alarm off.

alarm_settings (Model224AlarmParameters):

See Model224AlarmParameters class. Optional if alarm_enable is set to False.

get_alarm_parameters(*input_channel*)

Returns the present state of all alarm parameters.

Args:

input_channel (str):
Specifies which input to configure.

Returns:

(dict):
{“alarm_enable”: bool, “alarm_settings”: Model224AlarmParameters}.

get_alarm_status(*input_channel*)

Returns the high state and low state of the alarm for the specified channel.

Args:

input_channel (str):
Specifies which input channel to read from. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(dict):
{“high_state”: bool, “low_state”: bool}

reset_alarm_status()

Clears the high and low status of all alarms.

set_curve_header(*curve_number*, *curve_header*)

Configures the user curve header.

Args:

curve_number (int):
Specifies which curve to configure. Options are: 21 - 59.

curve_header (Model224CurveHeader):
A Model224CurveHeader class object containing the desired curve information.

get_curve_header(*curve*)

Returns parameters set on a particular user curve header.

Args:

curve (int):
Specifies a curve to retrieve. Options are: 21 - 59.

Returns:

header (Model224CurveHeader):
A Model224CurveHeader class object containing the desired curve information.

set_curve_data_point(*curve*, *index*, *sensor_units*, *temperature*)

Configures a user curve point.

Args:

curve (int or str):
Specifies which curve to configure.

index (int):
Specifies the points index in the curve.

sensor_units (float):

Specifies sensor units for this point to 6 digits.

temperature (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

get_curve_data_point(*curve, index*)

Returns a standard or user curve data point.

Args:

curve (int):

Specifies which curve to query.

index (int):

Specifies the points index in the curve.

Returns:

curve_point (tuple):

(sensor_units: float, temp_value: float).

delete_curve(*curve*)

Deletes the user curve.

Args:

curve (int):

Specifies a user curve to delete.

generate_and_apply_soft_cal_curve(*source_curve, curve_number, serial_number, calibration_point_1, calibration_point_2=(0, 0), calibration_point_3=(0, 0)*)

Creates a SoftCal curve from 1-3 temperature/sensor points and a standard curve.

Inputs generated curve into the given curve number.

Args:

source_curve (Model224SoftCalSensorTypes):

The standard curve to use to generate the SoftCal curve from along with calibration points.

curve_number (int):

The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):

Serial number of the user curve. Maximum of 10 characters.

calibration_point_1 (tuple):

Tuple of two floats in the form (temperature_value, sensor_value).

calibration_point_2 (tuple):

Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple):

Tuple of two floats in the form (temperature_value, sensor_value). Optional parameter.

get_curve(*curve*)

Returns a list of all the data points in a particular curve.

Args:

curve (int):

Specifies which curve to set.

Return:**data_points (list):**

A list containing every point in the curve represented as a tuple (sensor_units: float, temp_value: float).

set_curve(*curve*, *data_points*)

Method to define a user curve using a list of data points.

Args:**curve (int):**

Specifies which curve to set.

data_points (list):

A list containing every point in the curve represented as a tuple
(sensor_units: float, temp_value: float).

get_relay_status(*relay_channel*)

Returns whether the specified relay is On or Off.

Args:**relay_channel (int):**

The relay channel to query. Options are: 1 or 2.

Returns:**(bool):**

True if relay is on, False if relay is off.

set_filter(*input_channel*, *filter_enabled*, *number_of_points*=8, *filter_reset_threshold*=10)

Enables or disables a filter for the readings of the specified input channel.

Filter is a running average that smooths input readings exponentially.

Args:**input_channel (str):**

The input to set or disable a filter for. Options are: A, B, C(1 - 5), D(1 - 5).

filter_enabled (bool):

Enables or disables a filter for the input channel. True for enabled, False for disabled.

number_of_points (int):

Specifies the number of points used for the filter. Inputting a larger number of points will slow down the instrument's response to changes in temperature. Options are: 2 - 64 Optional if disabling the filter function.

filter_reset_threshold (int):

Specifies the limit for restarting the filter, represented by a percent of the full scale reading. If raw reading differs from filtered value by more than this threshold, filter averaging resets. Options are: 1% - 10%. Optional if disabling the filter function.

get_filter(*input_channel*)

Retrieves information about the filter set on the specified input channel.

Args:**input_channel (str):**

The input to query for filter information. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(dict):

{“filter_enabled”: bool, “number_of_points”: int, “filter_reset_threshold”: int}

configure_input(*input_channel*, *settings*)

Configures a sensor for measurement input readings.

Args:

input_channel (str):

The input to configure the input for. Options are: A, B, C(1 - 5), D(1 - 5).

settings (Model224InputSensorSettings):

Object of the Model224InputSensorSettings containing information for sensor setup.

disable_input(*input_channel*)

Disables the selected input channel.

Args:

input_channel (str):

The input to disable. Options are: A, B, C(1 - 5), D(1 - 5).

get_input_configuration(*input_channel*)

Returns the configuration settings of the sensor at the specified input channel.

Args:

input_channel (str):

The input to query. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(Model224InputSensorSettings):

Object of type Model224InputSensorSettings containing information about the sensor at the given input_channel.

select_remote_interface(*remote_interface*)

Selects the remote interface to use for communications.

Args:

remote_interface (Model224RemoteInterface):

Object of enum type Model224RemoteInterface, representing the type of interface used for communications.

get_remote_interface()

Returns the remote interface being used for communications.

Returns:

(Model224RemoteInterface):

Object of enum type Model224RemoteInterface representing the interface being used for communications.

select_interface_mode(*interface_mode*)

Selects the mode for the remote interface being used.

Args:

interface_mode (Model224InterfaceMode):

Object of enum type Model224InterfaceMode representing the desired communication mode.

get_interface_mode()

Returns the mode of the remote interface.

Returns:**(Model224InterfaceMode):**

Object of enum type Model224InterfaceMode representing the communication mode.

set_display_field_settings(*field*, *input_channel*, *display_units*)

Configures a display field in custom display mode.

Args:**field (int):**

Specifies which display field to configure. Options are: 1 - 8.

input_channel (Model224InputChannel):

Defines which input to display.

display_units (Model224DisplayFieldUnits):

Defines which units to display reading in.

get_display_field_settings(*field*)

Returns the settings of a single display field in custom display mode.

Args:**field (int):**

Specifies the display field to query. Options are: 1 - 8.

Returns:**(dict):**

{“input_channel”: Model224InputChannel, “display_units”: Model224DisplayFieldUnits}

configure_display(*display_mode*, *number_of_fields=0*)

Configures the display of the instrument.

Args:**display_mode (Model224DisplayMode):**

Defines what mode to set the display in. Mode either defines which input to display, or sets up a custom display using display fields.

number_of_fields (Model224NumberOfFields):

Defines the number of display locations to display. Only valid if mode is set to CUSTOM.

get_display_configuration()

Returns the mode of the display.

If display mode is Custom, this method also returns the number of display fields in the custom display.

Returns:**(dict):**

{“display_mode”: DisplayMode, “number_of_fields”: NumberOfFields}.

turn_relay_on(*relay_number*)

Turns the specified relay on.

Args:**relay_number (int):**

The relay to turn on. Options are: 1 or 2.

turn_relay_off(*relay_number*)

Turns the specified relay off.

Args:

relay_number (int):

The relay to turn off. Options are: 1 or 2.

set_relay_alarms(*relay_number, activating_input_channel, alarm_relay_trigger_type*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:

relay_number (int):

The relay to configure. Options are: 1 or 2.

activating_input_channel (str):

Specifies which input alarm activates the relay when the relay is in alarm mode. Only applies if ALARM mode is chosen. Options are: A, B, C(1 - 5), D(1 - 5).

alarm_relay_trigger_type (Model224RelayControlAlarm):

Specifies the type of alarm that triggers the relay. Only applies if ALARM mode is chosen.

get_relay_alarm_control_parameters(*relay_number*)

Returns the relay alarm configuration for either of the two configurable relays.

Relay must be configured for alarm mode to retrieve parameters.

Args:

relay_number (int):

Specifies which relay to query. Options are: 1 or 2.

Return:

(dict):

{“activating_input_channel”: str, “alarm_relay_trigger_type”: Model224RelayControlAlarm}.

get_relay_control_mode(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):

Specifies which relay to query. Options are: 1 or 2.

Returns:

(Model224RelayControlMode):

The configured mode of the relay, represented as an object of the enum type Model224RelayControlMode.

connect_tcp(*ip_address, tcp_port, timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

write(command_string)

Alias of `command`. Send a command to the instrument.

Args:**command_string (str):**

A serial command.

Settings classes

```
class lakeshore.model_224.Model224AlarmParameters(high_value, low_value, deadband, latch_enable,
audible=None, visible=None)
```

Class used to disable or configure an alarm in conjunction with the `set/get_alarm_parameters()` method.

Special-members

`__init__`

```
class lakeshore.model_224.Model224InputSensorSettings(sensor_type, preferred_units,
sensor_range=None,
autorange_enabled=False,
compensation=False)
```

Class representing the parameters of a sensor in one of the instrument's inputs.

Special-members

`__init__`

```
class lakeshore.model_224.Model224CurveHeader(curve_name, serial_number, curve_data_format,
temperature_limit, coefficient)
```

A class that configures the user curve header and corresponding parameters.

Special-members

`__init__`

Enumeration objects

This section describes the Enum type objects that have been created to name various settings of the Model 224 series that are represented as an int or single character to the instrument. The purpose of these enum types is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

```
class lakeshore.model_224.Model224Enums
```

Class containing the enums relevant to the Model 224.

```
class InputSensorType(*values)
```

Enumeration for the type of sensor being used for a given input.

```
INPUT_DISABLED = 0
```

```
DIODE = 1
```

```
PLATINUM_RTD = 2
```

```
NTC_RTD = 3
```

```
class InputSensorUnits(*values)
```

Enumeration for the preferred units of an input sensor.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

class DiodeSensorRange(*values)

Enumeration for the voltage range of a diode sensor.

RANGE_2_POINT_5_VOLTS = 0

RANGE_10_VOLTS = 1

class PlatinumRTDSensorResistanceRange(*values)

Enumeration of the resistance range of a platinum RTD input sensor.

TEN_OHMS = 0

THIRTY_OHMS = 1

ONE_HUNDRED_OHMS = 2

THREE_HUNDRED_OHMS = 3

ONE_KILOHM = 4

THREE_KILOHMS = 5

TEN_KILOHMS = 6

class NTCRTDSensorResistanceRange(*values)

Enumeration of the resistance range of a NTC RTD input sensor.

TEN_OHMS = 0

THIRTY_OHMS = 1

ONE_HUNDRED_OHMS = 2

THREE_HUNDRED_OHMS = 3

ONE_KILOHM = 4

THREE_KILOHMS = 5

TEN_KILOHMS = 6

THIRTY_KILOHMS = 7

ONE_HUNDRED_KILOHMS = 8

class InterfaceMode(*values)

Enumeration for the mode of the remote interface.

LOCAL = 0

REMOTE = 1

REMOTE_LOCAL_LOCK = 2

```
class RemoteInterface(*values)
```

Enumeration for the remote interface being used to communicate with the instrument.

```
USB = 0
```

```
ETHERNET = 1
```

```
IEEE_488 = 2
```

```
class DisplayFieldUnits(*values)
```

Enumerated type defining how units are enumerated for settings and using Display Fields.

```
KELVIN = 1
```

```
CELSIUS = 2
```

```
SENSOR = 3
```

```
MINIMUM_DATA = 4
```

```
MAXIMUM_DATA = 5
```

```
class InputChannel(*values)
```

Enumerated type defining which input channels correspond to ints for setting and using Display Fields.

```
NO_INPUT = 0
```

```
INPUT_A = 1
```

```
INPUT_B = 2
```

```
INPUT_C = 3
```

```
INPUT_D1 = 4
```

```
INPUT_D2 = 5
```

```
INPUT_D3 = 6
```

```
INPUT_D4 = 7
```

```
INPUT_D5 = 8
```

```
INPUT_C2 = 9
```

```
INPUT_C3 = 10
```

```
INPUT_C4 = 11
```

```
INPUT_C5 = 12
```

```
class DisplayMode(*values)
```

Enumeration defining what input or information is shown on the front panel display.

```
INPUT_A = 0
```

```
INPUT_B = 1
```

```
INPUT_C = 2
```

```
INPUT_D1 = 3
CUSTOM = 4
ALL_INPUTS = 5
INPUT_D2 = 6
INPUT_D3 = 7
INPUT_D4 = 8
INPUT_D5 = 9
INPUT_C2 = 10
INPUT_C3 = 11
INPUT_C4 = 12
INPUT_C5 = 13
```

class NumberOfFields(*values)

Enumerated type specifying the number of display fields to configure in the Custom display mode.

```
LARGE_4 = 0
LARGE_8 = 1
LARGE_4_SMALL_8 = 2
SMALL_16 = 3
```

class RelayControlAlarm(*values)

Enumeration of the setting determining which alarm(s) cause a relay to activate in alarm mode.

```
LOW_ALARM = 0
HIGH_ALARM = 1
BOTH_ALARMS = 2
```

class RelayControlMode(*values)

Enumeration of the configured mode of a relay.

```
RELAY_OFF = 0
RELAY_ON = 1
ALARMS = 2
```

class CurveFormat(*values)

Enumerations specify formats for temperature sensor curves.

```
MILLIVOLT_PER_KELVIN = 1
VOLTS_PER_KELVIN = 2
OHMS_PER_KELVIN = 3
```

```
LOG_OHMS_PER_KELVIN = 4
```

```
class CurveTemperatureCoefficients(*values)
```

Enumerations specify positive/negative temperature sensor curve coefficients.

```
NEGATIVE = 1
```

```
POSITIVE = 2
```

```
class DiodeExcitationCurrent(*values)
```

Enum type representing the different excitation currents available for a diode sensor.

```
TEN_MICRO_AMPS = 0
```

```
ONE_MILLI_AMP = 1
```

```
class SoftCalSensorTypes(*values)
```

Enum type representing the standard curves used to generate a SoftCal curve.

The 3 standard curves each represent a different type of sensor that can be calibrated with a SoftCal curve.

```
DT_400 = 1
```

```
PT_100 = 6
```

```
PT_1000 = 7
```

Status register classes

```
lakeshore.model_224.Model224StandardEventRegister
```

alias of *StandardEventRegister*

```
class lakeshore.temperature_controllers.StandardEventRegister(operation_complete, query_error,
                                                             execution_error, command_error,
                                                             power_on)
```

Class object representing the standard event register.

```
bit_names = ['operation_complete', '', 'query_error', '', 'execution_error',
             'command_error', '', 'power_on']
```

```
class lakeshore.model_224.Model224ServiceRequestRegister(message_available, event_summary,
                                                         operation_summary)
```

Class object representing the Service Request Enable register.

```
bit_names = ['', '', '', '', 'message_available', 'event_summary',
             'operation_summary']
```

```
class lakeshore.model_224.Model224StatusByteRegister(message_available, event_summary,
                                                      master_summary_status, operation_summary)
```

Class object representing the status byte register.

```
bit_names = ['', '', '', '', 'message_available', 'event_summary',
             'master_summary_status', 'operation_summary']
```

```
class lakeshore.model_224.Model224ReadingStatusRegister(invalid_reading, temperature_under_range,  
temperature_over_range,  
sensor_units_zero,  
sensor_units_over_range)
```

Class object representing the reading status of an input.

While not a literal register, the return of an int representation of multiple booleans makes it convenient to represent this functionality as a register.

```
bit_names = ['invalid_reading', '', '', '', 'temperature_under_range',  
'temperature_over_range', 'sensor_units_zero', 'sensor_units_over_range']
```

Model 240 Input Modules

The 240 Series Input Modules employ distributed PLC control for large scale cryogenic temperature monitoring.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example Scripts

Model 240 Input Channel Setup Example

```
from lakeshore import Model240, Model240InputParameter  
from time import sleep  
  
# Connect to the first available Model 240 over USB  
my_model_240 = Model240()  
  
# Define the channel configuration for a sensor with a negative temperature coefficient,  
↪ autorange disabled  
# current reversal disabled, the channel enabled, and set to the 100 kOhm range  
rtd_config = Model240InputParameter(my_model_240.SensorTypes.NTC_RTDT, False, False, my_  
↪ model_240.Units.SENSOR, True,  
                                     my_model_240.InputRange.RANGE_NTCRTD_100_KIL_OHMS)  
  
# Apply the configuration to all channels  
for channel in range(1, 9):  
    my_model_240.set_input_parameter(channel, rtd_config)  
  
sleep(1)  
print("Reading from channel 5: {} ohms".format(my_model_240.get_sensor_reading(5)))
```

Model 240 Profibus Configuration Example

```
from lakeshore import Model240, Model240ProfiSlot  
  
# Connect to the first available Model 240 over USB  
my_model_240 = Model240()  
  
# Print the instrument's current PROFIBUS connection status to the console  
print("Profibus connection status: " + my_model_240.get_profibus_connection_status())
```

(continues on next page)

(continued from previous page)

```

# Configure the number of PROFIBUS slots for the instrument to present to the bus as a
↳modular station
# Setting the number of PROFIBUS slots to 2
my_model_240.set_profibus_slot_count(2)

# Create the ProfiSlot class object by specifying which input to associate the
# slot with and what temperature units the data will be presented in
# Setting the input channel as 2 and temperature units to Celsius
my_profibus_slot = Model240ProfiSlot(2, my_model_240.Units.CELSIUS)

# Configure what data to be presented on the given PROFIBUS slot
# Profibus slot 1 will be associated with channel 2
my_model_240.set_profibus_slot_configuration(1, my_profibus_slot)

# Print the PROFIBUS address
# An address of 126 indicates that it is not configured and it can then be set by a
↳PROFIBUS master
print("Profibus address: " + my_model_240.get_profibus_address())

# Set the desired address as 123
my_model_240.set_profibus_address("123")

# Acquiring settings that were configured above
print(my_model_240.get_profibus_slot_count())
slot_1_config = my_model_240.get_profibus_slot_configuration(1)
print(slot_1_config.slot_channel)
print(slot_1_config.slot_units)

```

Instrument class methods

class lakeshore.model_240.**Model240**(*serial_number=None, com_port=None, timeout=2.0, **kwargs*)

A class object representing the Lake Shore Model 240 channel modules.

get_identification()

Returns instrument's identification parameters.

Returns:

id (list):

List defining instrument's manufacturer, model, instrument serial, firmware version.

set_brightness(brightness_level)

Sets the brightness for the front panel display.

Args:

brightness_level (Model240BrightnessLevel):

Display brightness in percent.

get_brightness()

Returns the brightness level of front panel display.

Return:

brightness_level (Model240BrightnessLevel):

Display brightness in percent.

get_celsius_reading(*channel*)

Returns the temperature value in Celsius of channel selected.

Args:

channel (int):

Specifies channel (1-8).

set_factory_defaults()

Sets all configuration values to factory defaults and resets the instrument.

get_kelvin_reading(*channel*)

Returns the temperature value in Kelvin of channel selected.

Args:

channel (int):

Specifies channel (1-8).

get_fahrenheit_reading(*channel*)

Returns the temperature value in Fahrenheit of channel selected.

Args:

channel (int):

Specifies channel (1-8).

get_sensor_reading(*input_channel*)

Returns the sensor reading in the sensor's units.

Returns:

reading (float):

The raw sensor reading in the units of the connected sensor.

delete_curve(*curve*)

Deletes the user curve.

Args:

curve (int):

Specifies a user curve to delete.

set_curve_header(*input_channel*, *curve_header*)

Configures the user curve header.

Args:

input_channel (int):

Specifies which input_channel curve to configure (1 - 8).

curve_header (CurveHeader):

A CurveHeader class object containing the desired curve information.

get_curve_header(*curve*)

Returns parameters set on a particular user curve header.

Args:

curve:

Specifies a curve to retrieve.

Returns:

header (CurveHeader):

A CurveHeader class object containing the desired curve information.

set_curve_data_point(*channel, index, units, temp*)

Configures a user curve point.

Args:**channel (int):**

Specifies which channel curve to configure (1-8).

index (int):

Specifies the points index in the curve (1-200).

units (float):

Specifies sensor units for this point to 6 digits.

temp (float):

Specifies the corresponding temperature in Kelvin for this point to 6 digits.

get_curve_data_point(*channel, index*)

Returns a standard or user curve data point.

Args:**channel (int):**

Specifies channel (1-8).

index (int):

Specifies the points index in the curve (1-200).

set_filter(*channel, length*)

Sets the channel filter parameter.

Args:**channel (int):**

Specifies which channel to configure (1-8).

length (int):

Specifies the number of 1 ms points to average for each update (1-100).

get_filter(*channel*)

Returns the filter parameter.

Args:**channel (int):**

Specifies channel (1-8).

set_sensor_name(*channel, name*)

Names the sensor channel in specified channel.

Args:**channel (int):**

Specifies which channel to configure (1-8).

name (str):

Specifies the name to associate with the sensor channel.

get_sensor_name(*channel*)

Returns the sensor channel's name.

Args:

channel (int):

Specifies channel (1-8).

set_input_parameter(*channel, input_parameter*)

Sets channel type parameters.

Args:

channel (int):

Specifies which channel to configure (1-8).

input_parameter (InputParameter):

See InputParameter class.

get_input_parameter(*channel*)

Returns channel type parameter details.

Args:

channel (int):

Specifies channel (1-8).

set_modname(*name*)

Names module.

Args:

name (str):

Specifies the name or description to help identify the module.

get_modname()

Returns module name.

Returns:

modname (str):

Specifies name of module.

set_profibus_slot_count(*count*)

Configures the number of PROFIBUS slots for the instrument to present to the bus as a modular station.

Args:

count (int):

Specifies the number of PROFIBUS slots (1-8).

get_profibus_slot_count()

Returns the number of PROFIBUS slots for the instrument present to the bus as a modular station.

Returns:

slot_count (str):

Specifies PROFIBUS slot count.

set_profibus_address(*address*)

Configures the PROFIBUS address for the module.

An address of 126 indicates that it is not configured, and it then can be set by a PROFIBUS master.

Args:

address (str):
Specifies the PROFIBUS address (1-126).

get_profibus_address()

Returns the PROFIBUS address for the module.

Returns:

address (str):
Specifies PROFIBUS address of module.

set_profibus_slot_configuration(slot, profislot_config)

Configures what data to present on the given PROFIBUS slot.

Note that the correct number of slots must be configured with the PROFINUM command, or the slot may be ignored.

Args:

slot (int):
Specifies the slot to be configured.

profislot_config (Model240ProfiSlot):
A Model240ProfiSlot class object containing the desired PROFIBUS slot configuration information.

get_profibus_slot_configuration(slot_num)

Returns the slot configuration of the slot number.

Returns:

slot_config (Model240ProfiSlot):
See Model240ProfiSlot class.

get_profibus_connection_status()

Returns the connection status of PROFIBUS.

Returns:

status (str):
Specifies connection status of PROFIBUS.

get_channel_reading_status(channel)

Returns the current status indicator of the specified channel

The integer returned represents the sum of the bit weighting of the channel status flag bits. A “000” response indicates a valid reading is present.

Args:

channel (int):
Specifies which channel to query (1-8).

Returns:

bit_status (dict):
Dictionary containing the current status indicator.

get_sensor_units_channel_reading(channel)

Returns the sensor units value of channel being queried.

Args:

channel (int):

Specifies which channel to query (1-8).

command(*command_string*)

Send a command to the instrument.

Args:

command_string (str):

A serial command.

connect_tcp(*ip_address, tcp_port, timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

query(*query_string*)

Send a query to the instrument and return the response.

Args:

query_string (str):

A serial query ending in a question mark.

Returns:

The instrument query response as a string.

write(*command_string*)

Alias of `command`. Send a command to the instrument.

Args:

command_string (str):

A serial command.

Settings classes

This page describes the classes used throughout the 240 methods that interact with instrument settings and other methods that use objects and classes.

class `lakeshore.model_240.Model240CurveHeader`(*curve_name, serial_number, curve_data_format, temperature_limit, coefficient*)

A class that configures the user curve header and corresponding parameters.

Special-members

`__init__`

class `lakeshore.model_240.Model240InputParameter`(*sensor, auto_range_enable, current_reversal_enable, units, input_enable, input_range=None*)

Class used to retrieve and set an input channel's parameters and initial settings.

Special-members`__init__`**class** lakeshore.model_240.**Model240ProfiSlot**(*channel, temp_unit*)

Class used to configure and retrieve data for given PROFIBUS slot.

Special-members`__init__`**Enumeration objects**

This section describes the Enum type objects that have been created to name various settings of the Model 240 series that are represented as an int or single character to the instrument. The purpose of these enum types is to make the settings more descriptive and obvious to the user rather than interpreting the ints taken by the instrument.

class lakeshore.model_240.**Model240Enums**

Class containing the enums relevant to the Model 240.

class **Units**(**values*)

Enumerations that specify temperature units.

KELVIN = 1**CELSIUS = 2****SENSOR = 3****FAHRENHEIT = 4****class** **CurveFormat**(**values*)

Enumerations that specify temperature sensor curve format units.

VOLTS_PER_KELVIN = 2**OHMS_PER_KELVIN = 3****LOG_OHMS_PER_KELVIN = 4****class** **Coefficients**(**values*)

Enumerations that specify a positive or negative coefficient.

NEGATIVE = 1**POSITIVE = 2****class** **SensorTypes**(**values*)

Enumerations specify types of temperature sensors.

DIODE = 1**PLATINUM_RTD = 2****NTC_RTD = 3****class** **BrightnessLevel**(**values*)

Enumerations for the screen brightness levels.

OFF = 0

LOW = 25

MED_LOW = 50

MED_HIGH = 75

HIGH = 100

class TemperatureCoefficient(*values)

Enumerations specify positive/negative temperature sensor curve coefficients.

NEGATIVE = 1

POSITIVE = 2

class InputRange(*values)

Enumerations to specify the input range when auto-range is off.

RANGE_DIODE = 0

RANGE_PTRTD_1_KIL_OHMS = 0

RANGE_NTCRTD_10_OHMS = 0

RANGE_NTCRTD_30_OHMS = 1

RANGE_NTCRTD_100_OHMS = 2

RANGE_NTCRTD_1_KIL_OHMS = 4

RANGE_NTCRTD_3_KIL_OHMS = 5

RANGE_NTCRTD_10_KIL_OHMS = 6

RANGE_NTCRTD_30_KIL_OHMS = 7

RANGE_NTCRTD_100_KIL_OHMS = 8

2.4.6 Sources

Model 121 Programmable DC Current Source

The Lake Shore Model 121 provides low-noise, stable current.

More information about the instrument can be found [on our website](#) including the manual which has a list of all commands and queries.

Example Scripts

Setting a current

```
from lakeshore import Model121

# Connect to Model 121 over USB
current_source = Model121()

# Set current source to 150 nA and start
current_source.set_current(150e-9)
```

(continues on next page)

(continued from previous page)

```
# Query and print the current source level
print(current_source.get_current())
```

Instrument methods

```
class lakeshore.model_121.Model121(serial_number=None, com_port=None, baud_rate=57600,
                                   data_bits=7, stop_bits=1, parity='O', flow_control=False,
                                   handshaking=False, timeout=2.0, ip_address=None, tcp_port=7777,
                                   **kwargs)
```

A class object representing the Lake Shore Model 121 programmable DC current source.

command(*command_string*)

Sends a command to the instrument.

Args:

command_string (str): A serial command.

set_current(*current*)

Set and start outputting a specific current from the instrument.

Switches to the user current range and applies the specified current. Current can be set between 100nA and 100mA. The sign of the current determines the polarity of the output.

Args:

current (float): New current value as a floating point number. The minimum value is 100e-9 A and the maximum
is 100e-3 A. Positive or negative. Up to three significant digits.

get_current()

Returns the present user current setting in Amps.

Returns:

float: User current value in Amps.

enable_current()

Enables current source output.

disable_current()

Disables current source output.

reset_instrument()

Sets instrument parameters to power-up settings.

set_display_brightness(*brightness_level*)

Sets the display contrast for the front panel seven-segment display.

A higher number makes the display brighter. The default setting is 8. The display can be turned off by setting the brightness to 0.

Args:

brightness_level (int): The display brightness (0-15).

get_display_brightness()

Gets the display contrast for the front panel seven-segment display.

A higher number makes the display brighter. The default setting is 8. Brightness level 0 means the display is turned off.

Returns:

int: The display brightness (0-15).

get_compliance_limit_status()

Returns the voltage compliance status of the current source output.

Returns:

bool: False = normal operation. True = in compliance limit.

set_factory_defaults()

Sets all configuration values to factory defaults and resets the instrument.

lock_front_panel()

Locks the front panel keypad.

unlock_front_panel()

Unlocks the front panel keypad.

get_front_panel_lock_status()

Returns if the front panel keypad is locked or not.

Returns:

bool: True = Locked, False = Unlocked.

set_power_up_enable(*state*)

Specifies whether the output remains on or shuts off after power cycle.

Args:

state (bool): True = Enabled, False = Disabled.

save_current_state()

Saves the present range, polarity, and user current value.

This saved state will be loaded on future power ups.

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(*serial_number=None*, *com_port=None*, *baud_rate=None*, *data_bits=None*, *stop_bits=None*, *parity=None*, *timeout=None*, *handshaking=None*, *flow_control=None*)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

query(*query_string*)

Send a query to the instrument and return the response.

Args:

query_string (str):

A serial query ending in a question mark.

Returns:

The instrument query response as a string.

write(command_string)

Alias of command. Send a command to the instrument.

Args:**command_string (str):**

A serial command.

Model 155 Precision Current and Voltage Source

The Lake Shore 155 is a low noise, high precision current and voltage source.

More information about the instrument can be found [on our website](#) including the manual which has a list of all SCPI commands and queries.

Example Scripts

Below are a few example scripts for the Model 155 that use the Lake Shore Python driver.

Model 155 Sweep Example

```

from lakeshore import PrecisionSource

# The purpose of this script is to sweep frequency, amplitude, and offset of an output
↳ signal using
# a Lake Shore AC/DC 155 Precision Source

# Create a new instance of the Lake Shore 155 Precision Source.
# It will connect to the first instrument it finds via serial USB
my_source = PrecisionSource()

# Define a custom list of frequencies to sweep through
frequency_sweep_list = ['1', '10', '100', '250', '500', '750', '1000', '2000', '5000',
↳ '10000']

# Sweep frequency in voltage mode. Wait 1 second at each step
my_source.sweep_voltage(1, frequency_values=frequency_sweep_list)

# Creates a list of whole number offset values between -5V and 5V.
offset_sweep_list = range(-5, 6)
# Creates a list of amplitudes between 0 and 5V incrementing by 100mV
amplitude_sweep_list = [value/10 for value in range(0, 51)]
# Creates a list of frequencies starting with 0.1 Hz and increasing by powers of ten up
↳ to 10 kHz
frequency_sweep_list = [10**exponent for exponent in range(-1, 5)]

# Use the lists defined above to sweep across all combinations of the lists.
# For each combination, wait 10ms before moving to the next one.

```

(continues on next page)

(continued from previous page)

```
# Note that the dwell time will be limited by the response time of the serial_
↳communication.
my_source.sweep_voltage(0.01,
                        offset_values=offset_sweep_list,
                        amplitude_values=amplitude_sweep_list,
                        frequency_values=frequency_sweep_list)
```

Instrument class methods

```
class lakeshore.model_155.PrecisionSource(serial_number=None, com_port=None, baud_rate=115200,
                                         flow_control=False, timeout=2.0, ip_address=None,
                                         tcp_port=7777, **kwargs)
```

A class object representing a Lake Shore 155 precision I/V source.

```
sweep_voltage(dwell_time, offset_values=None, amplitude_values=None, frequency_values=None)
```

Sweep source output voltage parameters based on list arguments.

Args:

dwell_time (float):

The length of time in seconds to wait at each parameter combination. Note that the update rate will be limited by the SCPI communication response time. The response time is usually on the order of 10-30 milliseconds.

offset_values (list):

DC offset values in volts to sweep over.

amplitude_values (list):

Peak to peak values in volts to sweep over.

frequency_values (list):

Frequency values in Hertz to sweep over.

```
sweep_current(dwell_time, offset_values=None, amplitude_values=None, frequency_values=None)
```

Sweep the source output current parameters based on list arguments

Args:

dwell_time (float):

The length of time in seconds to wait at each parameter combination. Note that the update rate will be limited by the SCPI communication response time. The response time is usually on the order of 10-30 milliseconds.

offset_values (list):

DC offset values in volts to sweep over.

amplitude_values (list):

Peak to peak values in volts to sweep over.

frequency_values (list):

Frequency values in Hertz to sweep over.

```
enable_output()
```

Turns on the source output.

```
disable_output()
```

Turns off the source output.

set_output(*output_on*)

Configure the source output on or off.

Args:**output_on (bool):**

Turns the source output on when True, off when False.

route_terminals(*output_connections_location='REAR'*)

Configures whether the source output is routed through the front or rear connections.

Args:**output_connections_location (str):**

Valid options are: “REAR” (Output is routed out the rear connections), and “FRONT” (Output is routed out the front connections).

output_sine_current(*amplitude, frequency, offset=0.0, phase=0.0*)

Configures and enables the source output to be a sine wave current source.

Args:**amplitude (float):**

The peak current amplitude value in amps.

frequency (float):

The source frequency value in hertz.

offset (float):

The DC offset current in amps.

phase (float):

Shifts the phase of the output relative to the reference out. Must be between -180 and 180 degrees.

output_sine_voltage(*amplitude, frequency, offset=0.0, phase=0.0*)

Configures and enables the source output to be a sine wave voltage source.

Args:**amplitude (float):**

The peak voltage amplitude value in volts.

frequency (float):

The source frequency value in hertz.

offset (float):

The DC offset voltage in volts.

phase (float):

Shifts the phase of the output relative to the reference out. Must be between -180 and 180 degrees.

output_dc_current(*current_level*)

Configures the source output to be a DC current source.

Args:**current_level (float):**

The output current level in amps.

output_dc_voltage(*voltage_level*)

Configures the source output to be a DC current source.

Args:

voltage_level (float):

The output voltage level in volts.

get_output_settings()

Returns a dictionary of the output settings.

enable_autorange()

Enables the instrument to automatically select the best range for the given output parameters.

disable_autorange()

Enables the instrument to automatically select the best range for the given output parameters.

set_current_range(*current_range='100E-3'*)

Manually sets the current range when autorange is disabled.

Args:

current_range (str):

The range in amps. Valid ranges are: "100E-3", "10E-3", "1E-3", "100E-6", "10E-6", and "1E-6".

set_voltage_range(*voltage_range='10'*)

Manually sets the voltage range when autorange is disabled.

Args:

voltage_range (str):

The range in volts. Valid ranges are: "100", "10", "1", "0.1", and "0.01".

set_current_limit(*current_limit*)

Sets the highest settable current output value when in current mode.

Args:

current_limit (float):

The maximum settable current in amps. Must be between 0 and 100 milli-amps.

set_voltage_limit(*voltage_limit*)

Sets the highest settable voltage output value when in voltage mode.

Args:

voltage_limit (float):

The maximum settable voltage in amps. Must be between 0 and 100 volts.

set_current_mode_voltage_protection(*max_voltage*)

Sets the maximum voltage level permitted by the instrument when sourcing current.

Args:

max_voltage (float):

The maximum permissible voltage. Must be between 1 and 100 volts.

set_voltage_mode_current_protection(*max_current*)

Sets the maximum current level permitted by the instrument when sourcing voltage.

Args:

max_current (float):

The maximum permissible voltage. Must be between 1 and 100 volts.

enable_ac_high_voltage_compliance()

Configures the current mode compliance voltage to be 100V in AC output modes.

disable_ac_high_voltage_compliance()

Configures the current mode compliance voltage to be 10V in AC output modes.

command(*commands, check_errors=True)

Send an SCPI command or multiple commands to the instrument.

Args:**commands (str):**

Any number of SCPI commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

connect_tcp(ip_address, tcp_port, timeout)

Establishes a TCP connection with the instrument on the specified IP address.

connect_usb(serial_number=None, com_port=None, baud_rate=None, data_bits=None, stop_bits=None, parity=None, timeout=None, handshaking=None, flow_control=None)

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

Resets all system information such as settings, wi-fi connections, date and time, etc.

get_operation_event_enable_mask()

Returns the names of the operation event enable register bits and their values.

These values determine which operation bits propagate to the operation event register.

get_operation_events()

Returns the names of operation event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_present_operation_status()

Returns the names of the operation status register bits and their values.

get_present_questionable_status()

Returns the names of the questionable status register bits and their values.

get_questionable_event_enable_mask()

Returns the names of the questionable event enable register bits and their values.

These values determine which questionable bits propagate to the questionable event register.

get_questionable_events()

Returns the names of questionable event status register bits that are currently high.

The event register is latching and values are reset when queried.

get_service_request_enable_mask()

Returns the named bits of the status byte service request enable register.

This register determines which bits propagate to the master summary status bit.

get_standard_event_enable_mask()

Returns the names of the standard event enable register bits and their values.

These values determine which bits propagate to the standard event register.

get_standard_events()

Returns the names of the standard event register bits and their values.

get_status_byte()

Returns named bits of the status byte register and their values.

modify_operation_register_mask(*bit_name, value*)

Gets the operation condition register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_questionable_register_mask(*bit_name, value*)

Gets the questionable condition register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_service_request_mask(*bit_name, value*)

Gets the service request enable mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

modify_standard_event_register_mask(*bit_name, value*)

Gets the standard event register mask, changes a bit, and sets the register.

Args:

bit_name (str):

The name of the bit to modify.

value (bool):

Determines whether the bit masks (false) or passes (true) the corresponding state.

query(*queries, check_errors=True)

Sends an SCPI query or multiple queries to the instrument and return the response(s).

Args:**queries (str):**

Any number of SCPI queries or commands.

check_errors (bool):

Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default. Optional Parameter.

Returns:

The instrument query response as a string.

reset_measurement_settings()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_operation_event_enable_mask(register_mask)

Configures the values of the operation event enable register bits.

These values determine which operation bits propagate to the operation event register.

Args:**register_mask ([Instrument]OperationRegister):**

An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(register_mask)

Configures the values of the questionable event enable register bits.

These values determine which questionable bits propagate to the questionable event register.

Args:**register_mask ([Instrument]QuestionableRegister):**

An instrument specific QuestionableRegister class object with all bits configured true or false.

set_service_request_enable_mask(register_mask)

Configures values of the service request enable register bits.

This register determines which bits propagate to the master summary bit.

Args:**register_mask (StatusByteRegister):**

A StatusByteRegister class object with all bits configured true or false.

set_standard_event_enable_mask(register_mask)

Configures values of the standard event enable register bits.

These values determine which bits propagate to the standard event register.

Args:**register_mask (StandardEventRegister):**

A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of `command`. Send a command to the instrument.

Args:

command_string (str):

A serial command.

PYTHON MODULE INDEX

|

lakeshore.em_power_supply, 22
lakeshore.fast_hall_controller, 34
lakeshore.model_121, 261
lakeshore.model_155, 264
lakeshore.model_224, 235
lakeshore.model_240, 253
lakeshore.model_335, 93
lakeshore.model_336, 122
lakeshore.model_346, 157
lakeshore.model_350, 194
lakeshore.model_372, 198
lakeshore.model_425, 20
lakeshore.ssm_measure_module, 71
lakeshore.ssm_settings_profiles, 87
lakeshore.ssm_source_module, 54
lakeshore.ssm_system, 46
lakeshore.teslameter, 10

INDEX

Symbols

- `__init__` () (*lakeshore.model_336.Model336ControlLoopZoneSettings* method), 146
- `__init__` () (*lakeshore.model_336.Model336InputSensorSettings* method), 146
- `__init__` () (*lakeshore.model_346.Model346AlarmSettings* method), 189
- `__init__` () (*lakeshore.model_346.Model346InputSensorSettings* method), 189
- `__init__` () (*lakeshore.model_372.CurveHeader* method), 225
- `__init__` () (*lakeshore.model_372.Model372AlarmParameters* method), 225
- `__init__` () (*lakeshore.model_372.Model372ControlLoopZoneSettings* method), 224
- `__init__` () (*lakeshore.model_372.Model372HeaterOutputSettings* method), 223
- `__init__` () (*lakeshore.model_372.Model372InputChannelSettings* method), 223
- `__init__` () (*lakeshore.model_372.Model372InputSetupSettings* method), 223
- `__init__` () (*lakeshore.temperature_controllers.AlarmSettings* method), 147
- `__init__` () (*lakeshore.temperature_controllers.CurveHeader* method), 147
- A**
- `abort_sweeps` () (*lakeshore.ssm_system.SSMSystem* method), 51
- `ACTIVE_SCAN_CHANNEL` (*lakeshore.model_372.Model372Enums.DisplayInfo* attribute), 231
- `ALARMS` (*lakeshore.model_224.Model224Enums.RelayControlMode* attribute), 250
- `ALARMS` (*lakeshore.model_346.Model346Enums.RelayControlMode* attribute), 192
- `ALARMS` (*lakeshore.model_372.Model372Enums.RelayControlMode* attribute), 230
- `AlarmSettings` (class in *lakeshore.temperature_controllers*), 147
- `all_heaters_off` () (*lakeshore.model_346.Model346* method), 161
- `ALL_INPUTS` (*lakeshore.model_224.Model224Enums.DisplayMode* attribute), 250
- `ALL_INPUTS` (*lakeshore.model_336.Model336Enums.DisplaySetupMode* attribute), 152
- `apply_ac_current` () (*lakeshore.ssm_source_module.SourceModule* method), 61
- `apply_ac_voltage` () (*lakeshore.ssm_source_module.SourceModule* method), 62
- `apply_dc_current` () (*lakeshore.ssm_source_module.SourceModule* method), 60
- `apply_dc_voltage` () (*lakeshore.ssm_source_module.SourceModule* method), 62
- `as_integer_ratio` () (*lakeshore.model_346.Model346.AutotuneMode* method), 168
- `as_integer_ratio` () (*lakeshore.model_346.Model346.BrightnessLevel* method), 169
- `as_integer_ratio` () (*lakeshore.model_346.Model346.ControlTypes* method), 170
- `as_integer_ratio` () (*lakeshore.model_346.Model346.CurveFormat* method), 171
- `as_integer_ratio` () (*lakeshore.model_346.Model346.CurveTemperature* method), 172
- `as_integer_ratio` () (*lakeshore.model_346.Model346.DiodeCurrent* method), 173
- `as_integer_ratio` () (*lakeshore.model_346.Model346.DisplayFields* method), 174
- `as_integer_ratio` () (*lakeshore.model_346.Model346.DisplayFieldsSize* method), 175
- `as_integer_ratio` () (*lakeshore.model_346.Model346.HeaterError* method), 176
- `as_integer_ratio` () (*lakeshore.model_346.Model346.Interface* method), 177
- `as_integer_ratio` () (*lakeshore.model_346.Model346.InterfaceMode* method), 178
- `as_integer_ratio` () (*lakeshore.model_346.Model346.LanStatus* method), 178
- `AUTO_OFF` (*lakeshore.model_335.Model335Enums.WarmupControl* attribute), 119
- `auto_phase` () (*lakeshore.ssm_measure_module.MeasureModule* method), 75

B

- bit_count() (lakeshore.model_346.Model346.AutotuneMode attribute), 168
- bit_count() (lakeshore.model_346.Model346.BrightnessLevel attribute), 169
- bit_count() (lakeshore.model_346.Model346.ControlTypes attribute), 170
- bit_count() (lakeshore.model_346.Model346.CurveFormat attribute), 171
- bit_count() (lakeshore.model_346.Model346.CurveTemperatureCoefficient attribute), 172
- bit_count() (lakeshore.model_346.Model346.DiodeCurrent attribute), 173
- bit_count() (lakeshore.model_346.Model346.DisplayFields attribute), 174
- bit_count() (lakeshore.model_346.Model346.DisplayFieldsSize attribute), 175
- bit_count() (lakeshore.model_346.Model346.HeaterError attribute), 176
- bit_count() (lakeshore.model_346.Model346.InterfaceMode attribute), 176
- bit_count() (lakeshore.model_346.Model346.InterfaceMode attribute), 177
- bit_count() (lakeshore.model_346.Model346.LanStatus attribute), 178
- bit_length() (lakeshore.model_346.Model346.AutotuneMode attribute), 168
- bit_length() (lakeshore.model_346.Model346.BrightnessLevel attribute), 169
- bit_length() (lakeshore.model_346.Model346.ControlTypes attribute), 170
- bit_length() (lakeshore.model_346.Model346.CurveFormat attribute), 171
- bit_length() (lakeshore.model_346.Model346.CurveTemperatureCoefficient attribute), 172
- bit_length() (lakeshore.model_346.Model346.DiodeCurrent attribute), 173
- bit_length() (lakeshore.model_346.Model346.DisplayFields attribute), 174
- bit_length() (lakeshore.model_346.Model346.DisplayFieldsSize attribute), 175
- bit_length() (lakeshore.model_346.Model346.HeaterError attribute), 175
- bit_length() (lakeshore.model_346.Model346.InterfaceMode attribute), 176
- bit_length() (lakeshore.model_346.Model346.InterfaceMode attribute), 177
- bit_length() (lakeshore.model_346.Model346.LanStatus attribute), 178
- bit_names (lakeshore.model_224.Model224ReadingStatus attribute), 252
- bit_names (lakeshore.model_224.Model224ServiceRequest attribute), 251
- bit_names (lakeshore.model_224.Model224StatusByteRegister attribute), 251
- bit_names (lakeshore.model_336.Model336InputReadingStatus attribute), 148
- bit_names (lakeshore.model_336.Model336ServiceRequestEnable attribute), 148
- bit_names (lakeshore.model_336.Model336StatusByteRegister attribute), 148
- bit_names (lakeshore.model_346.Model346OperationRegister attribute), 194
- bit_names (lakeshore.model_346.Model346QuestionableRegister attribute), 194
- bit_names (lakeshore.xip_instrument.StandardEventRegister attribute), 193
- bit_names (lakeshore.xip_instrument.StatusByteRegister attribute), 194
- BOTH_ALARMS (lakeshore.model_224.Model224Enums.RelayControlAlarm attribute), 250
- BOTH_ALARMS_ACTIVE (lakeshore.model_346.Model346Enums.RelayTherm attribute), 192

C

- CAPACITANCE (lakeshore.model_336.Model336Enums.InputSensorType attribute), 152
- CELSIUS (lakeshore.model_224.Model224Enums.DisplayFieldUnits attribute), 249
- CELSIUS (lakeshore.model_224.Model224Enums.InputSensorUnits attribute), 248
- CELSIUS (lakeshore.model_240.Model240Enums.Units attribute), 259
- CELSIUS (lakeshore.model_335.Model335Enums.DisplayFieldUnits attribute), 120
- CELSIUS (lakeshore.model_335.Model335Enums.MonitorOutUnits attribute), 118
- CELSIUS (lakeshore.model_336.Model336Enums.DisplayFieldUnits attribute), 153
- CELSIUS (lakeshore.model_346.Model346Enums.InputSensorUnits attribute), 190
- CHANNEL_A (lakeshore.model_335.Model335Enums.InputSensor attribute), 118
- CHANNEL_A (lakeshore.model_336.Model336Enums.InputChannel attribute), 151
- CHANNEL_B (lakeshore.model_335.Model335Enums.InputSensor attribute), 118
- CHANNEL_B (lakeshore.model_336.Model336Enums.InputChannel attribute), 151
- CHANNEL_C (lakeshore.model_336.Model336Enums.InputChannel attribute), 151
- CHANNEL_D (lakeshore.model_336.Model336Enums.InputChannel attribute), 151
- CHANNEL_D2 (lakeshore.model_336.Model336Enums.InputChannel attribute), 151
- CHANNEL_D3 (lakeshore.model_336.Model336Enums.InputChannel attribute), 151

CHANNEL_D4 (*lakeshore.model_336.Model336Enums.InputChannel* attribute), 151

CHANNEL_D5 (*lakeshore.model_336.Model336Enums.InputChannel* attribute), 151

clear_interface() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.Teslameter.Teslameter* method), 27

clear_interface_command() (*lakeshore.model_224.Model224* method), 236

clear_interface_command() (*lakeshore.model_346.Model346* method), 179

clear_relative_resistance_baseline() (*lakeshore.ssm_measure_module.MeasureModule* method), 85

CLOSED_LOOP (*lakeshore.model_335.Model335Enums.HeaterOutputMode* attribute), 119

CLOSED_LOOP (*lakeshore.model_336.Model336Enums.HeaterOutputMode* attribute), 153

CLOSED_LOOP (*lakeshore.model_372.Model372Enums.OutputMode* attribute), 229

command() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.Teslameter.Teslameter* method), 23

command() (*lakeshore.fast_hall_controller.FastHall* method), 38

command() (*lakeshore.model_121.Model121* method), 261

command() (*lakeshore.model_155.PrecisionSource* method), 267

command() (*lakeshore.model_224.Model224* method), 235

command() (*lakeshore.model_240.Model240* method), 258

command() (*lakeshore.model_346.Model346* method), 179

command() (*lakeshore.model_350.Model350* method), 194

command() (*lakeshore.model_425.Model425* method), 20

command() (*lakeshore.ssm_system.SSMSystem* method), 51

command() (*lakeshore.teslameter.Teslameter* method), 16

configure_cmF() (*lakeshore.ssm_source_module.SourceModule* method), 58

configure_cmr() (*lakeshore.ssm_source_module.SourceModule* method), 58

configure_corrected_analog_output_scaling() (*lakeshore.teslameter.Teslameter* method), 14

configure_current_range() (*lakeshore.ssm_measure_module.MeasureModule* method), 74

configure_current_range() (*lakeshore.ssm_source_module.SourceModule* method), 59

configure_display() (*lakeshore.model_224.Model224* method), 245

configure_field_control_limits() (*lakeshore.teslameter.Teslameter* method), 13

configure_field_control_output_mode() (*lakeshore.teslameter.Teslameter* method), 13

configure_field_control_pid() (*lakeshore.teslameter.Teslameter* method), 13

configure_field_measurement_setup() (*lakeshore.teslameter.Teslameter* method), 12

configure_field_units() (*lakeshore.teslameter.Teslameter* method), 13

configure_i_range() (*lakeshore.ssm_measure_module.MeasureModule* method), 74

configure_i_range() (*lakeshore.ssm_source_module.SourceModule* method), 59

configure_input() (*lakeshore.model_224.Model224* method), 244

configure_input_highpass_filter() (*lakeshore.ssm_measure_module.MeasureModule* method), 74

configure_input_lowpass_filter() (*lakeshore.ssm_measure_module.MeasureModule* method), 73

configure_mon_out() (*lakeshore.ssm_system.SSMSystem* method), 49

configure_mon_out_manual_mode() (*lakeshore.ssm_system.SSMSystem* method), 50

configure_qualifier() (*lakeshore.teslameter.Teslameter* method), 15

configure_ref_out() (*lakeshore.ssm_system.SSMSystem* method), 48

configure_sync() (*lakeshore.ssm_source_module.SourceModule* method), 56

configure_temperature_compensation() (*lakeshore.teslameter.Teslameter* method), 12

configure_voltage_range() (*lakeshore.ssm_measure_module.MeasureModule* method), 75

configure_voltage_range() (*lakeshore.ssm_source_module.SourceModule* method), 62

conjugate() (*lakeshore.model_346.Model346.AutotuneMode* method), 168

- conjugate() (*lakeshore.model_346.Model346.BrightnessLevel* method), 169
- conjugate() (*lakeshore.model_346.Model346.ControlType* method), 170
- conjugate() (*lakeshore.model_346.Model346.CurveForm* method), 171
- conjugate() (*lakeshore.model_346.Model346.CurveTemperatureCoefficient* method), 172
- conjugate() (*lakeshore.model_346.Model346.DiodeCurrent* method), 173
- conjugate() (*lakeshore.model_346.Model346.DisplayFieldOfView* method), 174
- conjugate() (*lakeshore.model_346.Model346.DisplayFieldOfView* method), 174
- conjugate() (*lakeshore.model_346.Model346.HeaterError* method), 175
- conjugate() (*lakeshore.model_346.Model346.Interface* method), 176
- conjugate() (*lakeshore.model_346.Model346.InterfaceMode* method), 177
- conjugate() (*lakeshore.model_346.Model346.LanStatus* method), 178
- connect_tcp() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 31
- connect_tcp() (*lakeshore.fast_hall_controller.FastHall* method), 38
- connect_tcp() (*lakeshore.model_121.Model121* method), 262
- connect_tcp() (*lakeshore.model_155.PrecisionSource* method), 267
- connect_tcp() (*lakeshore.model_224.Model224* method), 246
- connect_tcp() (*lakeshore.model_240.Model240* method), 258
- connect_tcp() (*lakeshore.model_346.Model346* method), 179
- connect_tcp() (*lakeshore.model_350.Model350* method), 195
- connect_tcp() (*lakeshore.model_425.Model425* method), 20
- connect_tcp() (*lakeshore.ssm_system.SSMSystem* method), 51
- connect_tcp() (*lakeshore.teslameter.Teslameter* method), 16
- connect_usb() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 31
- connect_usb() (*lakeshore.fast_hall_controller.FastHall* method), 38
- connect_usb() (*lakeshore.model_121.Model121* method), 262
- connect_usb() (*lakeshore.model_155.PrecisionSource* method), 267
- connect_usb() (*lakeshore.model_224.Model224* method), 246
- connect_usb() (*lakeshore.model_240.Model240* method), 258
- connect_usb() (*lakeshore.model_346.Model346* method), 179
- connect_usb() (*lakeshore.model_350.Model350* method), 195
- connect_usb() (*lakeshore.model_425.Model425* method), 20
- connect_usb() (*lakeshore.ssm_system.SSMSystem* method), 51
- connect_usb() (*lakeshore.teslameter.Teslameter* method), 16
- contact_check_manual_parameters (class in *lakeshore.fast_hall_controller*), 41
- contact_check_optimized_parameters (class in *lakeshore.fast_hall_controller*), 41
- continue_dc_hall() (*lakeshore.fast_hall_controller.FastHall* method), 34
- CONTINUOUS (*lakeshore.model_335.Model335Enums.WarmupControl* attribute), 119
- CONTROL (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229
- CONTROL_INPUT (*lakeshore.model_372.Model372Enums.DisplayMode* attribute), 230
- create() (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 87
- CS_NEG (*lakeshore.model_372.Model372Enums.MonitorOutputSource* attribute), 230
- CS_POS (*lakeshore.model_372.Model372Enums.MonitorOutputSource* attribute), 230
- CURRENT (*lakeshore.model_335.Model335Enums.HeaterOutputDisplay* attribute), 119
- CURRENT (*lakeshore.model_335.Model335Enums.HeaterOutType* attribute), 119
- CURRENT (*lakeshore.model_346.Model346Enums.HeaterOutputUnits* attribute), 191
- CURRENT (*lakeshore.model_372.Model372Enums.AutoRangeMode* attribute), 230
- CURRENT (*lakeshore.model_372.Model372Enums.SensorExcitationMode* attribute), 230
- CurveHeader (class in *lakeshore.model_372*), 225
- CurveHeader (class in *lakeshore.temperature_controllers*), 147
- CUSTOM (*lakeshore.model_224.Model224Enums.DisplayMode* attribute), 250
- CUSTOM (*lakeshore.model_335.Model335Enums.DisplaySetup* attribute), 120
- CUSTOM (*lakeshore.model_336.Model336Enums.DisplaySetupMode* attribute), 152
- CUSTOM (*lakeshore.model_372.Model372Enums.DisplayMode* attribute), 230

D

DataSourceMnemonic (class in

lakeshore.ssm_system_enums.SSMSystemEnums), `disable_ac_high_voltage_compliance()`
 89 (lakeshore.model_155.PrecisionSource method), 267
DCHallParameters (class in *lakeshore.fast_hall_controller*), 42 `disable_all_sweeping()`
`delete()` (lakeshore.ssm_settings_profiles.SettingsProfiles method), 88 (lakeshore.ssm_source_module.SourceModule method), 66
`delete_all()` (lakeshore.ssm_settings_profiles.SettingsProfiles method), 88 `disable_aurorange()`
 (lakeshore.model_155.PrecisionSource method), 266
`delete_curve()` (lakeshore.model_224.Model224 method), 242 `disable_bias_voltage()`
 (lakeshore.model_240.Model240 method), 254 (lakeshore.ssm_measure_module.MeasureModule method), 73
`delete_curve()` (lakeshore.model_346.Model346 method), 179 `disable_cmf()` (lakeshore.ssm_source_module.SourceModule method), 58
denominator (lakeshore.model_346.Model346.AutotuneMode attribute), 169 `disable_cmr()` (lakeshore.ssm_source_module.SourceModule method), 58
denominator (lakeshore.model_346.Model346.BrightnessLevel attribute), 170 `disable_current()` (lakeshore.model_121.Model121 method), 261
denominator (lakeshore.model_346.Model346.ControlType attribute), 171 `disable_guards()` (lakeshore.ssm_source_module.SourceModule method), 57
denominator (lakeshore.model_346.Model346.CurveForm attribute), 172 `disable_high_frequency_filters()`
 (lakeshore.teslameter.Teslameter method), 15
denominator (lakeshore.model_346.Model346.CurveTemperatureCoefficient attribute), 173 `disable_input()` (lakeshore.model_224.Model224 method), 244
denominator (lakeshore.model_346.Model346.DiodeCurrent attribute), 173 `disable_input_filters()`
 (lakeshore.ssm_measure_module.MeasureModule method), 74
denominator (lakeshore.model_346.Model346.DisplayFields attribute), 174 `disable_lock_in_averaging()`
 (lakeshore.ssm_measure_module.MeasureModule method), 77
denominator (lakeshore.model_346.Model346.DisplayFields attribute), 175 `disable_lock_in_fir()`
 (lakeshore.ssm_measure_module.MeasureModule method), 77
denominator (lakeshore.model_346.Model346.HeaterError attribute), 176 `disable_lock_in_iir()`
 (lakeshore.ssm_measure_module.MeasureModule method), 76
denominator (lakeshore.model_346.Model346.Interface attribute), 177 `disable_lock_in_lowpass()`
 (lakeshore.ssm_measure_module.MeasureModule method), 76
denominator (lakeshore.model_346.Model346.InterfaceMode attribute), 178 `disable_mon_out()` (lakeshore.ssm_system.SSMSystem method), 49
denominator (lakeshore.model_346.Model346.LanStatus attribute), 179 `disable_output()` (lakeshore.model_155.PrecisionSource method), 264
DIGITAL_INPUT (lakeshore.model_346.Model346.Enums.RelayControlMode attribute), 192 `disable_qualifier()`
 (lakeshore.teslameter.Teslameter method), 15
DIODE (lakeshore.model_224.Model224.Enums.InputSensorType attribute), 247 `disable_qualifier_latching()`
 (lakeshore.teslameter.Teslameter method), 15
DIODE (lakeshore.model_240.Model240.Enums.SensorTypes attribute), 259 `disable_sweeping()` (lakeshore.ssm_source_module.SourceModule method), 66
DIODE (lakeshore.model_335.Model335.Enums.InputSensorType attribute), 118
DIODE (lakeshore.model_336.Model336.Enums.InputSensorType attribute), 152
DIODE (lakeshore.model_346.Model346.Enums.InputSensorType attribute), 190
DIODE_DT670 (lakeshore.model_346.Model346.Enums.StandardCurveType attribute), 191
`disable()` (lakeshore.ssm_source_module.SourceModule method), 55

DISABLED (*lakeshore.model_335.Model335Enums.InputSensorType* method), 68
 attribute), 118
 DISABLED (*lakeshore.model_336.Model336Enums.InputSensorType* (*lakeshore.ssm_source_module.SourceModule*
 attribute), 152
 DISABLED (*lakeshore.model_346.Model346Enums.HeaterAndOrangeOffsetSweepStepAndMeasure* (*lakeshore.ssm_source_module.SourceModule*
 attribute), 191
 DISABLED (*lakeshore.model_346.Model346Enums.InputSensorType* method), 69
 attribute), 190
 disconnect_tcp() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* (*lakeshore.ssm_source_module.SourceModule*
 method), 31
 method), 70
 disconnect_tcp() (*lakeshore.fast_hall_controller.FastHall* do_dc_sweep_step_and_measure() (*lakeshore.ssm_source_module.SourceModule*
 method), 38
 method), 67
 disconnect_tcp() (*lakeshore.model_121.Model121*
 method), 262
 disconnect_tcp() (*lakeshore.model_155.PrecisionSource*
 method), 267
 disconnect_tcp() (*lakeshore.model_224.Model224*
 method), 246
 disconnect_tcp() (*lakeshore.model_240.Model240*
 method), 258
 disconnect_tcp() (*lakeshore.model_346.Model346*
 method), 179
 disconnect_tcp() (*lakeshore.model_350.Model350*
 method), 195
 disconnect_tcp() (*lakeshore.model_425.Model425*
 method), 20
 disconnect_tcp() (*lakeshore.ssm_system.SSMSystem*
 method), 51
 disconnect_tcp() (*lakeshore.teslameter.Teslameter*
 method), 16
 disconnect_usb() (*lakeshore.em_power_supply.ElectromagnetPowerSupply*
 method), 31
 disconnect_usb() (*lakeshore.fast_hall_controller.FastHall*
 method), 38
 disconnect_usb() (*lakeshore.model_121.Model121*
 method), 262
 disconnect_usb() (*lakeshore.model_155.PrecisionSource*
 method), 267
 disconnect_usb() (*lakeshore.model_224.Model224*
 method), 246
 disconnect_usb() (*lakeshore.model_240.Model240*
 method), 258
 disconnect_usb() (*lakeshore.model_346.Model346*
 method), 180
 disconnect_usb() (*lakeshore.model_350.Model350*
 method), 195
 disconnect_usb() (*lakeshore.model_425.Model425*
 method), 20
 disconnect_usb() (*lakeshore.ssm_system.SSMSystem*
 method), 52
 disconnect_usb() (*lakeshore.teslameter.Teslameter*
 method), 16
 do_ac_amplitude_sweep_step_and_measure()
 (*lakeshore.ssm_source_module.SourceModule*

do_ac_frequency_sweep_step_and_measure()
 (*lakeshore.ssm_source_module.SourceModule*
 method), 69
 do_ac_offset_sweep_step_and_measure()
 (*lakeshore.ssm_source_module.SourceModule*
 method), 70
 do_ac_parameter_sweep_step_and_measure()
 (*lakeshore.ssm_source_module.SourceModule*
 method), 67
 do_dc_sweep_step_and_measure()
 (*lakeshore.ssm_source_module.SourceModule*
 method), 67
 DT_400 (*lakeshore.model_224.Model224Enums.SoftCalSensorTypes*
 attribute), 251

E
 EIGHT (*lakeshore.model_372.Model372Enums.InputChannel*
 attribute), 229
 EITHER_ALARM_ACTIVE
 (*lakeshore.model_346.Model346Enums.RelayThermometryCond*
 attribute), 192
 ElectromagnetPowerSupply (class in
 lakeshore.em_power_supply), 22
 ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister
 (class in *lakeshore.em_power_supply*), 23
 ElectromagnetPowerSupply.EMPowerSupplyOperationalErrorsRegister
 (class in *lakeshore.em_power_supply*), 23
 ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister
 (class in *lakeshore.em_power_supply*), 23
 ElectromagnetPowerSupply.EMPowerSupplyServiceRequestEnable
 (class in *lakeshore.em_power_supply*), 22
 ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusF
 (class in *lakeshore.em_power_supply*), 23
 ElectromagnetPowerSupply.EMPowerSupplyStatusByteRegister
 (class in *lakeshore.em_power_supply*), 22
 ELEVEN (*lakeshore.model_372.Model372Enums.InputChannel*
 attribute), 229
 enable() (*lakeshore.ssm_source_module.SourceModule*
 method), 55
 enable_ac_high_voltage_compliance()
 (*lakeshore.model_155.PrecisionSource*
 method), 267
 enable_aurorange() (*lakeshore.model_155.PrecisionSource*
 method), 266
 enable_bias_voltage()
 (*lakeshore.ssm_measure_module.MeasureModule*
 method), 73
 enable_cmfc() (*lakeshore.ssm_source_module.SourceModule*
 method), 58
 enable_cmrc() (*lakeshore.ssm_source_module.SourceModule*
 method), 58
 enable_current() (*lakeshore.model_121.Model121*
 method), 261

enable_guards() (*lakeshore.ssm_source_module.SourceModule* method), 57
 enable_high_frequency_filters() (*lakeshore.teslameter.Teslameter* method), 14
 enable_lock_in_averaging() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 enable_lock_in_fir() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 enable_lock_in_iir() (*lakeshore.ssm_measure_module.MeasureModule* method), 76
 enable_lock_in_lowpass() (*lakeshore.ssm_measure_module.MeasureModule* method), 76
 enable_mon_out() (*lakeshore.ssm_system.SSMSystem* method), 49
 enable_output() (*lakeshore.model_155.PrecisionSource* method), 264
 enable_qualifier() (*lakeshore.teslameter.Teslameter* method), 15
 enable_qualifier_latching() (*lakeshore.teslameter.Teslameter* method), 15
 enable_ref_out() (*lakeshore.ssm_system.SSMSystem* method), 48
 ENABLED (*lakeshore.model_346.Model346Enums.HeaterAuto* attribute), 191
 ETHERNET (*lakeshore.model_224.Model224Enums.RemoteInterface* attribute), 249
F
 factory_reset() (*lakeshore.fast_hall_controller.FastHall* method), 38
 factory_reset() (*lakeshore.model_155.PrecisionSource* method), 267
 factory_reset() (*lakeshore.model_346.Model346* method), 180
 factory_reset() (*lakeshore.ssm_system.SSMSystem* method), 52
 factory_reset() (*lakeshore.teslameter.Teslameter* method), 16
 FAHRENHEIT (*lakeshore.model_240.Model240Enums.Units* attribute), 259
 FALSE (*lakeshore.model_346.Model346Enums.RelayFunction* attribute), 193
 FastHall (class in *lakeshore.fast_hall_controller*), 34
 FastHallLinkParameters (class in *lakeshore.fast_hall_controller*), 42
 FastHallManualParameters (class in *lakeshore.fast_hall_controller*), 41
 FastHallOperationRegister (class in *lakeshore.fast_hall_controller*), 41
 FastHallQuestionableRegister (class in *lakeshore.fast_hall_controller*), 41
 fetch_multiple() (*lakeshore.ssm_measure_module.MeasureModule* method), 86
 fetch_multiple() (*lakeshore.ssm_source_module.SourceModule* method), 54
 fetch_multiple() (*lakeshore.ssm_system.SSMSystem* method), 50
 FIFTEEN (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229
 FIFTY_MILLIVOLT (*lakeshore.model_335.Model335Enums.Thermocouple* attribute), 119
 FIFTY_MILLIVOLT (*lakeshore.model_336.Model336Enums.Thermocouple* attribute), 153
 FIFTY_MILLIVOLTS (*lakeshore.model_346.Model346Enums.Thermocouple* attribute), 191
 FIVE (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229
 FOUR (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229
 FOUR_LOOP (*lakeshore.model_336.Model336Enums.DisplaySetupMode* attribute), 152
 FOURTEEN (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229
 FourWireParameters (class in *lakeshore.fast_hall_controller*), 42
 FREQUENCY_11_POINT_6_HZ (*lakeshore.model_372.Model372Enums.InputFrequency* attribute), 232
 FREQUENCY_13_POINT_7_HZ (*lakeshore.model_372.Model372Enums.InputFrequency* attribute), 232
 FREQUENCY_16_POINT_2_HZ (*lakeshore.model_372.Model372Enums.InputFrequency* attribute), 232
 FREQUENCY_18_POINT_2_HZ (*lakeshore.model_372.Model372Enums.InputFrequency* attribute), 232
 FREQUENCY_9_POINT_8_HZ (*lakeshore.model_372.Model372Enums.InputFrequency* attribute), 231
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 22
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23
 from_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* class method), 23

class method), 23

FUNCTION_OUTPUT (*lakeshore.model_346.Model346Enums.RelayControlMode* attribute), 192

G

GENERAL_PURPOSE_INPUT_STATES (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceAttribute* attribute), 90

GENERAL_PURPOSE_OUTPUT_STATES (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceAttribute* attribute), 90

generate_and_apply_soft_cal_curve() (*lakeshore.model_224.Model224* method), 242

get_actual_averaging_time() (*lakeshore.ssm_measure_module.MeasureModule* method), 72

get_actual_readback_nplcycles() (*lakeshore.ssm_source_module.SourceModule* method), 68

get_alarm_parameters() (*lakeshore.model_224.Model224* method), 240

get_alarm_settings() (*lakeshore.model_346.Model346* method), 159

get_alarm_status() (*lakeshore.model_224.Model224* method), 241

get_all_inputs_celsius_reading() (*lakeshore.model_224.Model224* method), 237

get_analog_out_state() (*lakeshore.model_346.Model346* method), 161

get_analog_output() (*lakeshore.teslameter.Teslameter* method), 14

get_analog_output_settings() (*lakeshore.model_346.Model346* method), 164

get_analog_output_signal() (*lakeshore.teslameter.Teslameter* method), 14

get_automatic_coupling() (*lakeshore.ssm_source_module.SourceModule* method), 57

get_averaging_time() (*lakeshore.ssm_measure_module.MeasureModule* method), 72

get_band_pass_filter_center() (*lakeshore.teslameter.Teslameter* method), 15

get_bias_voltage() (*lakeshore.ssm_measure_module.MeasureModule* method), 73

get_bias_voltage_enabled() (*lakeshore.ssm_measure_module.MeasureModule* method), 73

get_brightness() (*lakeshore.model_240.Model240* method), 253

get_buffered_data_points() (*lakeshore.teslameter.Teslameter* method), 10

get_calculated_heater_output() (*lakeshore.model_346.Model346* method), 162

get_celsius_reading() (*lakeshore.model_224.Model224* method), 237

get_celsius_reading() (*lakeshore.model_240.Model240* method), 253

get_channel_reading_status() (*lakeshore.model_240.Model240* method), 257

get_cmf_source() (*lakeshore.ssm_source_module.SourceModule* method), 57

get_cmf_state() (*lakeshore.ssm_source_module.SourceModule* method), 58

get_cmr_source() (*lakeshore.ssm_source_module.SourceModule* method), 57

get_cmr_state() (*lakeshore.ssm_source_module.SourceModule* method), 58

get_compliance_limit_status() (*lakeshore.model_121.Model121* method), 262

get_contact_check_measurement_results() (*lakeshore.fast_hall_controller.FastHall* method), 35

get_contact_check_running_status() (*lakeshore.fast_hall_controller.FastHall* method), 34

get_contact_check_setup_results() (*lakeshore.fast_hall_controller.FastHall* method), 35

get_control_loop_zone_table() (*lakeshore.model_346.Model346* method), 166

get_control_setpoint() (*lakeshore.model_346.Model346* method), 180

get_corrected_analog_output_scaling() (*lakeshore.teslameter.Teslameter* method), 14

get_coupling() (*lakeshore.ssm_measure_module.MeasureModule* method), 72

get_coupling() (*lakeshore.ssm_source_module.SourceModule* method), 56

get_current() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 73

method), 25
 get_current() (*lakeshore.model_121.Model121 method*), 261
 get_current_ac_range() (*lakeshore.ssm_source_module.SourceModule method*), 59
 get_current_amplitude() (*lakeshore.ssm_source_module.SourceModule method*), 60
 get_current_autorange_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 74
 get_current_autorange_status() (*lakeshore.ssm_source_module.SourceModule method*), 59
 get_current_dc_range() (*lakeshore.ssm_source_module.SourceModule method*), 59
 get_current_limit() (*lakeshore.ssm_source_module.SourceModule method*), 61
 get_current_limit_status() (*lakeshore.ssm_source_module.SourceModule method*), 61
 get_current_offset() (*lakeshore.ssm_source_module.SourceModule method*), 60
 get_current_output_limit_high() (*lakeshore.ssm_source_module.SourceModule method*), 65
 get_current_output_limit_low() (*lakeshore.ssm_source_module.SourceModule method*), 65
 get_current_range() (*lakeshore.ssm_measure_module.MeasureModule method*), 74
 get_current_range() (*lakeshore.ssm_source_module.SourceModule method*), 58
 get_curve() (*lakeshore.model_224.Model224 method*), 242
 get_curve() (*lakeshore.model_346.Model346 method*), 180
 get_curve_data_point() (*lakeshore.model_224.Model224 method*), 242
 get_curve_data_point() (*lakeshore.model_240.Model240 method*), 255
 get_curve_data_point() (*lakeshore.model_346.Model346 method*), 180
 get_curve_header() (*lakeshore.model_224.Model224 method*), 241
 get_curve_header() (*lakeshore.model_240.Model240 method*), 254
 get_curve_header() (*lakeshore.model_346.Model346 method*), 180
 get_curve_number_of_points() (*lakeshore.model_346.Model346 method*), 167
 get_dark_mode_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 81
 get_dark_mode_state() (*lakeshore.ssm_source_module.SourceModule method*), 64
 get_data() (*lakeshore.ssm_system.SSMSystem method*), 47
 get_dc() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
 get_dc_field() (*lakeshore.teslameter.Teslameter method*), 11
 get_dc_field_xyz() (*lakeshore.teslameter.Teslameter method*), 11
 get_dc_hall_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 36
 get_dc_hall_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 34
 get_dc_hall_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 36
 get_dc_hall_waiting_status() (*lakeshore.fast_hall_controller.FastHall method*), 34
 get_dc_maximum() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
 get_dc_minimum() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
 get_dc_relative() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
 get_description() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 87
 get_detected_line_frequency() (*lakeshore.ssm_system.SSMSystem method*), 50
 get_digital_high_pass_filter_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 82
 get_digital_input_status() (*lakeshore.model_346.Model346 method*), 158
 get_disable_on_compliance() (*lakeshore.ssm_source_module.SourceModule method*), 65
 get_display_brightness() (*lakeshore.em_power_supply.ElectromagnetPowerSupply*

method), 26
get_display_brightness() (*lakeshore.model_121.Model121 method*), 261
get_display_configuration() (*lakeshore.model_224.Model224 method*), 245
get_display_contrast() (*lakeshore.model_224.Model224 method*), 238
get_display_field_settings() (*lakeshore.model_224.Model224 method*), 245
get_duty() (*lakeshore.ssm_source_module.SourceModule method*), 56
get_enable_state() (*lakeshore.ssm_source_module.SourceModule method*), 55
get_fahrenheit_reading() (*lakeshore.model_240.Model240 method*), 254
get_fasthall_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 35
get_fasthall_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 34
get_fasthall_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 35
get_field_control_limits() (*lakeshore.teslameter.Teslameter method*), 13
get_field_control_open_loop_voltage() (*lakeshore.teslameter.Teslameter method*), 14
get_field_control_output_mode() (*lakeshore.teslameter.Teslameter method*), 13
get_field_control_pid() (*lakeshore.teslameter.Teslameter method*), 13
get_field_control_setpoint() (*lakeshore.teslameter.Teslameter method*), 13
get_field_measurement_setup() (*lakeshore.teslameter.Teslameter method*), 12
get_field_units() (*lakeshore.teslameter.Teslameter method*), 13
get_filter() (*lakeshore.model_224.Model224 method*), 243
get_filter() (*lakeshore.model_240.Model240 method*), 255
get_filter() (*lakeshore.model_346.Model346 method*), 157
get_filter_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 73
get_four_wire_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 36
get_four_wire_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 34
get_four_wire_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 36
get_frequency() (*lakeshore.ssm_source_module.SourceModule method*), 56
get_frequency() (*lakeshore.teslameter.Teslameter method*), 11
get_frequency_filter_type() (*lakeshore.teslameter.Teslameter method*), 14
get_frequency_range_threshold() (*lakeshore.ssm_measure_module.MeasureModule method*), 81
get_front_panel_lock_code() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 27
get_front_panel_lock_status() (*lakeshore.model_121.Model121 method*), 262
get_front_panel_status() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 26
get_gain_allocation_strategy() (*lakeshore.ssm_measure_module.MeasureModule method*), 73
get_guard_state() (*lakeshore.ssm_source_module.SourceModule method*), 57
get_hardware_error_condition() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 30
get_hardware_error_enable_mask() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 30
get_hardware_error_event() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 30
get_head_cal_datetime() (*lakeshore.ssm_system.SSMSystem method*), 49
get_head_cal_temperature() (*lakeshore.ssm_system.SSMSystem method*), 49
get_head_self_cal_datetime() (*lakeshore.ssm_system.SSMSystem method*), 49
get_head_self_cal_status() (*lakeshore.ssm_system.SSMSystem method*), 49
get_head_self_cal_temperature()

(*lakeshore.ssm_system.SSMSystem* method), 49

`get_heater_autorange()` (*lakeshore.model_346.Model346* method), 160

`get_heater_limit()` (*lakeshore.model_346.Model346* method), 166

`get_heater_output()` (*lakeshore.model_346.Model346* method), 180

`get_heater_output_measurements()` (*lakeshore.model_346.Model346* method), 162

`get_heater_pid()` (*lakeshore.model_346.Model346* method), 181

`get_heater_range()` (*lakeshore.model_346.Model346* method), 160

`get_heater_setup()` (*lakeshore.model_346.Model346* method), 162

`get_heater_status()` (*lakeshore.model_346.Model346* method), 181

`get_high_pass_filter_cutoff()` (*lakeshore.teslameter.Teslameter* method), 15

`get_highpass_corner_frequency()` (*lakeshore.ssm_measure_module.MeasureModule* method), 73

`get_highpass_rolloff()` (*lakeshore.ssm_measure_module.MeasureModule* method), 73

`get_hw_version()` (*lakeshore.ssm_measure_module.MeasureModule* method), 71

`get_hw_version()` (*lakeshore.ssm_source_module.SourceModule* method), 55

`get_i_ac_range()` (*lakeshore.ssm_source_module.SourceModule* method), 59

`get_i_amplitude()` (*lakeshore.ssm_source_module.SourceModule* method), 60

`get_i_autorange_status()` (*lakeshore.ssm_measure_module.MeasureModule* method), 74

`get_i_autorange_status()` (*lakeshore.ssm_source_module.SourceModule* method), 59

`get_i_dc_range()` (*lakeshore.ssm_source_module.SourceModule* method), 59

`get_i_limit()` (*lakeshore.ssm_source_module.SourceModule* method), 61

`get_i_limit_status()` (*lakeshore.ssm_source_module.SourceModule* method), 61

`get_i_offset()` (*lakeshore.ssm_source_module.SourceModule* method), 60

`get_i_range()` (*lakeshore.ssm_measure_module.MeasureModule* method), 74

`get_i_range()` (*lakeshore.ssm_source_module.SourceModule* method), 58

`get_identification()` (*lakeshore.model_240.Model240* method), 253

`get_identify_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 81

`get_identify_state()` (*lakeshore.ssm_source_module.SourceModule* method), 64

`get_iee_488()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27

`get_ieee_488()` (*lakeshore.model_224.Model224* method), 239

`get_ieee_interface_mode()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27

`get_input_configuration()` (*lakeshore.model_224.Model224* method), 244

`get_input_configuration()` (*lakeshore.ssm_measure_module.MeasureModule* method), 72

`get_input_curve()` (*lakeshore.model_224.Model224* method), 240

`get_input_curve()` (*lakeshore.model_346.Model346* method), 181

`get_input_diode_excitation_current()` (*lakeshore.model_224.Model224* method), 238

`get_input_parameter()` (*lakeshore.model_240.Model240* method), 256

`get_input_reading_operation_status()` (*lakeshore.model_346.Model346* method), 164

`get_input_reading_status()` (*lakeshore.model_346.Model346* method), 164

`get_input_sensor()` (*lakeshore.model_346.Model346* method), 160

`get_installed_option_cards()` (*lakeshore.model_346.Model346* method), 168

`get_interface_mode()` (*lakeshore.model_224.Model224* method), 244

`get_internal_water()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 26

`get_json()` (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 87

`get_kelvin_reading()` (*lakeshore.model_224.Model224* method), 237
`get_kelvin_reading()` (*lakeshore.model_240.Model240* method), 254
`get_kelvin_reading()` (*lakeshore.model_346.Model346* method), 181
`get_keypad_lock()` (*lakeshore.model_224.Model224* method), 239
`get_keypad_lock()` (*lakeshore.model_346.Model346* method), 164
`get_led_state()` (*lakeshore.model_224.Model224* method), 239
`get_limits()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24
`get_line_frequency()` (*lakeshore.ssm_system.SSMSystem* method), 50
`get_line_frequency_detection_error_status()` (*lakeshore.ssm_system.SSMSystem* method), 50
`get_list()` (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 87
`get_load_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 86
`get_load_state()` (*lakeshore.ssm_source_module.SourceModule* method), 65
`get_lock_in_averaging_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 77
`get_lock_in_dc()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_dc_immediate()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_equivalent_noise_bandwidth()` (*lakeshore.ssm_measure_module.MeasureModule* method), 76
`get_lock_in_fir_cycles()` (*lakeshore.ssm_measure_module.MeasureModule* method), 77
`get_lock_in_fir_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 77
`get_lock_in_frequency()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_iir_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 76
`get_lock_in_lowpass_state()` (*lakeshore.ssm_measure_module.MeasureModule* method), 76
`get_lock_in_r()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_r_maximum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_r_minimum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_reference_cycles()` (*lakeshore.ssm_measure_module.MeasureModule* method), 77
`get_lock_in_rolloff()` (*lakeshore.ssm_measure_module.MeasureModule* method), 76
`get_lock_in_settle_time()` (*lakeshore.ssm_measure_module.MeasureModule* method), 76
`get_lock_in_theta()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_theta_maximum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_theta_minimum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 80
`get_lock_in_time_constant()` (*lakeshore.ssm_measure_module.MeasureModule* method), 75
`get_lock_in_x()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_x_maximum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_x_minimum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_y()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_y_maximum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_lock_in_y_minimum()` (*lakeshore.ssm_measure_module.MeasureModule* method), 79
`get_low_pass_filter_cutoff()` (*lakeshore.teslameter.Teslameter* method), 14
`get_lowpass_corner_frequency()` (*lakeshore.ssm_measure_module.MeasureModule* method), 73
`get_lowpass_rolloff()` (*lakeshore.ssm_measure_module.MeasureModule* method), 73
`get_magnet_water()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24

method), 26

get_manual_output() (lakeshore.model_346.Model346 method), 182

get_max_min() (lakeshore.teslameter.Teslameter method), 11

get_max_min_peaks() (lakeshore.teslameter.Teslameter method), 11

get_measure_function() (lakeshore.ssm_measure_module.MeasureModule method), 86

get_measure_module() (lakeshore.ssm_system.SSMSystem method), 47

get_measure_module_by_name() (lakeshore.ssm_system.SSMSystem method), 47

get_measure_pod() (lakeshore.ssm_system.SSMSystem method), 47

get_measured_current() (lakeshore.em_power_supply.ElectromagnetPowerSupply method), 25

get_measured_voltage() (lakeshore.em_power_supply.ElectromagnetPowerSupply method), 25

get_min_max_data() (lakeshore.model_224.Model224 method), 239

get_min_max_data() (lakeshore.model_346.Model346 method), 182

get_mode() (lakeshore.ssm_measure_module.MeasureModule method), 72

get_model() (lakeshore.ssm_measure_module.MeasureModule method), 71

get_model() (lakeshore.ssm_source_module.SourceModule method), 55

get_modname() (lakeshore.model_240.Model240 method), 256

get_mon_out_manual_level() (lakeshore.ssm_system.SSMSystem method), 50

get_mon_out_mode() (lakeshore.ssm_system.SSMSystem method), 49

get_mon_out_scale() (lakeshore.ssm_system.SSMSystem method), 49

get_mon_out_state() (lakeshore.ssm_system.SSMSystem method), 49

get_multiple() (lakeshore.ssm_measure_module.MeasureModule method), 78

get_multiple() (lakeshore.ssm_source_module.SourceModule method), 54

get_multiple() (lakeshore.ssm_system.SSMSystem method), 47

get_multiple_min_max_values() (lakeshore.ssm_system.SSMSystem method), 47

get_name() (lakeshore.ssm_measure_module.MeasureModule method), 71

get_name() (lakeshore.ssm_source_module.SourceModule method), 54

get_negative_peak() (lakeshore.ssm_measure_module.MeasureModule method), 79

get_negative_peak_maximum() (lakeshore.ssm_measure_module.MeasureModule method), 79

get_negative_peak_minimum() (lakeshore.ssm_measure_module.MeasureModule method), 79

get_notes() (lakeshore.ssm_measure_module.MeasureModule method), 71

get_notes() (lakeshore.ssm_source_module.SourceModule method), 54

get_num_measure_channels() (lakeshore.ssm_system.SSMSystem method), 46

get_num_source_channels() (lakeshore.ssm_system.SSMSystem method), 46

get_operation_event_condition() (lakeshore.em_power_supply.ElectromagnetPowerSupply method), 29

get_operation_event_enable_mask() (lakeshore.em_power_supply.ElectromagnetPowerSupply method), 29

get_operation_event_enable_mask() (lakeshore.fast_hall_controller.FastHall method), 38

get_operation_event_enable_mask() (lakeshore.model_155.PrecisionSource method), 267

get_operation_event_enable_mask() (lakeshore.model_346.Model346 method), 182

get_operation_event_enable_mask() (lakeshore.ssm_measure_module.MeasureModule method), 81

get_operation_event_enable_mask() (lakeshore.ssm_source_module.SourceModule method), 63

get_operation_event_enable_mask() (lakeshore.ssm_system.SSMSystem method), 52

get_operation_event_enable_mask() (lakeshore.teslameter.Teslameter method), 16

get_operation_event_event() (lakeshore.em_power_supply.ElectromagnetPowerSupply method), 29

get_operation_events() (lakeshore.fast_hall_controller.FastHall method), 38

get_operation_events() (lakeshore.model_155.PrecisionSource method), 267

`get_operation_events()` (*lakeshore.model_346.Model346 method*), 182
`get_operation_events()` (*lakeshore.ssm_measure_module.MeasureModule method*), 81
`get_operation_events()` (*lakeshore.ssm_source_module.SourceModule method*), 63
`get_operation_events()` (*lakeshore.ssm_system.SSMSystem method*), 52
`get_operation_events()` (*lakeshore.teslameter.Teslameter method*), 16
`get_operational_error_condition()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 31
`get_operational_error_enable_mask()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 31
`get_operational_error_event()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 31
`get_option_card_channels()` (*lakeshore.model_346.Model346 method*), 168
`get_option_card_slot()` (*lakeshore.model_346.Model346 method*), 158
`get_option_card_type()` (*lakeshore.model_346.Model346 method*), 168
`get_output_group()` (*lakeshore.model_346.Model346 method*), 163
`get_output_mode()` (*lakeshore.model_346.Model346 method*), 163
`get_output_operation_status()` (*lakeshore.model_346.Model346 method*), 165
`get_output_percent_limit()` (*lakeshore.model_346.Model346 method*), 161
`get_output_settings()` (*lakeshore.model_155.PrecisionSource method*), 266
`get_output_status()` (*lakeshore.model_346.Model346 method*), 165
`get_overload_status()` (*lakeshore.ssm_measure_module.MeasureModule method*), 80
`get_peak_to_peak()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_peak_to_peak_maximum()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_peak_to_peak_minimum()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_pll_lock_status()` (*lakeshore.ssm_measure_module.MeasureModule method*), 80
`get_positive_peak()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_positive_peak_maximum()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_positive_peak_minimum()` (*lakeshore.ssm_measure_module.MeasureModule method*), 79
`get_present_operation_status()` (*lakeshore.fast_hall_controller.FastHall method*), 38
`get_present_operation_status()` (*lakeshore.model_155.PrecisionSource method*), 267
`get_present_operation_status()` (*lakeshore.model_346.Model346 method*), 182
`get_present_operation_status()` (*lakeshore.ssm_measure_module.MeasureModule method*), 80
`get_present_operation_status()` (*lakeshore.ssm_source_module.SourceModule method*), 63
`get_present_operation_status()` (*lakeshore.ssm_system.SSMSystem method*), 52
`get_present_operation_status()` (*lakeshore.teslameter.Teslameter method*), 16
`get_present_questionable_status()` (*lakeshore.fast_hall_controller.FastHall method*), 38
`get_present_questionable_status()` (*lakeshore.model_155.PrecisionSource method*), 267
`get_present_questionable_status()` (*lakeshore.model_346.Model346 method*), 182
`get_present_questionable_status()` (*lakeshore.ssm_measure_module.MeasureModule method*), 80
`get_present_questionable_status()` (*lakeshore.ssm_source_module.SourceModule method*), 63
`get_present_questionable_status()` (*lakeshore.ssm_system.SSMSystem method*), 52

<code>get_present_questionable_status()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 16	(<i>lakeshore.model_346.Model346</i> method), 182
<code>get_probe_information()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 11	<code>get_questionable_events()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 80
<code>get_profibus_address()</code> (<i>lakeshore.model_240.Model240</i> method), 257	<code>get_questionable_events()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 63
<code>get_profibus_connection_status()</code> (<i>lakeshore.model_240.Model240</i> method), 257	<code>get_questionable_events()</code> (<i>lakeshore.ssm_system.SSMSystem</i> method), 52
<code>get_profibus_slot_configuration()</code> (<i>lakeshore.model_240.Model240</i> method), 257	<code>get_questionable_events()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 17
<code>get_profibus_slot_count()</code> (<i>lakeshore.model_240.Model240</i> method), 256	<code>get_ramp_rate()</code> (<i>lakeshore.em_power_supply.ElectromagnetPowerSupply</i> method), 24
<code>get_programming_mode()</code> (<i>lakeshore.em_power_supply.ElectromagnetPowerSupply</i> method), 27	<code>get_ramp_segment()</code> (<i>lakeshore.em_power_supply.ElectromagnetPowerSupply</i> method), 25
<code>get_qualifier_configuration()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 15	<code>get_ramp_segments_enable()</code> (<i>lakeshore.em_power_supply.ElectromagnetPowerSupply</i> method), 25
<code>get_qualifier_latching_setting()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 15	<code>get_ramp_setpoint()</code> (<i>lakeshore.model_346.Model346</i> method), 166
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.fast_hall_controller.FastHall</i> method), 38	<code>get_readback_dc()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 68
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.model_155.PrecisionSource</i> method), 267	<code>get_readback_nplcycles()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 68
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.model_346.Model346</i> method), 182	<code>get_readback_rms()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 68
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 80	<code>get_reading_status()</code> (<i>lakeshore.model_224.Model224</i> method), 236
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 63	<code>get_ref_in_edge()</code> (<i>lakeshore.ssm_system.SSMSystem</i> method), 48
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.ssm_system.SSMSystem</i> method), 52	<code>get_ref_out_source()</code> (<i>lakeshore.ssm_system.SSMSystem</i> method), 48
<code>get_questionable_event_enable_mask()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 16	<code>get_ref_out_state()</code> (<i>lakeshore.ssm_system.SSMSystem</i> method), 48
<code>get_questionable_events()</code> (<i>lakeshore.fast_hall_controller.FastHall</i> method), 38	<code>get_reference_harmonic()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 75
<code>get_questionable_events()</code> (<i>lakeshore.model_155.PrecisionSource</i> method), 267	<code>get_reference_phase_shift()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 75
<code>get_questionable_events()</code>	<code>get_reference_source()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 75
	<code>get_relative_baseline()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 78
	<code>get_relative_field()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 11

<code>get_relative_field_baseline()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 12	<code>get_resistance_observation_time_actual()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 85
<code>get_relay_alarm_control_parameters()</code> (<i>lakeshore.model_224.Model224</i> method), 246	<code>get_resistance_observation_time_enbw()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 85
<code>get_relay_alarm_control_parameters()</code> (<i>lakeshore.model_346.Model346</i> method), 167	<code>get_resistance_observation_time_requested()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 85
<code>get_relay_control_mode()</code> (<i>lakeshore.model_224.Model224</i> method), 246	<code>get_resistance_observation_time_state()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 84
<code>get_relay_control_mode()</code> (<i>lakeshore.model_346.Model346</i> method), 182	<code>get_resistance_optimization_state()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 84
<code>get_relay_status()</code> (<i>lakeshore.model_224.Model224</i> method), 243	<code>get_resistance_phase()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82
<code>get_relay_status()</code> (<i>lakeshore.model_346.Model346</i> method), 183	<code>get_resistance_quadrature()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82
<code>get_remote_interface()</code> (<i>lakeshore.model_224.Model224</i> method), 244	<code>get_resistance_range()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 84
<code>get_requested_averaging_time()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 72	<code>get_resistance_relative_baseline()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 85
<code>get_requested_readback_nplcycles()</code> (<i>lakeshore.ssm_source_module.SourceModule</i> method), 68	<code>get_resistance_relative_dc()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 85
<code>get_resistance()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82	<code>get_resistance_source()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 83
<code>get_resistance_auto_range()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 84	<code>get_resistivity_measurement_results()</code> (<i>lakeshore.fast_hall_controller.FastHall</i> method), 36
<code>get_resistance_capacitance()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 83	<code>get_resistivity_running_status()</code> (<i>lakeshore.fast_hall_controller.FastHall</i> method), 34
<code>get_resistance_compensated()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 83	<code>get_resistivity_setup_results()</code> (<i>lakeshore.fast_hall_controller.FastHall</i> method), 36
<code>get_resistance_dc()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82	<code>get_rms()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 79
<code>get_resistance_excitation_type()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 83	<code>get_rms_field()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 11
<code>get_resistance_inphase()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82	<code>get_rms_field_xyz()</code> (<i>lakeshore.teslameter.Teslameter</i> method), 11
<code>get_resistance_magnitude()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 82	<code>get_rms_maximum()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 79
<code>get_resistance_mode()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 84	<code>get_rms_minimum()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i> method), 79
	<code>get_rms_relative()</code> (<i>lakeshore.ssm_measure_module.MeasureModule</i>

<code>method</code>), 79	<code>(lakeshore.em_power_supply.ElectromagnetPowerSupply</code>
<code>get_self_cal_datetime()</code>	<code>method)</code> , 28
<code>(lakeshore.ssm_measure_module.MeasureModule</code>	<code>get_service_request_enable_mask()</code>
<code>method)</code> , 86	<code>(lakeshore.fast_hall_controller.FastHall</code>
<code>get_self_cal_datetime()</code>	<code>method)</code> , 38
<code>(lakeshore.ssm_source_module.SourceModule</code>	<code>get_service_request_enable_mask()</code>
<code>method)</code> , 65	<code>(lakeshore.model_155.PrecisionSource</code>
<code>get_self_cal_status()</code>	<code>method)</code> , 268
<code>(lakeshore.ssm_measure_module.MeasureModule</code>	<code>get_service_request_enable_mask()</code>
<code>method)</code> , 71	<code>(lakeshore.model_346.Model346</code>
<code>get_self_cal_status()</code>	<code>method)</code> , 183
<code>(lakeshore.ssm_source_module.SourceModule</code>	<code>get_service_request_enable_mask()</code>
<code>method)</code> , 55	<code>(lakeshore.ssm_system.SSMSystem</code>
<code>get_self_cal_temperature()</code>	<code>method)</code> , 52
<code>(lakeshore.ssm_measure_module.MeasureModule</code>	<code>get_service_request_enable_mask()</code>
<code>method)</code> , 86	<code>(lakeshore.teslameter.Teslameter</code>
<code>get_self_cal_temperature()</code>	<code>method)</code> , 17
<code>(lakeshore.ssm_source_module.SourceModule</code>	<code>get_setpoint_ramp_parameter()</code>
<code>method)</code> , 66	<code>(lakeshore.model_346.Model346</code>
<code>get_self_test()</code> <code>(lakeshore.em_power_supply.ElectromagnetPowerSupply</code>	<code>method)</code> , 183
<code>method)</code> , 28	<code>get_setpoint_ramp_status()</code>
<code>get_self_test()</code> <code>(lakeshore.model_224.Model224</code>	<code>(lakeshore.model_346.Model346</code>
<code>method)</code> , 236	<code>method)</code> , 184
<code>get_self_test()</code> <code>(lakeshore.model_346.Model346</code>	<code>get_settling_status()</code>
<code>method)</code> , 183	<code>(lakeshore.ssm_measure_module.MeasureModule</code>
<code>get_sensor_name()</code> <code>(lakeshore.model_224.Model224</code>	<code>method)</code> , 80
<code>method)</code> , 238	<code>get_shape()</code> <code>(lakeshore.ssm_source_module.SourceModule</code>
<code>get_sensor_name()</code> <code>(lakeshore.model_240.Model240</code>	<code>method)</code> , 55
<code>method)</code> , 255	<code>get_source_function()</code>
<code>get_sensor_name()</code> <code>(lakeshore.model_346.Model346</code>	<code>(lakeshore.ssm_source_module.SourceModule</code>
<code>method)</code> , 183	<code>method)</code> , 55
<code>get_sensor_reading()</code>	<code>get_source_module()</code>
<code>(lakeshore.model_224.Model224</code>	<code>(lakeshore.ssm_system.SSMSystem</code>
<code>method)</code> , 237	<code>method)</code> , 46
<code>get_sensor_reading()</code>	<code>get_source_module_by_name()</code>
<code>(lakeshore.model_240.Model240</code>	<code>(lakeshore.ssm_system.SSMSystem</code>
<code>method)</code> , 254	<code>method)</code> , 47
<code>get_sensor_reading()</code>	<code>get_source_pod()</code> <code>(lakeshore.ssm_system.SSMSystem</code>
<code>(lakeshore.model_346.Model346</code>	<code>method)</code> , 46
<code>method)</code> , 183	<code>get_source_sweep_state()</code>
<code>get_sensor_units_channel_reading()</code>	<code>(lakeshore.ssm_source_module.SourceModule</code>
<code>(lakeshore.model_240.Model240</code>	<code>method)</code> , 66
<code>method)</code> , 257	<code>get_source_sweep_step_size()</code>
<code>get_serial()</code> <code>(lakeshore.ssm_measure_module.MeasureModule</code>	<code>(lakeshore.ssm_source_module.SourceModule</code>
<code>method)</code> , 71	<code>method)</code> , 66
<code>get_serial()</code> <code>(lakeshore.ssm_source_module.SourceModule</code>	<code>get_stability_settings()</code>
<code>method)</code> , 55	<code>(lakeshore.model_346.Model346</code>
<code>get_service_request()</code>	<code>method)</code> , 159
<code>(lakeshore.model_224.Model224</code>	<code>get_standard_event_enable_mask()</code>
<code>method)</code> , 236	<code>(lakeshore.fast_hall_controller.FastHall</code>
<code>get_service_request()</code>	<code>method)</code> , 38
<code>(lakeshore.model_346.Model346</code>	<code>get_standard_event_enable_mask()</code>
<code>method)</code> , 183	<code>(lakeshore.model_155.PrecisionSource</code>
<code>get_service_request_enable_mask()</code>	<code>method)</code> , 268

`get_standard_event_enable_mask()` (*lakeshore.model_224.Model224* method), 236
`get_standard_event_enable_mask()` (*lakeshore.model_346.Model346* method), 184
`get_standard_event_enable_mask()` (*lakeshore.ssm_system.SSMSystem* method), 52
`get_standard_event_enable_mask()` (*lakeshore.teslameter.Teslameter* method), 17
`get_standard_event_status_enable_mask()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 28
`get_standard_event_status_event()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 29
`get_standard_events()` (*lakeshore.fast_hall_controller.FastHall* method), 39
`get_standard_events()` (*lakeshore.model_155.PrecisionSource* method), 268
`get_standard_events()` (*lakeshore.model_346.Model346* method), 184
`get_standard_events()` (*lakeshore.ssm_system.SSMSystem* method), 52
`get_standard_events()` (*lakeshore.teslameter.Teslameter* method), 17
`get_status_byte()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 28
`get_status_byte()` (*lakeshore.fast_hall_controller.FastHall* method), 39
`get_status_byte()` (*lakeshore.model_155.PrecisionSource* method), 268
`get_status_byte()` (*lakeshore.model_224.Model224* method), 236
`get_status_byte()` (*lakeshore.model_346.Model346* method), 184
`get_status_byte()` (*lakeshore.ssm_system.SSMSystem* method), 52
`get_status_byte()` (*lakeshore.teslameter.Teslameter* method), 17
`get_summary()` (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 87
`get_sweep_configuration()` (*lakeshore.ssm_source_module.SourceModule* method), 66
`get_sync_phase_shift()` (*lakeshore.ssm_source_module.SourceModule* method), 56
`get_sync_source()` (*lakeshore.ssm_source_module.SourceModule* method), 64
`get_sync_state()` (*lakeshore.ssm_source_module.SourceModule* method), 56
`get_temperature()` (*lakeshore.teslameter.Teslameter* method), 11
`get_temperature_compensation_manual_temp()` (*lakeshore.teslameter.Teslameter* method), 12
`get_temperature_compensation_source()` (*lakeshore.teslameter.Teslameter* method), 12
`get_temperature_limit()` (*lakeshore.model_346.Model346* method), 184
`get_thermocouple_compensation_offset()` (*lakeshore.model_346.Model346* method), 167
`get_thermocouple_junction_temp()` (*lakeshore.model_346.Model346* method), 165
`get_threshold()` (*lakeshore.model_346.Model346* method), 158
`get_unlocked_status()` (*lakeshore.ssm_measure_module.MeasureModule* method), 81
`get_valid_for_restore()` (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 88
`get_voltage_ac_range()` (*lakeshore.ssm_source_module.SourceModule* method), 61
`get_voltage_amplitude()` (*lakeshore.ssm_source_module.SourceModule* method), 62
`get_voltage_aurorange_status()` (*lakeshore.ssm_measure_module.MeasureModule* method), 75
`get_voltage_aurorange_status()` (*lakeshore.ssm_source_module.SourceModule* method), 62
`get_voltage_dc_range()` (*lakeshore.ssm_source_module.SourceModule* method), 62
`get_voltage_limit()` (*lakeshore.ssm_source_module.SourceModule* method), 63
`get_voltage_limit_status()` (*lakeshore.ssm_source_module.SourceModule* method), 63
`get_voltage_offset()` (*lakeshore.ssm_source_module.SourceModule* method), 62
`get_voltage_output_limit_high()` (*lakeshore.ssm_source_module.SourceModule* method), 64

`get_voltage_output_limit_low()` (*lakeshore.ssm_source_module.SourceModule* method), 64
`get_voltage_range()` (*lakeshore.ssm_measure_module.MeasureModule* method), 75
`get_voltage_range()` (*lakeshore.ssm_source_module.SourceModule* method), 61
`get_voltage_sense_mode()` (*lakeshore.ssm_measure_module.MeasureModule* method), 86
`get_voltage_sense_mode()` (*lakeshore.ssm_source_module.SourceModule* method), 68
`get_website_login()` (*lakeshore.model_224.Model224* method), 240
`go_to_current_mode()` (*lakeshore.ssm_source_module.SourceModule* method), 55
`go_to_voltage_mode()` (*lakeshore.ssm_source_module.SourceModule* method), 55
GREATER_THAN (*lakeshore.model_346.Model346Enums.ThresholdComparison* attribute), 193
H
HIGH (*lakeshore.model_240.Model240Enums.BrightnessLevel* attribute), 260
HIGH (*lakeshore.model_335.Model335Enums.HeaterRange* attribute), 119
HIGH (*lakeshore.model_336.Model336Enums.HeaterRange* attribute), 153
HIGH (*lakeshore.model_346.Model346Enums.HeaterRange* attribute), 191
HIGH (*lakeshore.model_346.Model346Enums.RelayDigitalInputCondition* attribute), 193
HIGH_ALARM (*lakeshore.model_224.Model224Enums.RelayControlAlarm* attribute), 250
HIGH_ALARM (*lakeshore.model_346.Model346Enums.RelayControlAlarm* attribute), 192
HIGH_ALARM_ACTIVE (*lakeshore.model_346.Model346Enums.RelayTripartiteCondition* attribute), 192
HUNDRED_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 118
HUNDRED_OHM (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 152
HUNDRED_THOUSAND_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange* attribute), 190
I
IEEE_488 (*lakeshore.model_224.Model224Enums.RemoteInterface* attribute), 249
imag (*lakeshore.model_346.Model346.AutotuneMode* attribute), 169
imag (*lakeshore.model_346.Model346.BrightnessLevel* attribute), 170
imag (*lakeshore.model_346.Model346.ControlTypes* attribute), 171
imag (*lakeshore.model_346.Model346.CurveFormat* attribute), 172
imag (*lakeshore.model_346.Model346.CurveTemperatureCoefficient* attribute), 172
imag (*lakeshore.model_346.Model346.DiodeCurrent* attribute), 173
imag (*lakeshore.model_346.Model346.DisplayFields* attribute), 174
imag (*lakeshore.model_346.Model346.DisplayFieldsSize* attribute), 175
imag (*lakeshore.model_346.Model346.HeaterError* attribute), 176
imag (*lakeshore.model_346.Model346.Interface* attribute), 177
imag (*lakeshore.model_346.Model346.InterfaceMode* attribute), 178
imag (*lakeshore.model_346.Model346.LanStatus* attribute), 179
initiate_sweeps() (*lakeshore.ssm_system.SSMSystem* method), 51
INPUT_A (*lakeshore.model_224.Model224Enums.DisplayMode* attribute), 249
INPUT_A (*lakeshore.model_224.Model224Enums.InputChannel* attribute), 249
INPUT_A (*lakeshore.model_335.Model335Enums.DisplaySetup* attribute), 120
INPUT_A (*lakeshore.model_335.Model335Enums.InputChannel* attribute), 120
INPUT_A (*lakeshore.model_336.Model336Enums.DisplaySetupMode* attribute), 151
INPUT_A_MAX_MIN (*lakeshore.model_335.Model335Enums.DisplaySetup* attribute), 120
INPUT_B (*lakeshore.model_224.Model224Enums.DisplayMode* attribute), 249
INPUT_B (*lakeshore.model_224.Model224Enums.InputChannel* attribute), 249
INPUT_B (*lakeshore.model_224.Model224Enums.InputChannel* attribute), 120
INPUT_B (*lakeshore.model_335.Model335Enums.DisplaySetup* attribute), 120
INPUT_B (*lakeshore.model_335.Model335Enums.InputChannel* attribute), 120
INPUT_B (*lakeshore.model_336.Model336Enums.DisplaySetupMode* attribute), 151
INPUT_B_MAX_MIN (*lakeshore.model_335.Model335Enums.DisplaySetup* attribute), 120
INPUT_C (*lakeshore.model_224.Model224Enums.DisplayMode* attribute), 249
INPUT_C (*lakeshore.model_224.Model224Enums.InputChannel* attribute), 249

lakeshore.model_224
 module, 235

lakeshore.model_240
 module, 253

lakeshore.model_335
 module, 93

lakeshore.model_336
 module, 122

lakeshore.model_346
 module, 157

lakeshore.model_350
 module, 194

lakeshore.model_372
 module, 198

lakeshore.model_425
 module, 20

lakeshore.ssm_measure_module
 module, 71

lakeshore.ssm_settings_profiles
 module, 87

lakeshore.ssm_source_module
 module, 54

lakeshore.ssm_system
 module, 46

lakeshore.teslameter
 module, 10

LARGE_4 (*lakeshore.model_224.Model224Enums.NumberOfFields*
 attribute), 120

LARGE_4_SMALL_8 (*lakeshore.model_224.Model224Enums.NumberOfFields*
 attribute), 250

LARGE_8 (*lakeshore.model_224.Model224Enums.NumberOfFields*
 attribute), 231

LESS_THAN (*lakeshore.model_346.Model346Enums.ThresholdComparison*
 attribute), 193

LEVEL_SENSE (*lakeshore.model_346.Model346Enums.OptionCardType*
 attribute), 193

load_modules() (*lakeshore.ssm_system.SSMSystem*
 method), 46

LOCAL (*lakeshore.model_224.Model224Enums.InterfaceMode*
 attribute), 248

lock_front_panel() (*lakeshore.model_121.Model121*
 method), 262

log_buffered_data_to_file()
 (*lakeshore.teslameter.Teslameter* method),
 11

log_data_to_csv_file()
 (*lakeshore.ssm_system.SSMSystem* method), 47

LOG_OHMS_PER_KELVIN
 (*lakeshore.model_224.Model224Enums.CurveFormat*
 attribute), 250

LOG_OHMS_PER_KELVIN
 (*lakeshore.model_240.Model240Enums.CurveFormat*
 attribute), 259

LOGOHM_PER_KELVIN (*lakeshore.model_372.Model372Enums.CurveFormat*
 attribute), 231

LOW (*lakeshore.model_240.Model240Enums.BrightnessLevel*
 attribute), 259

LOW (*lakeshore.model_335.Model335Enums.HeaterRange*
 attribute), 119

LOW (*lakeshore.model_336.Model336Enums.HeaterRange*
 attribute), 153

LOW (*lakeshore.model_346.Model346Enums.HeaterRange*
 attribute), 191

LOW (*lakeshore.model_346.Model346Enums.RelayDigitalInputCondition*
 attribute), 193

LOW_ALARM (*lakeshore.model_224.Model224Enums.RelayControlAlarm*
 attribute), 250

LOW_ALARM (*lakeshore.model_346.Model346Enums.RelayControlAlarm*
 attribute), 192

LOW_ALARM_ACTIVE (*lakeshore.model_346.Model346Enums.RelayThermom*
 attribute), 192

LOW_AND_HIGH_ALARM (*lakeshore.model_346.Model346Enums.RelayContr*
 attribute), 192

LOW_OR_HIGH_ALARM (*lakeshore.model_346.Model346Enums.RelayContro*
 attribute), 192

M

MAXIMUM_DATA (*lakeshore.model_224.Model224Enums.DisplayFieldUnits*
 attribute), 249

MAXIMUM_DATA (*lakeshore.model_335.Model335Enums.DisplayFieldUnits*
 attribute), 153

MAXIMUM_DATA (*lakeshore.model_336.Model336Enums.DisplayFieldUnits*
 attribute), 153

MAXIMUM_DATA (*lakeshore.model_372.Model372Enums.DisplayFieldUnits*
 attribute), 89

MEASURE_DC (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMe*
 attribute), 89

MEASURE_NEGATIVE_PEAK
 (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMne*
 attribute), 89

MEASURE_OVERLOAD (*lakeshore.ssm_system_enums.SSMSystemEnums.Data*
 attribute), 89

MEASURE_PEAK_TO_PEAK
 (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMne*
 attribute), 89

MEASURE_POSITIVE_PEAK
 (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMne*
 attribute), 89

MEASURE_R (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceM*
 attribute), 89

MEASURE_RANGE (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSou*
 attribute), 89

MEASURE_REFERENCE_FREQUENCY
 (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSourceMne*
 attribute), 90

MEASURE_RMS (*lakeshore.ssm_system_enums.SSMSystemEnums.DataSource*
 attribute), 89

MEASURE_SETTLING (<i>lakeshore.ssm_system_enums.SSMSystemEnums</i> attribute), 90	<i>Model224Enums.InputSensorType</i> (class in <i>lakeshore.model_224</i>), 247
MEASURE_THETA (<i>lakeshore.ssm_system_enums.SSMSystemEnums</i> attribute), 89	<i>Model224Enums.InputSensorUnits</i> (class in <i>lakeshore.model_224</i>), 247
MEASURE_UNLOCK (<i>lakeshore.ssm_system_enums.SSMSystemEnums</i> attribute), 90	<i>Model224Enums.RemoteInterfaceMode</i> (class in <i>lakeshore.model_224</i>), 248
MEASURE_X (<i>lakeshore.ssm_system_enums.SSMSystemEnums</i> attribute), 89	<i>Model224Enums.NTCRTDSensorResistanceRange</i> (class in <i>lakeshore.model_224</i>), 248
MEASURE_Y (<i>lakeshore.ssm_system_enums.SSMSystemEnums</i> attribute), 89	<i>Model224Enums.NumberOfFields</i> (class in <i>lakeshore.model_224</i>), 250
MEASUREMENT_INPUT (<i>lakeshore.model_372.Model372Enums</i> attribute), 230	<i>Model224Enums.PlatinumRTDSensorResistanceRange</i> (class in <i>lakeshore.model_224</i>), 248
MeasureModule (class in <i>lakeshore.ssm_measure_module</i>), 71	<i>Model224Enums.RelayControlAlarm</i> (class in <i>lakeshore.model_224</i>), 250
MED_HIGH (<i>lakeshore.model_240.Model240Enums.BrightnessLevel</i> attribute), 260	<i>Model224Enums.RelayControlMode</i> (class in <i>lakeshore.model_224</i>), 250
MED_LOW (<i>lakeshore.model_240.Model240Enums.BrightnessLevel</i> attribute), 260	<i>Model224Enums.RemoteInterface</i> (class in <i>lakeshore.model_224</i>), 248
MEDIUM (<i>lakeshore.model_335.Model335Enums.HeaterRange</i> attribute), 119	<i>Model224Enums.SoftCalSensorTypes</i> (class in <i>lakeshore.model_224</i>), 251
MEDIUM (<i>lakeshore.model_336.Model336Enums.HeaterRange</i> attribute), 153	<i>Model224InputSensorSettings</i> (class in <i>lakeshore.model_224</i>), 247
MILLIVOLT_PER_KELVIN (<i>lakeshore.model_224.Model224Enums.CurveFormat</i> attribute), 250	<i>Model224ReadingStatusRegister</i> (class in <i>lakeshore.model_224</i>), 251
MINIMUM_DATA (<i>lakeshore.model_224.Model224Enums.DisplayFieldUnits</i> attribute), 249	<i>Model224ServiceRequestRegister</i> (class in <i>lakeshore.model_224</i>), 251
MINIMUM_DATA (<i>lakeshore.model_335.Model335Enums.DisplayFieldUnits</i> attribute), 120	<i>Model224StandardEventRegister</i> (in module <i>lakeshore.model_224</i>), 251
MINIMUM_DATA (<i>lakeshore.model_336.Model336Enums.DisplayFieldUnits</i> attribute), 153	<i>Model224StatusByteRegister</i> (class in <i>lakeshore.model_224</i>), 251
MINIMUM_DATA (<i>lakeshore.model_372.Model372Enums.DisplayFieldUnits</i> attribute), 231	<i>Model240</i> (class in <i>lakeshore.model_240</i>), 253
<i>Model121</i> (class in <i>lakeshore.model_121</i>), 261	<i>Model240CurveHeader</i> (class in <i>lakeshore.model_240</i>), 258
<i>Model224</i> (class in <i>lakeshore.model_224</i>), 235	<i>Model240Enums</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224AlarmParameters</i> (class in <i>lakeshore.model_224</i>), 247	<i>Model240Enums.BrightnessLevel</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224CurveHeader</i> (class in <i>lakeshore.model_224</i>), 247	<i>Model240Enums.Coefficients</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224Enums</i> (class in <i>lakeshore.model_224</i>), 247	<i>Model240Enums.CurveFormat</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224Enums.CurveFormat</i> (class in <i>lakeshore.model_224</i>), 250	<i>Model240Enums.InputRange</i> (class in <i>lakeshore.model_240</i>), 260
<i>Model224Enums.CurveTemperatureCoefficients</i> (class in <i>lakeshore.model_224</i>), 251	<i>Model240Enums.SensorTypes</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224Enums.DiodeExcitationCurrent</i> (class in <i>lakeshore.model_224</i>), 251	<i>Model240Enums.TemperatureCoefficient</i> (class in <i>lakeshore.model_240</i>), 260
<i>Model224Enums.DiodeSensorRange</i> (class in <i>lakeshore.model_224</i>), 248	<i>Model240Enums.Units</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224Enums.DisplayFieldUnits</i> (class in <i>lakeshore.model_224</i>), 249	<i>Model240InputParameter</i> (class in <i>lakeshore.model_240</i>), 258
<i>Model224Enums.DisplayMode</i> (class in <i>lakeshore.model_224</i>), 249	<i>Model240ProfiSlot</i> (class in <i>lakeshore.model_240</i>), 259
<i>Model224Enums.InputChannel</i> (class in <i>lakeshore.model_224</i>), 249	<i>Model335ControlLoopZoneSettings</i> (class in <i>lakeshore.model_335</i>), 114

Model335Enums (class in lakeshore.model_335), 118	Model336Enums.DisplaySetupMode (class in lakeshore.model_336), 151
Model335Enums.DiodeRange (class in lakeshore.model_335), 118	Model336Enums.HeaterOutputMode (class in lakeshore.model_336), 153
Model335Enums.DisplayFieldUnits (class in lakeshore.model_335), 120	Model336Enums.HeaterRange (class in lakeshore.model_336), 153
Model335Enums.DisplaySetup (class in lakeshore.model_335), 120	Model336Enums.HeaterVoltageRange (class in lakeshore.model_336), 153
Model335Enums.HeaterOutputDisplay (class in lakeshore.model_335), 119	Model336Enums.InputChannel (class in lakeshore.model_336), 151
Model335Enums.HeaterOutputMode (class in lakeshore.model_335), 119	Model336Enums.InputSensorType (class in lakeshore.model_336), 152
Model335Enums.HeaterOutType (class in lakeshore.model_335), 119	Model336Enums.RTDRange (class in lakeshore.model_336), 152
Model335Enums.HeaterRange (class in lakeshore.model_335), 119	Model336Enums.ThermocoupleRange (class in lakeshore.model_336), 153
Model335Enums.HeaterVoltageRange (class in lakeshore.model_335), 120	Model336InputReadingStatus (class in lakeshore.model_336), 148
Model335Enums.InputChannel (class in lakeshore.model_335), 120	Model336InputSensorSettings (class in lakeshore.model_336), 146
Model335Enums.InputSensor (class in lakeshore.model_335), 118	Model336OperationEvent (in module lakeshore.model_336), 148
Model335Enums.InputSensorType (class in lakeshore.model_335), 118	Model336ServiceRequestEnable (class in lakeshore.model_336), 148
Model335Enums.MonitorOutUnits (class in lakeshore.model_335), 118	Model336StandardEventRegister (in module lakeshore.model_336), 148
Model335Enums.RTDRange (class in lakeshore.model_335), 118	Model336StatusByteRegister (class in lakeshore.model_336), 148
Model335Enums.ThermocoupleRange (class in lakeshore.model_335), 119	Model346 (class in lakeshore.model_346), 157
Model335Enums.WarmupControl (class in lakeshore.model_335), 119	Model346.AutotuneMode (class in lakeshore.model_346), 168
Model335InputReadingStatus (class in lakeshore.model_335), 115	Model346.BrightnessLevel (class in lakeshore.model_346), 169
Model335InputSensorSettings (class in lakeshore.model_335), 114	Model346.ControlTypes (class in lakeshore.model_346), 170
Model335OperationEvent (in module lakeshore.model_335), 115	Model346.CurveFormat (class in lakeshore.model_346), 171
Model335ServiceRequestEnable (class in lakeshore.model_335), 115	Model346.CurveTemperatureCoefficient (class in lakeshore.model_346), 172
Model335StandardEventRegister (in module lakeshore.model_335), 115	Model346.DiodeCurrent (class in lakeshore.model_346), 173
Model335StatusByteRegister (class in lakeshore.model_335), 115	Model346.DisplayFields (class in lakeshore.model_346), 174
Model336AlarmSettings (in module lakeshore.model_336), 147	Model346.DisplayFieldsSize (class in lakeshore.model_346), 174
Model336ControlLoopZoneSettings (class in lakeshore.model_336), 146	Model346.HeaterError (class in lakeshore.model_346), 175
Model336CurveHeader (in module lakeshore.model_336), 147	Model346.Interface (class in lakeshore.model_346), 176
Model336Enums (class in lakeshore.model_336), 151	Model346.InterfaceMode (class in lakeshore.model_346), 177
Model336Enums.DiodeRange (class in lakeshore.model_336), 152	Model346.LanStatus (class in lakeshore.model_346), 178
Model336Enums.DisplayFieldUnits (class in lakeshore.model_336), 153	Model346AlarmSettings (class in lakeshore.model_346), 178

<i>lakeshore.model_346</i>), 189	Model350 (<i>class in lakeshore.model_350</i>), 194
Model346CurveHeader (<i>in module lakeshore.model_346</i>), 189	Model372AlarmParameters (<i>class in lakeshore.model_372</i>), 225
Model346Enums (<i>class in lakeshore.model_346</i>), 190	Model372ControlLoopZoneSettings (<i>class in lakeshore.model_372</i>), 224
Model346Enums.DiodeRange (<i>class in lakeshore.model_346</i>), 190	Model372CurveHeader (<i>in module lakeshore.model_372</i>), 225
Model346Enums.HeaterAutorange (<i>class in lakeshore.model_346</i>), 191	Model372Enums (<i>class in lakeshore.model_372</i>), 228
Model346Enums.HeaterOutputUnits (<i>class in lakeshore.model_346</i>), 191	Model372Enums.AutoRangeMode (<i>class in lakeshore.model_372</i>), 230
Model346Enums.HeaterRange (<i>class in lakeshore.model_346</i>), 191	Model372Enums.ControlInputCurrentRange (<i>class in lakeshore.model_372</i>), 233
Model346Enums.HeaterRangeAnalogOutput (<i>class in lakeshore.model_346</i>), 191	Model372Enums.CurveFormat (<i>class in lakeshore.model_372</i>), 231
Model346Enums.InputSensorType (<i>class in lakeshore.model_346</i>), 190	Model372Enums.DisplayFieldUnits (<i>class in lakeshore.model_372</i>), 231
Model346Enums.InputSensorUnits (<i>class in lakeshore.model_346</i>), 190	Model372Enums.DisplayInfo (<i>class in lakeshore.model_372</i>), 231
Model346Enums.NTCRTDRange (<i>class in lakeshore.model_346</i>), 190	Model372Enums.DisplayMode (<i>class in lakeshore.model_372</i>), 230
Model346Enums.OptionCardType (<i>class in lakeshore.model_346</i>), 193	Model372Enums.HeaterOutput (<i>class in lakeshore.model_372</i>), 234
Model346Enums.OutputMode (<i>class in lakeshore.model_346</i>), 191	Model372Enums.InputChannel (<i>class in lakeshore.model_372</i>), 229
Model346Enums.PTCRTDRange (<i>class in lakeshore.model_346</i>), 190	Model372Enums.InputFrequency (<i>class in lakeshore.model_372</i>), 231
Model346Enums.RelayControlAlarm (<i>class in lakeshore.model_346</i>), 192	Model372Enums.InputSensorUnits (<i>class in lakeshore.model_372</i>), 230
Model346Enums.RelayControlMode (<i>class in lakeshore.model_346</i>), 192	Model372Enums.MeasurementInputCurrentRange (<i>class in lakeshore.model_372</i>), 232
Model346Enums.RelayDigitalInputCondition (<i>class in lakeshore.model_346</i>), 193	Model372Enums.MeasurementInputResistance (<i>class in lakeshore.model_372</i>), 233
Model346Enums.RelayFunctionOutputCondition (<i>class in lakeshore.model_346</i>), 193	Model372Enums.MeasurementInputVoltageRange (<i>class in lakeshore.model_372</i>), 232
Model346Enums.RelayOutputStatusCondition (<i>class in lakeshore.model_346</i>), 192	Model372Enums.MonitorOutputSource (<i>class in lakeshore.model_372</i>), 230
Model346Enums.RelayThermometryCondition (<i>class in lakeshore.model_346</i>), 192	Model372Enums.OutputMode (<i>class in lakeshore.model_372</i>), 229
Model346Enums.StandardCurveType (<i>class in lakeshore.model_346</i>), 191	Model372Enums.RelayControlMode (<i>class in lakeshore.model_372</i>), 230
Model346Enums.ThermocoupleRange (<i>class in lakeshore.model_346</i>), 190	Model372Enums.SampleHeaterOutputRange (<i>class in lakeshore.model_372</i>), 231
Model346Enums.ThresholdComparison (<i>class in lakeshore.model_346</i>), 193	Model372Enums.SensorExcitationMode (<i>class in lakeshore.model_372</i>), 229
Model346InputSensorSettings (<i>class in lakeshore.model_346</i>), 188	Model372HeaterOutputSettings (<i>class in lakeshore.model_372</i>), 223
Model346OperationRegister (<i>class in lakeshore.model_346</i>), 194	Model372InputChannelSettings (<i>class in lakeshore.model_372</i>), 223
Model346QuestionableRegister (<i>class in lakeshore.model_346</i>), 194	Model372InputSetupSettings (<i>class in lakeshore.model_372</i>), 223
Model346StandardEventRegister (<i>in module lakeshore.model_346</i>), 193	Model372ReadingStatusRegister (<i>class in lakeshore.model_372</i>), 225
Model346StatusByteRegister (<i>in module lakeshore.model_346</i>), 193	Model372ServiceRequestEnable (<i>class in lakeshore.model_372</i>), 226

Model372StandardEventRegister (*in module* *lakeshore.model_372*), 225
lakeshore.model_372), 225
Model372StatusByteRegister (*class in* *lakeshore.model_372*), 226
Model425 (*class in* *lakeshore.model_425*), 20
modify_operation_register_mask() (*lakeshore.fast_hall_controller.FastHall* *method*), 39
modify_operation_register_mask() (*lakeshore.model_155.PrecisionSource* *method*), 268
modify_operation_register_mask() (*lakeshore.model_346.Model346* *method*), 184
modify_operation_register_mask() (*lakeshore.ssm_system.SSMSystem* *method*), 52
modify_operation_register_mask() (*lakeshore.teslameter.Teslameter* *method*), 17
modify_questionable_register_mask() (*lakeshore.fast_hall_controller.FastHall* *method*), 39
modify_questionable_register_mask() (*lakeshore.model_155.PrecisionSource* *method*), 268
modify_questionable_register_mask() (*lakeshore.model_346.Model346* *method*), 184
modify_questionable_register_mask() (*lakeshore.ssm_system.SSMSystem* *method*), 52
modify_questionable_register_mask() (*lakeshore.teslameter.Teslameter* *method*), 17
modify_service_request_mask() (*lakeshore.fast_hall_controller.FastHall* *method*), 39
modify_service_request_mask() (*lakeshore.model_155.PrecisionSource* *method*), 268
modify_service_request_mask() (*lakeshore.model_346.Model346* *method*), 185
modify_service_request_mask() (*lakeshore.ssm_system.SSMSystem* *method*), 53
modify_service_request_mask() (*lakeshore.teslameter.Teslameter* *method*), 17
modify_standard_event_register_mask() (*lakeshore.fast_hall_controller.FastHall* *method*), 39
modify_standard_event_register_mask() (*lakeshore.model_155.PrecisionSource* *method*), 268
modify_standard_event_register_mask() (*lakeshore.model_346.Model346* *method*), 185
modify_standard_event_register_mask() (*lakeshore.ssm_system.SSMSystem* *method*), 53
modify_standard_event_register_mask() (*lakeshore.teslameter.Teslameter* *method*), 17
module
lakeshore.em_power_supply, 22
lakeshore.fast_hall_controller, 34
lakeshore.model_121, 261
lakeshore.model_155, 264
lakeshore.model_224, 235
lakeshore.model_240, 253
lakeshore.model_335, 93
lakeshore.model_336, 122
lakeshore.model_346, 157
lakeshore.model_350, 194
lakeshore.model_372, 198
lakeshore.model_425, 20
lakeshore.ssm_measure_module, 71
lakeshore.ssm_settings_profiles, 87
lakeshore.ssm_source_module, 54
lakeshore.ssm_system, 46
lakeshore.teslameter, 10
MONITOR_OUT (*lakeshore.model_335.Model335Enums.HeaterOutputMode* *attribute*), 119
MONITOR_OUT (*lakeshore.model_336.Model336Enums.HeaterOutputMode* *attribute*), 153
MONITOR_OUT (*lakeshore.model_346.Model346Enums.OutputMode* *attribute*), 191
MONITOR_OUT (*lakeshore.model_372.Model372Enums.OutputMode* *attribute*), 229
MOTOR_CONTROL (*lakeshore.model_346.Model346Enums.OptionCardType* *attribute*), 193
N
NEGATIVE (*lakeshore.model_224.Model224Enums.CurveTemperatureCoefficient* *attribute*), 251
NEGATIVE (*lakeshore.model_240.Model240Enums.Coefficients* *attribute*), 259
NEGATIVE (*lakeshore.model_240.Model240Enums.TemperatureCoefficient* *attribute*), 260
NINE (*lakeshore.model_372.Model372Enums.InputChannel* *attribute*), 229
NO_INPUT (*lakeshore.model_224.Model224Enums.InputChannel* *attribute*), 249
NONE (*lakeshore.model_335.Model335Enums.InputChannel* *attribute*), 120
NONE (*lakeshore.model_335.Model335Enums.InputSensor* *attribute*), 118
NONE (*lakeshore.model_336.Model336Enums.InputChannel* *attribute*), 151

NONE (*lakeshore.model_346.Model346Enums.OptionCardType* attribute), 193

NONE (*lakeshore.model_372.Model372Enums.DisplayInfo* attribute), 231

NONE (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229

NTC_RTD (*lakeshore.model_224.Model224Enums.InputSensorType* attribute), 247

NTC_RTD (*lakeshore.model_240.Model240Enums.SensorType* attribute), 259

NTC_RTD (*lakeshore.model_335.Model335Enums.InputSensorType* attribute), 118

NTC_RTD (*lakeshore.model_336.Model336Enums.InputSensorType* attribute), 152

NTC_RTD (*lakeshore.model_346.Model346Enums.InputSensorType* attribute), 190

NTC_RX102A_AA (*lakeshore.model_346.Model346Enums.StimulusType* attribute), 191

NTC_RX103A_AA (*lakeshore.model_346.Model346Enums.StimulusType* attribute), 191

NTC_RX202A_AA (*lakeshore.model_346.Model346Enums.StimulusType* attribute), 191

numerator (*lakeshore.model_346.Model346.AutotuneMode* attribute), 169

numerator (*lakeshore.model_346.Model346.BrightnessLevel* attribute), 170

numerator (*lakeshore.model_346.Model346.ControlTypes* attribute), 171

numerator (*lakeshore.model_346.Model346.CurveFormat* attribute), 172

numerator (*lakeshore.model_346.Model346.CurveTemperatureCoefficient* attribute), 173

numerator (*lakeshore.model_346.Model346.DiodeCurrent* attribute), 173

numerator (*lakeshore.model_346.Model346.DisplayFields* attribute), 174

numerator (*lakeshore.model_346.Model346.DisplayFieldsSize* attribute), 175

numerator (*lakeshore.model_346.Model346.HeaterError* attribute), 176

numerator (*lakeshore.model_346.Model346.Interface* attribute), 177

numerator (*lakeshore.model_346.Model346.InterfaceMode* attribute), 178

numerator (*lakeshore.model_346.Model346.LanStatus* attribute), 179

OFF (*lakeshore.model_240.Model240Enums.BrightnessLevel* attribute), 259

OFF (*lakeshore.model_335.Model335Enums.HeaterOutputMode* attribute), 119

OFF (*lakeshore.model_335.Model335Enums.HeaterRange* attribute), 119

OFF (*lakeshore.model_336.Model336Enums.HeaterOutputMode* attribute), 153

OFF (*lakeshore.model_336.Model336Enums.HeaterRange* attribute), 153

OFF (*lakeshore.model_346.Model346Enums.HeaterRange* attribute), 191

OFF (*lakeshore.model_346.Model346Enums.HeaterRangeAnalogOutput* attribute), 191

OFF (*lakeshore.model_346.Model346Enums.OutputMode* attribute), 191

OFF (*lakeshore.model_346.Model346Enums.RelayControlMode* attribute), 192

OFF (*lakeshore.model_372.Model372Enums.AutoRangeMode* attribute), 230

OFF (*lakeshore.model_372.Model372Enums.MonitorOutputSource* attribute), 230

OFF (*lakeshore.model_372.Model372Enums.OutputMode* attribute), 229

OFF (*lakeshore.model_372.Model372Enums.SampleHeaterOutputRange* attribute), 231

OHM_PER_KELVIN (*lakeshore.model_372.Model372Enums.CurveFormat* attribute), 231

OHM_PER_KELVIN_CUBIC_SPLINE (*lakeshore.model_372.Model372Enums.CurveFormat* attribute), 231

OHMS (*lakeshore.model_372.Model372Enums.DisplayFieldUnits* attribute), 231

OHMS (*lakeshore.model_372.Model372Enums.InputSensorUnits* attribute), 230

OHMS_PER_KELVIN (*lakeshore.model_224.Model224Enums.CurveFormat* attribute), 250

OHMS_PER_KELVIN (*lakeshore.model_240.Model240Enums.CurveFormat* attribute), 259

ON (*lakeshore.model_346.Model346Enums.HeaterRangeAnalogOutput* attribute), 191

ON (*lakeshore.model_346.Model346Enums.RelayControlMode* attribute), 192

ONE (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 229

ONE_HUNDRED_KILOHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistance* attribute), 248

ONE_HUNDRED_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange* attribute), 190

ONE_HUNDRED_OHM (*lakeshore.model_346.Model346Enums.PTCRTDRange* attribute), 190

ONE_HUNDRED_OHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistance* attribute), 248

ONE_HUNDRED_OHMS (*lakeshore.model_224.Model224Enums.PlatinumRTDResistance* attribute), 248

ONE_HUNDRED_THOUSAND_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 119

ONE_HUNDRED_THOUSAND_OHM

- (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 193
 - (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 153
 - ONE_KILOHM (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* attribute), 248
 - ONE_KILOHM (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* attribute), 248
 - ONE_MILLI_AMP (*lakeshore.model_224.Model224Enums.DiodeExcitationCurrent* attribute), 251
 - ONE_THOUSAND_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 118
 - ONE_THOUSAND_OHM (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 152
 - ONE_THOUSAND_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange* attribute), 190
 - ONE_THOUSAND_OHM (*lakeshore.model_346.Model346Enums.PTCRTDRange* attribute), 190
 - OPEN_LOOP (*lakeshore.model_335.Model335Enums.HeaterOutputMode* attribute), 119
 - OPEN_LOOP (*lakeshore.model_336.Model336Enums.HeaterOutputMode* attribute), 153
 - OPEN_LOOP (*lakeshore.model_346.Model346Enums.OutputMode* attribute), 191
 - OPEN_LOOP (*lakeshore.model_372.Model372Enums.OutputMode* attribute), 229
 - OperationEvent (class in *lakeshore.temperature_controllers*), 115
 - OUTPUT_1 (*lakeshore.model_335.Model335Enums.InputChannel* attribute), 120
 - OUTPUT_2 (*lakeshore.model_335.Model335Enums.InputChannel* attribute), 120
 - output_dc_current() (*lakeshore.model_155.PrecisionSource* method), 265
 - output_dc_voltage() (*lakeshore.model_155.PrecisionSource* method), 265
 - OUTPUT_LOAD_FAULT (*lakeshore.model_346.Model346Enums.RelayOutputStatusCondition* attribute), 192
 - OUTPUT_SETPOINT_RAMPING_COMPLETE (*lakeshore.model_346.Model346Enums.RelayOutputStatusCondition* attribute), 193
 - output_sine_current() (*lakeshore.model_155.PrecisionSource* method), 265
 - output_sine_voltage() (*lakeshore.model_155.PrecisionSource* method), 265
 - OUTPUT_STABILIZING (*lakeshore.model_346.Model346Enums.RelayOutputStatusCondition* attribute), 193
 - OUTPUT_STABLE (*lakeshore.model_346.Model346Enums.RelayOutputStatusCondition* attribute), 193
 - OUTPUT_STATUS (*lakeshore.model_346.Model346Enums.RelayControlMode* attribute), 192
 - OUTPUT_WARM (*lakeshore.model_346.Model346Enums.RelayOutputStatusCondition* attribute), 193
 - PID (*lakeshore.model_346.Model346Enums.OutputMode* attribute), 260
 - PLATINUM_RTD (*lakeshore.model_224.Model224Enums.InputSensorType* attribute), 247
 - PLATINUM_RTD (*lakeshore.model_240.Model240Enums.SensorTypes* attribute), 259
 - PLATINUM_RTD (*lakeshore.model_335.Model335Enums.InputSensorType* attribute), 118
 - PLATINUM_RTD (*lakeshore.model_336.Model336Enums.InputSensorType* attribute), 152
 - PLATINUM_RTD (*lakeshore.model_346.Model346Enums.InputSensorType* attribute), 190
 - POSITIVE (*lakeshore.model_224.Model224Enums.CurveTemperatureCoefficient* attribute), 251
 - POSITIVE (*lakeshore.model_240.Model240Enums.Coefficients* attribute), 260
 - POSITIVE (*lakeshore.model_240.Model240Enums.TemperatureCoefficient* attribute), 260
 - POWER (*lakeshore.model_335.Model335Enums.HeaterOutputDisplay* attribute), 119
 - POWER (*lakeshore.model_346.Model346Enums.HeaterOutputUnits* attribute), 191
 - PrecisionSource (class in *lakeshore.model_155*), 264
 - PTC_PT100 (*lakeshore.model_224.Model224Enums.SoftCalSensorTypes* attribute), 251
 - PTC_PT1000 (*lakeshore.model_224.Model224Enums.SoftCalSensorTypes* attribute), 251
 - PTC_PT100 (*lakeshore.model_346.Model346Enums.StandardCurveType* attribute), 191
- ## Q
- QUADRATURE (*lakeshore.model_372.Model372Enums.DisplayFieldUnits* attribute), 231
 - query() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24
 - query() (*lakeshore.fast_hall_controller.FastHall* method), 39
 - query() (*lakeshore.model_121.Model121* method), 262
 - query() (*lakeshore.model_155.PrecisionSource* method), 269
 - query() (*lakeshore.model_224.Model224* method), 235
 - query() (*lakeshore.model_240.Model240* method), 258
 - query() (*lakeshore.model_346.Model346* method), 185
 - query() (*lakeshore.model_350.Model350* method), 195
 - query() (*lakeshore.model_425.Model425* method), 20
 - query() (*lakeshore.ssm_system.SSMSystem* method), 53
 - query() (*lakeshore.testmeter.Teslameter* method), 18

RANGE_NTCRTD_3_KIL_OHMS (*lakeshore.model_240.Model240Enums.InputRange* attribute), 260 (*lakeshore.model_346.Model346* method), 164
reset_contact_check_measurement() (*lakeshore.fast_hall_controller.FastHall* method), 37
RANGE_PTRTD_1_KIL_OHMS (*lakeshore.model_240.Model240Enums.InputRange* attribute), 260
reset_control_setpoint()
read_multiple() (*lakeshore.ssm_measure_module.MeasureModule* method), 86 (*lakeshore.model_346.Model346* method), 166
read_multiple() (*lakeshore.ssm_system.SSMSystem* method), 50
reset_dc_hall_measurement() (*lakeshore.fast_hall_controller.FastHall* method), 37
real (*lakeshore.model_346.Model346.AutotuneMode* attribute), 169
reset_fasthall_measurement() (*lakeshore.fast_hall_controller.FastHall* method), 37
real (*lakeshore.model_346.Model346.BrightnessLevel* attribute), 170
reset_four_wire_measurement() (*lakeshore.fast_hall_controller.FastHall* method), 37
real (*lakeshore.model_346.Model346.ControlTypes* attribute), 171
reset_head_self_calibration()
real (*lakeshore.model_346.Model346.CurveFormat* attribute), 172
reset_head_self_calibration() (*lakeshore.ssm_system.SSMSystem* method), 50
real (*lakeshore.model_346.Model346.CurveTemperatureCoefficient* attribute), 172
reset_instrument() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27
real (*lakeshore.model_346.Model346.DiodeCurrent* attribute), 173
reset_instrument() (*lakeshore.model_121.Model121* method), 261
real (*lakeshore.model_346.Model346.DisplayFields* attribute), 174
reset_instrument() (*lakeshore.model_224.Model224* method), 236
real (*lakeshore.model_346.Model346.DisplayFieldsSize* attribute), 175
reset_instrument() (*lakeshore.model_346.Model346* method), 185
real (*lakeshore.model_346.Model346.HeaterError* attribute), 176
reset_max_min() (*lakeshore.teslameter.Teslameter* method), 11
real (*lakeshore.model_346.Model346.Interface* attribute), 177
reset_measurement_settings()
real (*lakeshore.model_346.Model346.InterfaceMode* attribute), 178
reset_measurement_settings() (*lakeshore.fast_hall_controller.FastHall* method), 40
real (*lakeshore.model_346.Model346.LanStatus* attribute), 179
reset_measurement_settings() (*lakeshore.model_155.PrecisionSource* method), 162
RELATIVE_TIME (*lakeshore.ssm_system_enums.SSMSystemEnums.DonutShadeMode* attribute), 89
reset_measurement_settings()
RELAY_OFF (*lakeshore.model_224.Model224Enums.RelayControlMode* attribute), 250 (*lakeshore.model_346.Model346* method), 185
RELAY_OFF (*lakeshore.model_372.Model372Enums.RelayControlMode* attribute), 230
reset_measurement_settings() (*lakeshore.ssm_system.SSMSystem* method), 53
RELAY_ON (*lakeshore.model_224.Model224Enums.RelayControlMode* attribute), 250
reset_measurement_settings() (*lakeshore.teslameter.Teslameter* method),
RELAY_ON (*lakeshore.model_372.Model372Enums.RelayControlMode* attribute), 230
reset_min_max_data()
REMOTE (*lakeshore.model_224.Model224Enums.InterfaceMode* attribute), 248 (*lakeshore.model_224.Model224* method), 239
REMOTE_LOCAL_LOCK (*lakeshore.model_224.Model224Enums.InterfaceMode* attribute), 248
reset_min_max_data() (*lakeshore.model_346.Model346* method),
rename() (*lakeshore.ssm_settings_profiles.SettingsProfiles* method), 87 164
reset_alarm_status() (*lakeshore.teslameter.Teslameter* method), 15
(*lakeshore.model_224.Model224* method), 241
reset_alarm_status() (*lakeshore.fast_hall_controller.FastHall* method), 15
reset_resistivity_measurement() (*lakeshore.fast_hall_controller.FastHall* method), 15

method), 37
 reset_self_cal() (*lakeshore.ssm_measure_module.MeasureModule* *method*), 72
 reset_self_cal() (*lakeshore.ssm_source_module.SourceModule* *method*), 55
 reset_settings() (*lakeshore.ssm_measure_module.MeasureModule* *method*), 86
 reset_settings() (*lakeshore.ssm_source_module.SourceModule* *method*), 65
 reset_status_register_masks() (*lakeshore.fast_hall_controller.FastHall* *method*), 40
 reset_status_register_masks() (*lakeshore.model_155.PrecisionSource* *method*), 269
 reset_status_register_masks() (*lakeshore.model_346.Model346* *method*), 185
 reset_status_register_masks() (*lakeshore.ssm_system.SSMSystem* *method*), 53
 reset_status_register_masks() (*lakeshore.teslameter.Teslameter* *method*), 18
 reset_sweeps() (*lakeshore.ssm_system.SSMSystem* *method*), 51
 ResistivityLinkParameters (*class* in *lakeshore.fast_hall_controller*), 43
 ResistivityManualParameters (*class* in *lakeshore.fast_hall_controller*), 42
 restore() (*lakeshore.ssm_settings_profiles.SettingsProfiles* *method*), 88
 route_terminals() (*lakeshore.model_155.PrecisionSource* *method*), 265
 ROX102B (*lakeshore.model_372.Model372Enums.AutoRangeMode* *attribute*), 230
 run_complete_contact_check_manual() (*lakeshore.fast_hall_controller.FastHall* *method*), 36
 run_complete_contact_check_optimized() (*lakeshore.fast_hall_controller.FastHall* *method*), 36
 run_complete_dc_hall() (*lakeshore.fast_hall_controller.FastHall* *method*), 37
 run_complete_fasthall_link() (*lakeshore.fast_hall_controller.FastHall* *method*), 36
 run_complete_fasthall_manual() (*lakeshore.fast_hall_controller.FastHall* *method*), 36
 run_complete_four_wire() (*lakeshore.fast_hall_controller.FastHall* *method*), 37
 run_complete_resistivity_link() (*lakeshore.fast_hall_controller.FastHall* *method*), 37
 run_complete_resistivity_manual() (*lakeshore.fast_hall_controller.FastHall* *method*), 37
 run_head_self_calibration() (*lakeshore.ssm_system.SSMSystem* *method*), 50
 run_self_cal() (*lakeshore.ssm_measure_module.MeasureModule* *method*), 72
 run_self_cal() (*lakeshore.ssm_source_module.SourceModule* *method*), 55

S

SAMPLE_HEATER (*lakeshore.model_372.Model372Enums.DisplayInfo* *attribute*), 231
 SAMPLE_HEATER_ZONE (*lakeshore.model_372.Model372Enums.RelayControl* *attribute*), 230
 save_current_state() (*lakeshore.model_121.Model121* *method*), 262
 SCANNER (*lakeshore.model_346.Model346Enums.OptionCardType* *attribute*), 193
 select_interface_mode() (*lakeshore.model_224.Model224* *method*), 244
 select_remote_interface() (*lakeshore.model_224.Model224* *method*), 244
 SENSOR (*lakeshore.model_224.Model224Enums.DisplayFieldUnits* *attribute*), 249
 SENSOR (*lakeshore.model_224.Model224Enums.InputSensorUnits* *attribute*), 248
 SENSOR (*lakeshore.model_240.Model240Enums.Units* *attribute*), 259
 SENSOR (*lakeshore.model_335.Model335Enums.MonitorOutUnits* *attribute*), 118
 SENSOR_FAULT (*lakeshore.model_346.Model346Enums.RelayThermometry* *attribute*), 192
 SENSOR_NAME (*lakeshore.model_335.Model335Enums.DisplayFieldUnits* *attribute*), 120
 SENSOR_NAME (*lakeshore.model_336.Model336Enums.DisplayFieldUnits* *attribute*), 153
 SENSOR_NAME (*lakeshore.model_372.Model372Enums.DisplayFieldUnits* *attribute*), 231
 SENSOR_UNITS (*lakeshore.model_335.Model335Enums.DisplayFieldUnits* *attribute*), 120
 SENSOR_UNITS (*lakeshore.model_336.Model336Enums.DisplayFieldUnits* *attribute*), 153
 set_alarm_parameters() (*lakeshore.model_224.Model224* *method*), 240
 set_alarm_settings() (*lakeshore.model_346.Model346* *method*), 158

`set_analog_out_state()` (*lakeshore.model_346.Model346 method*), (*lakeshore.ssm_source_module.SourceModule method*), 61
`set_analog_output()` (*lakeshore.teslameter.Teslameter method*), 14
`set_analog_output_settings()` (*lakeshore.model_346.Model346 method*), 163
`set_analog_output_signal()` (*lakeshore.teslameter.Teslameter method*), 14
`set_automatic_coupling()` (*lakeshore.ssm_source_module.SourceModule method*), 57
`set_averaging_time()` (*lakeshore.ssm_measure_module.MeasureModule method*), 72
`set_band_pass_filter_center()` (*lakeshore.teslameter.Teslameter method*), 15
`set_bias_voltage()` (*lakeshore.ssm_measure_module.MeasureModule method*), 73
`set_brightness()` (*lakeshore.model_240.Model240 method*), 253
`set_cmf_source()` (*lakeshore.ssm_source_module.SourceModule method*), 57
`set_cmf_state()` (*lakeshore.ssm_source_module.SourceModule method*), 58
`set_cmr_source()` (*lakeshore.ssm_source_module.SourceModule method*), 57
`set_cmr_state()` (*lakeshore.ssm_source_module.SourceModule method*), 58
`set_control_loop_zone_table()` (*lakeshore.model_346.Model346 method*), 166
`set_control_setpoint()` (*lakeshore.model_346.Model346 method*), 185
`set_coupling()` (*lakeshore.ssm_measure_module.MeasureModule method*), 72
`set_coupling()` (*lakeshore.ssm_source_module.SourceModule method*), 56
`set_current()` (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 25
`set_current()` (*lakeshore.model_121.Model121 method*), 261
`set_current_amplitude()` (*lakeshore.ssm_source_module.SourceModule method*), 60
`set_current_limit()` (*lakeshore.model_155.PrecisionSource method*), 266
`set_current_limit()` (*lakeshore.ssm_source_module.SourceModule method*), 61
`set_current_mode_voltage_protection()` (*lakeshore.model_155.PrecisionSource method*), 266
`set_current_offset()` (*lakeshore.ssm_source_module.SourceModule method*), 60
`set_current_output_limit_high()` (*lakeshore.ssm_source_module.SourceModule method*), 65
`set_current_output_limit_low()` (*lakeshore.ssm_source_module.SourceModule method*), 65
`set_current_ramp_configuration()` (*lakeshore.ssm_source_module.SourceModule method*), 67
`set_current_range()` (*lakeshore.model_155.PrecisionSource method*), 266
`set_curve()` (*lakeshore.model_224.Model224 method*), 241
`set_curve()` (*lakeshore.model_346.Model346 method*), 185
`set_curve_data_point()` (*lakeshore.model_224.Model224 method*), 241
`set_curve_data_point()` (*lakeshore.model_240.Model240 method*), 255
`set_curve_data_point()` (*lakeshore.model_346.Model346 method*), 186
`set_curve_header()` (*lakeshore.model_224.Model224 method*), 241
`set_curve_header()` (*lakeshore.model_240.Model240 method*), 254
`set_curve_header()` (*lakeshore.model_346.Model346 method*), 186
`set_dark_mode_state()` (*lakeshore.ssm_measure_module.MeasureModule method*), 81
`set_dark_mode_state()` (*lakeshore.ssm_source_module.SourceModule method*), 64
`set_description()` (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 87
`set_digital_high_pass_filter_state()` (*lakeshore.ssm_measure_module.MeasureModule method*), 82
`set_disable_on_compliance()` (*lakeshore.ssm_source_module.SourceModule method*), 65
`set_display_brightness()`

(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 30
method), 26

set_display_brightness()
(*lakeshore.model_121.Model121* method), 261

set_display_contrast()
(*lakeshore.model_224.Model224* method), 238

set_display_field_settings()
(*lakeshore.model_224.Model224* method), 245

set_duty()
(*lakeshore.ssm_source_module.SourceModule* method), 56

set_enable_state()
(*lakeshore.ssm_source_module.SourceModule* method), 55

set_factory_defaults()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27

set_factory_defaults()
(*lakeshore.model_121.Model121* method), 262

set_factory_defaults()
(*lakeshore.model_240.Model240* method), 254

set_field_control_open_loop_voltage()
(*lakeshore.teslameter.Teslameter* method), 13

set_field_control_setpoint()
(*lakeshore.teslameter.Teslameter* method), 13

set_filter()
(*lakeshore.model_224.Model224* method), 243

set_filter()
(*lakeshore.model_240.Model240* method), 255

set_filter()
(*lakeshore.model_346.Model346* method), 157

set_frequency()
(*lakeshore.ssm_source_module.SourceModule* method), 56

set_frequency_filter_type()
(*lakeshore.teslameter.Teslameter* method), 14

set_frequency_range_threshold()
(*lakeshore.ssm_measure_module.MeasureModule* method), 81

set_front_panel_lock()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 26

set_gain_allocation_strategy()
(*lakeshore.ssm_measure_module.MeasureModule* method), 73

set_guard_state()
(*lakeshore.ssm_source_module.SourceModule* method), 57

set_hardware_error_enable_mask()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 239

set_heater_aurorange()
(*lakeshore.model_346.Model346* method), 160

set_heater_limit()
(*lakeshore.model_346.Model346* method), 165

set_heater_pid()
(*lakeshore.model_346.Model346* method), 186

set_heater_range()
(*lakeshore.model_346.Model346* method), 160

set_heater_setup()
(*lakeshore.model_346.Model346* method), 161

set_high_pass_filter_cutoff()
(*lakeshore.teslameter.Teslameter* method), 15

set_i_amplitude()
(*lakeshore.ssm_source_module.SourceModule* method), 60

set_i_limit()
(*lakeshore.ssm_source_module.SourceModule* method), 61

set_i_offset()
(*lakeshore.ssm_source_module.SourceModule* method), 60

set_identify_state()
(*lakeshore.ssm_measure_module.MeasureModule* method), 81

set_identify_state()
(*lakeshore.ssm_source_module.SourceModule* method), 64

set_ieee_488()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27

set_ieee_488()
(*lakeshore.model_224.Model224* method), 238

set_ieee_interface_mode()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 27

set_input_configuration()
(*lakeshore.ssm_measure_module.MeasureModule* method), 72

set_input_curve()
(*lakeshore.model_224.Model224* method), 239

set_input_curve()
(*lakeshore.model_346.Model346* method), 186

set_input_diode_excitation_current()
(*lakeshore.model_224.Model224* method), 237

set_input_parameter()
(*lakeshore.model_240.Model240* method), 256

set_input_sensor()
(*lakeshore.model_346.Model346* method), 159

set_internal_water()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 26

set_keypad_lock()
(*lakeshore.model_224.Model224* method), 239

set_keypad_lock() (*lakeshore.model_346.Model346* method), 54
 (*method*), 164
 set_led_state() (*lakeshore.model_224.Model224* method), 239
 set_limits() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24
 set_lock_in_averaging_state() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 set_lock_in_fir_cycles() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 set_lock_in_fir_state() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 set_lock_in_iir_state() (*lakeshore.ssm_measure_module.MeasureModule* method), 76
 set_lock_in_lowpass_state() (*lakeshore.ssm_measure_module.MeasureModule* method), 76
 set_lock_in_reference_cycles() (*lakeshore.ssm_measure_module.MeasureModule* method), 77
 set_lock_in_rolloff() (*lakeshore.ssm_measure_module.MeasureModule* method), 76
 set_lock_in_time_constant() (*lakeshore.ssm_measure_module.MeasureModule* method), 75
 set_low_pass_filter_cutoff() (*lakeshore.teslameter.Teslameter* method), 14
 set_magnet_water() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 26
 set_manual_output() (*lakeshore.model_346.Model346* method), 187
 set_measure_function() (*lakeshore.ssm_measure_module.MeasureModule* method), 86
 set_mode() (*lakeshore.ssm_measure_module.MeasureModule* method), 72
 set_modname() (*lakeshore.model_240.Model240* method), 256
 set_mon_out_manual_level() (*lakeshore.ssm_system.SSMSystem* method), 50
 set_mon_out_mode() (*lakeshore.ssm_system.SSMSystem* method), 49
 set_mon_out_state() (*lakeshore.ssm_system.SSMSystem* method), 49
 set_name() (*lakeshore.ssm_measure_module.MeasureModule* method), 71
 set_name() (*lakeshore.ssm_source_module.SourceModule* method), 55
 set_operation_event_enable_mask() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 29
 set_operation_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall* method), 40
 set_operation_event_enable_mask() (*lakeshore.model_155.PrecisionSource* method), 269
 set_operation_event_enable_mask() (*lakeshore.model_346.Model346* method), 187
 set_operation_event_enable_mask() (*lakeshore.ssm_measure_module.MeasureModule* method), 81
 set_operation_event_enable_mask() (*lakeshore.ssm_source_module.SourceModule* method), 64
 set_operation_event_enable_mask() (*lakeshore.ssm_system.SSMSystem* method), 53
 set_operation_event_enable_mask() (*lakeshore.teslameter.Teslameter* method), 18
 set_operational_error_enable_mask() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 30
 set_output() (*lakeshore.model_155.PrecisionSource* method), 264
 set_output_mode() (*lakeshore.model_346.Model346* method), 163
 set_output_percent_limit() (*lakeshore.model_346.Model346* method), 162
 set_output_percent_limit() (*lakeshore.model_346.Model346* method), 161
 set_power_up_enable() (*lakeshore.model_121.Model121* method), 262
 set_profibus_address() (*lakeshore.model_240.Model240* method), 256
 set_profibus_slot_configuration() (*lakeshore.model_240.Model240* method), 257
 set_profibus_slot_count() (*lakeshore.model_240.Model240* method), 256
 set_programming_mode() (*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 54

method), 27

set_qualifier_latching_setting()
(*lakeshore.teslameter.Teslameter* method), 15

set_questionable_event_enable_mask()
(*lakeshore.fast_hall_controller.FastHall* method), 40

set_questionable_event_enable_mask()
(*lakeshore.model_155.PrecisionSource* method), 269

set_questionable_event_enable_mask()
(*lakeshore.model_346.Model346* method), 187

set_questionable_event_enable_mask()
(*lakeshore.ssm_measure_module.MeasureModule* method), 80

set_questionable_event_enable_mask()
(*lakeshore.ssm_source_module.SourceModule* method), 63

set_questionable_event_enable_mask()
(*lakeshore.ssm_system.SSMSystem* method), 53

set_questionable_event_enable_mask()
(*lakeshore.teslameter.Teslameter* method), 18

set_ramp_rate()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24

set_ramp_segment()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 24

set_ramp_segments_enable()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 25

set_readback_nplcycles()
(*lakeshore.ssm_source_module.SourceModule* method), 68

set_ref_in_edge()
(*lakeshore.ssm_system.SSMSystem* method), 48

set_ref_out_source()
(*lakeshore.ssm_system.SSMSystem* method), 48

set_ref_out_state()
(*lakeshore.ssm_system.SSMSystem* method), 48

set_reference_harmonic()
(*lakeshore.ssm_measure_module.MeasureModule* method), 75

set_reference_phase_shift()
(*lakeshore.ssm_measure_module.MeasureModule* method), 75

set_reference_source()
(*lakeshore.ssm_measure_module.MeasureModule* method), 75

set_relative_baseline()
(*lakeshore.ssm_measure_module.MeasureModule* method), 78

set_relative_field_baseline()
(*lakeshore.teslameter.Teslameter* method), 12

set_relay_alarms()
(*lakeshore.model_224.Model224* method), 246

set_relay_alarms()
(*lakeshore.model_346.Model346* method), 167

set_resistance_auto_range()
(*lakeshore.ssm_measure_module.MeasureModule* method), 84

set_resistance_excitation_type()
(*lakeshore.ssm_measure_module.MeasureModule* method), 83

set_resistance_mode()
(*lakeshore.ssm_measure_module.MeasureModule* method), 83

set_resistance_observation_time_requested()
(*lakeshore.ssm_measure_module.MeasureModule* method), 85

set_resistance_observation_time_state()
(*lakeshore.ssm_measure_module.MeasureModule* method), 84

set_resistance_optimization_state()
(*lakeshore.ssm_measure_module.MeasureModule* method), 84

set_resistance_range()
(*lakeshore.ssm_measure_module.MeasureModule* method), 84

set_resistance_source()
(*lakeshore.ssm_measure_module.MeasureModule* method), 83

set_sensor_name()
(*lakeshore.model_224.Model224* method), 238

set_sensor_name()
(*lakeshore.model_240.Model240* method), 255

set_sensor_name()
(*lakeshore.model_346.Model346* method), 187

set_service_request()
(*lakeshore.model_224.Model224* method), 236

set_service_request()
(*lakeshore.model_346.Model346* method), 187

set_service_request_enable_mask()
(*lakeshore.em_power_supply.ElectromagnetPowerSupply* method), 28

set_service_request_enable_mask()
(*lakeshore.fast_hall_controller.FastHall* method), 40

set_service_request_enable_mask()
(*lakeshore.model_155.PrecisionSource* method), 269

set_service_request_enable_mask()

(lakeshore.model_346.Model346 method), 187
set_service_request_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 54
set_service_request_enable_mask() (*lakeshore.teslameter.Teslameter method*), 18
set_setpoint_ramp_parameter() (*lakeshore.model_346.Model346 method*), 188
set_shape() (*lakeshore.ssm_source_module.SourceModule method*), 55
set_source_function() (*lakeshore.ssm_source_module.SourceModule method*), 55
set_stability_settings() (*lakeshore.model_346.Model346 method*), 159
set_standard_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 40
set_standard_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 269
set_standard_event_enable_mask() (*lakeshore.model_224.Model224 method*), 236
set_standard_event_enable_mask() (*lakeshore.model_346.Model346 method*), 188
set_standard_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 54
set_standard_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 18
set_standard_event_status_enable_mask() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 28
set_sweep_configuration() (*lakeshore.ssm_source_module.SourceModule method*), 66
set_temperature_limit() (*lakeshore.model_346.Model346 method*), 188
set_threshold() (*lakeshore.model_346.Model346 method*), 158
set_to_factory_defaults() (*lakeshore.model_224.Model224 method*), 236
set_voltage_amplitude() (*lakeshore.ssm_source_module.SourceModule method*), 62
set_voltage_limit() (*lakeshore.model_155.PrecisionSource method*), 266
set_voltage_limit() (*lakeshore.ssm_source_module.SourceModule method*), 63
set_voltage_mode_current_protection() (*lakeshore.model_155.PrecisionSource method*), 266
set_voltage_offset() (*lakeshore.ssm_source_module.SourceModule method*), 62
set_voltage_output_limit_high() (*lakeshore.ssm_source_module.SourceModule method*), 64
set_voltage_output_limit_low() (*lakeshore.ssm_source_module.SourceModule method*), 64
set_voltage_ramp_configuration() (*lakeshore.ssm_source_module.SourceModule method*), 66
set_voltage_range() (*lakeshore.model_155.PrecisionSource method*), 266
set_voltage_sense_mode() (*lakeshore.ssm_measure_module.MeasureModule method*), 86
set_voltage_sense_mode() (*lakeshore.ssm_source_module.SourceModule method*), 68
set_wait_to_continue() (*lakeshore.model_224.Model224 method*), 236
set_website_login() (*lakeshore.model_224.Model224 method*), 240
 SETPOINT_1 (*lakeshore.model_335.Model335Enums.InputChannel attribute*), 120
 SETPOINT_2 (*lakeshore.model_335.Model335Enums.InputChannel attribute*), 120
 SettingsProfiles (class in *lakeshore.ssm_settings_profiles*), 87
setup_ac_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
setup_dc_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 77
setup_lock_in_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 78
 SEVEN (*lakeshore.model_372.Model372Enums.InputChannel attribute*), 229
 SIX (*lakeshore.model_372.Model372Enums.InputChannel attribute*), 229
 SIXTEEN (*lakeshore.model_372.Model372Enums.InputChannel attribute*), 229

stream_data() (*lakeshore.ssm_system.SSMSystem* THERMOCOUPLE (*lakeshore.model_336.Model336Enums.InputSensorType* method), 47 attribute), 152

sweep_current() (*lakeshore.model_155.PrecisionSource* THERMOCOUPLE (*lakeshore.model_346.Model346Enums.InputSensorType* method), 264 attribute), 190

sweep_voltage() (*lakeshore.model_155.PrecisionSource* THERMOCOUPLE (*lakeshore.model_346.Model346Enums.OptionCardType* method), 264 attribute), 193

SYSTEM_STATUS (*lakeshore.model_346.Model346Enums.RelayThermometryControl* THERMOCOUPLE_TYPE_E attribute), 192 (lakeshore.model_346.Model346Enums.StandardCurveType attribute), 192

T

TARE_RELATIVE_FIELD (*lakeshore.model_346.Model346Enums.StandardCurveType* THERMOCOUPLE_TYPE_K attribute), 191 (lakeshore.model_346.Model346Enums.StandardCurveType attribute), 191

TARE_RELATIVE_FIELD (*lakeshore.teslameter.Teslameter* method), 11 THERMOMETRY_INPUT (*lakeshore.model_346.Model346Enums.RelayControl* attribute), 192

TEMPERATURE_EXTRAPOLATED (*lakeshore.model_346.Model346Enums.RelayThermometryControl* THIRTEEN (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 192 attribute), 229

TEMPERATURE_FAULT (*lakeshore.model_346.Model346Enums.RelayThermometryControl* THIRTY_KILOHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* attribute), 192 attribute), 248

TEN (*lakeshore.model_372.Model372Enums.InputChannel* THIRTY_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 229 attribute), 118

TEN_KILOHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* THIRTY_OHM (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 248 attribute), 152

TEN_KILOHMS (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* THIRTY_OHM (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* attribute), 248 attribute), 248

TEN_MICRO_AMPS (*lakeshore.model_224.Model224Enums.DiodeExecutionCurrent* THIRTY_OHM (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* attribute), 251 attribute), 248

TEN_OHM (*lakeshore.model_335.Model335Enums.RTDRange* THIRTY_THOUSAND_OHM attribute), 118 (lakeshore.model_335.Model335Enums.RTDRange attribute), 119

TEN_OHM (*lakeshore.model_336.Model336Enums.RTDRange* THIRTY_THOUSAND_OHM attribute), 152 (lakeshore.model_336.Model336Enums.RTDRange attribute), 153

TEN_OHM (*lakeshore.model_346.Model346Enums.PTCRTDRange* THIRTY_THOUSAND_OHM attribute), 190 (lakeshore.model_346.Model346Enums.NTCRTDRange attribute), 190

TEN_OHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* THIRTY_THOUSAND_OHM attribute), 248 (lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange attribute), 190

TEN_OHMS (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* THREE (*lakeshore.model_372.Model372Enums.InputChannel* attribute), 248 attribute), 229

TEN_THOUSAND_OHM (*lakeshore.model_335.Model335Enums.RTDRange* THREE_HUNDRED_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 119 attribute), 118

TEN_THOUSAND_OHM (*lakeshore.model_336.Model336Enums.RTDRange* THREE_HUNDRED_OHM (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 153 attribute), 152

TEN_THOUSAND_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange* THREE_HUNDRED_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange* attribute), 190 attribute), 190

TEN_VOLTS (*lakeshore.model_335.Model335Enums.DiodeRange* THREE_HUNDRED_OHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* attribute), 118 attribute), 248

TEN_VOLTS (*lakeshore.model_336.Model336Enums.DiodeRange* THREE_HUNDRED_OHMS (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* attribute), 152 attribute), 248

Teslameter (class in *lakeshore.teslameter*), 10

TeslameterOperationRegister (class in *lakeshore.teslameter*), 19 THREE_KILOHMS (*lakeshore.model_224.Model224Enums.NTCRTDSensorResistanceRange* attribute), 248

TeslameterQuestionableRegister (class in *lakeshore.teslameter*), 19 THREE_KILOHMS (*lakeshore.model_224.Model224Enums.PlatinumRTDSensorResistanceRange* attribute), 248

THERMOCOUPLE (*lakeshore.model_335.Model335Enums.InputSensorType* THREE_THOUSAND_OHM (*lakeshore.model_335.Model335Enums.RTDRange* attribute), 118 attribute), 119

THREE_THOUSAND_OHM (*lakeshore.model_336.Model336Enums.RTDRange* attribute), 119

- attribute*), 152
 - THREE_THOUSAND_OHM (*lakeshore.model_346.Model346Enums.NTCRTDRange*
attribute), 190
 - THRESHOLD_1 (*lakeshore.model_346.Model346Enums.RelayThermometryCondition*
attribute), 192
 - THRESHOLD_2 (*lakeshore.model_346.Model346Enums.RelayThermometryCondition*
attribute), 192
 - THRESHOLD_3 (*lakeshore.model_346.Model346Enums.RelayThermometryCondition*
attribute), 192
 - THRESHOLD_4 (*lakeshore.model_346.Model346Enums.RelayThermometryCondition*
attribute), 192
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyHardwareErrorsRegister
method), 23
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyOperationalErrorsRegister
method), 23
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyOperationEventRegister
method), 23
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyServiceRequestEnableRegister
method), 22
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyStandardEventStatusRegister
method), 23
 - to_integer() (*lakeshore.em_power_supply.ElectromagnetPowerSupply.EMPow*
erSupplyStatusByteRegister
method), 23
 - TRUE (*lakeshore.model_346.Model346Enums.RelayFunctionOutputCondition*
attribute), 193
 - turn_relay_off() (*lakeshore.model_224.Model224*
method), 245
 - turn_relay_off() (*lakeshore.model_346.Model346*
method), 159
 - turn_relay_on() (*lakeshore.model_224.Model224*
method), 245
 - turn_relay_on() (*lakeshore.model_346.Model346*
method), 159
 - TWELVE (*lakeshore.model_372.Model372Enums.InputChannel*
attribute), 229
 - TWO (*lakeshore.model_372.Model372Enums.InputChannel*
attribute), 229
 - TWO_INPUT_A (*lakeshore.model_335.Model335Enums.DisplaySetup*
attribute), 120
 - TWO_INPUT_B (*lakeshore.model_335.Model335Enums.DisplaySetup*
attribute), 120
 - TWO_LOOP (*lakeshore.model_335.Model335Enums.DisplaySetup*
attribute), 120
 - TWO_POINT_FIVE_VOLTS
(*lakeshore.model_335.Model335Enums.DiodeRange*
attribute), 118
 - TWO_POINT_FIVE_VOLTS
(*lakeshore.model_336.Model336Enums.DiodeRange*
attribute), 152
 - TWO_POINT_FIVE_VOLTS
(*lakeshore.model_346.Model346Enums.DiodeRange*
attribute), 190
- ## U
- unload() (*lakeshore.ssm_measure_module.MeasureModule*
method), 86
 - unload() (*lakeshore.ssm_source_module.SourceModule*
method), 65
 - unlock_front_panel() (*lakeshore.model_121.Model121*
method),
262
 - update() (*lakeshore.ssm_settings_profiles.SettingsProfiles*
method), 88
 - USB (*lakeshore.model_224.Model224Enums.RemoteInterface*
attribute), 245
 - use_ac_coupling() (*lakeshore.ssm_measure_module.MeasureModule*
method), 72
 - use_ac_coupling() (*lakeshore.ssm_source_module.SourceModule*
method), 31
 - use_dc_coupling() (*lakeshore.ssm_measure_module.MeasureModule*
method), 72
 - use_dc_coupling() (*lakeshore.ssm_source_module.SourceModule*
method), 31
- ## V
- VAD_CONTROL (*lakeshore.model_372.Model372Enums.MonitorOutputSource*
attribute), 230
 - VAD_MEASUREMENT (*lakeshore.model_372.Model372Enums.MonitorOutput*
attribute), 230
 - VCM_NEG (*lakeshore.model_372.Model372Enums.MonitorOutputSource*
attribute), 230
 - VCM_POS (*lakeshore.model_372.Model372Enums.MonitorOutputSource*
attribute), 230
 - VDIF (*lakeshore.model_372.Model372Enums.MonitorOutputSource*
attribute), 230
 - VOLTAGE (*lakeshore.model_335.Model335Enums.HeaterOutType*
attribute), 119
 - VOLTAGE (*lakeshore.model_372.Model372Enums.SensorExcitationMode*
attribute), 229
 - VOLTAGE_OFF (*lakeshore.model_335.Model335Enums.HeaterVoltageRange*
attribute), 120
 - VOLTAGE_OFF (*lakeshore.model_336.Model336Enums.HeaterVoltageRange*
attribute), 153
 - VOLTAGE_ON (*lakeshore.model_335.Model335Enums.HeaterVoltageRange*
attribute), 120
 - VOLTAGE_ON (*lakeshore.model_336.Model336Enums.HeaterVoltageRange*
attribute), 153
 - VOLTS_PER_KELVIN (*lakeshore.model_224.Model224Enums.CurveFormat*
attribute), 250
 - VOLTS_PER_KELVIN (*lakeshore.model_240.Model240Enums.CurveFormat*
attribute), 259
- ## W
- WARM_UP_HEATER (*lakeshore.model_372.Model372Enums.HeaterOutput*
attribute), 234
 - WARMUP (*lakeshore.model_372.Model372Enums.OutputMode*
attribute), 229

WARMUP_HEATER (*lakeshore.model_372.Model372Enums.DisplayInfo attribute*), 231

WARMUP_HEATER_ZONE (*lakeshore.model_372.Model372Enums.RelayControlMode attribute*), 230

WARMUP_SUPPLY (*lakeshore.model_335.Model335Enums.HeaterOutputMode attribute*), 119

WARMUP_SUPPLY (*lakeshore.model_336.Model336Enums.HeaterOutputMode attribute*), 153

write() (*lakeshore.em_power_supply.ElectromagnetPowerSupply method*), 31

write() (*lakeshore.fast_hall_controller.FastHall method*), 40

write() (*lakeshore.model_121.Model121 method*), 263

write() (*lakeshore.model_155.PrecisionSource method*), 269

write() (*lakeshore.model_224.Model224 method*), 247

write() (*lakeshore.model_240.Model240 method*), 258

write() (*lakeshore.model_346.Model346 method*), 188

write() (*lakeshore.model_350.Model350 method*), 195

write() (*lakeshore.model_425.Model425 method*), 20

write() (*lakeshore.ssm_system.SSMSystem method*), 54

write() (*lakeshore.teslameter.Teslameter method*), 19

Z

zero_dc_resistance() (*lakeshore.ssm_measure_module.MeasureModule method*), 85

zero_relative_baseline() (*lakeshore.ssm_measure_module.MeasureModule method*), 78

ZONE (*lakeshore.model_335.Model335Enums.HeaterOutputMode attribute*), 119

ZONE (*lakeshore.model_336.Model336Enums.HeaterOutputMode attribute*), 153

ZONE (*lakeshore.model_346.Model346Enums.OutputMode attribute*), 191

ZONE (*lakeshore.model_372.Model372Enums.OutputMode attribute*), 229