Lake Shore Python Driver Documentation

Release 1.5.3

Lake Shore Cryotronics, Inc.

Jul 02, 2021

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 . . . 8
2.3.1 Thread Safety . . . 8
2.3.2 Logging . . . 8
2.3.3 Status Registers . . . 8
2.3.4 Instrument initialization options . . . 9
2.3.5 Python 2 compatibility . . . 9
2.4 Supported Instruments . . . 9
2.4.1 Magnetics Instruments . . . 10
2.4.2 Magnet System Power Supplies . . . 20
2.4.3 Materials Characterization . . . 21
2.4.4 Temperature Controllers . . . 60
2.4.5 Temperature Monitors . . . 164
2.4.6 Sources . . . 192

Python Module Index **201**

Index **203**

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.

CHAPTER 1

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.

CHAPTER 2

Table of contents

2.1 Installation

2.1.1 Python version

The Lake Shore Python driver is compatible with Python 3.x.

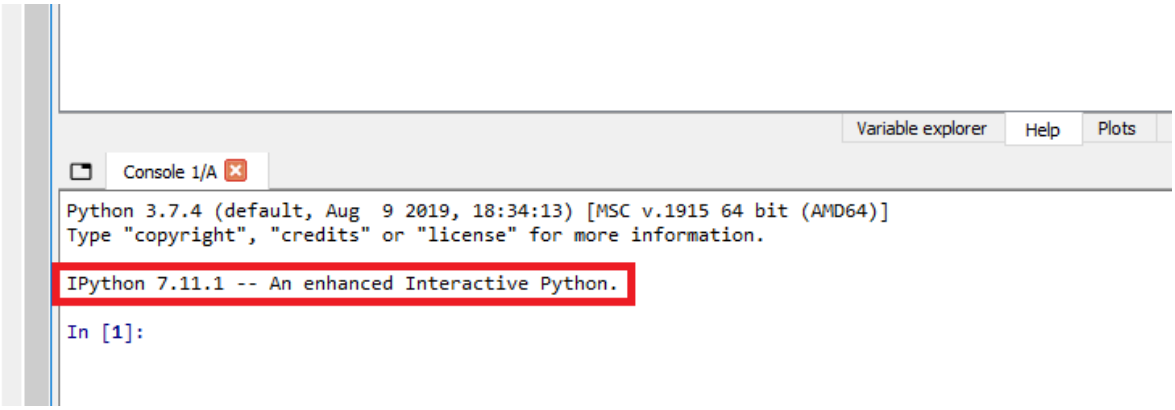
2.1.2 Install the Lake Shore Python driver

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).

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 Teslameter

my_instrument = Teslameter()
# 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 Teslameter

my_instrument = Teslameter()
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 treads 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 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 Magnetics 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 namedtuples 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)
- "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"
- "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
- "CLLOOP" (closed loop control)
- "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 setpoint will transition to a new setpoint. 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 are:
- "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" (Corrrected X channel field measurement)
- "YCOR" (Corrected Y channel field measurement)
- "ZCOR" (Corrected Z channel field measurement)
- "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):

- "LPASS" (low pass filter)
- "HPASS" (high pass filter)
- "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)

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)

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 qualifer * "OVER" * "UNDER" * "BETWEEN" * "OUTSIDE" * "ABSBETWEEN" * "ABSOUTSIDE"

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 a SCPI command or multiple commands to the instrument

Args:

commands (str): Any number of SCPI commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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*)
Send a SCPI query or multiple queries to the instrument and return the response(s)

Args:

queries (str): Any number of SCPI queries or commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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.

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*)

Initialize self. See help(type(self)) for accurate signature.

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*)

Initialize self. See help(type(self)) for accurate signature.

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.**Model425**(*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.

2.4.2 Magnet System Power Supplies

Model 643 Electromagnet Power Supply

The Model 643 electromagnet power supply is a linear, bipolar current source providing true 4-quadrant output.

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_643.**Model643**(*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 643 electromagnet magnet power supply

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.

Model 648 Electromagnet Power Supply

The Model 648 electromagnet power supply is a robust, fault-tolerant 9 kW supply optimized for powering large 7 or 10 in research electromagnets.

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_648.**Model648**(*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 648 electromagnet power supply

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.

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")
```

(continues on next page)

(continued from previous page)

```
# 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) + 'A\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)

    # 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*)
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 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)

sample_type(str):

- Indicates sample type. Options are:
- "VDP" (Van der Pauw sample)
- "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)

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)

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)

sample_type(str):

- Indicates sample type. Options are:
- "VDP" (Van der Pauw sample)
- "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)

sample_type(str):

- Indicates sample type. Options are:
- "VDP" (Van der Pauw sample)
- "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 a SCPI command or multiple commands to the instrument

Args:

commands (str): Any number of SCPI commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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*)
Send a SCPI query or multiple queries to the instrument and return the response(s)

Args:

queries (str): Any number of SCPI queries or commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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.

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_curren*
field_control_slew_rate_limit, *field_control_at_voltage_limit*, *current_measurement_overload*, *voltage_measurement_overload*, *invalid_probe*, *invalid_calibration*, *inter_processor_communication_erro*
*field_measurement_communication_*
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.

__init__(*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*)

The constructor for ContackCheckManualParameters class.

Args:

excitation_type (str):

- The excitation type used for the measurement. Options are:
- "CURRENT"
- "VOLTAGE"

excitation_start_value (float): The starting excitation value For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_end_value (float): The ending excitation value For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_range (float or str):

- Excitation range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V: voltage excitation
  - amps in the range of -100e-3 to 100e-3 A: current excitation

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

compliance_limit (float): For voltage excitation, specify the current limit 100e-9 to 100e-3 A For current excitation, specify the voltage compliance 1.00 to 10.0 V

number_of_points (int): The number of points to measure between the excitation start and end. 0 - 100

min_r_squared (float): The minimum R^2 desired. Default is 0.9999.

blanking_time (float or str):

- The time in seconds to wait for hardware to settle before gathering readings. Range of time is 0.5 ms - 300 s with a resolution of 0.1 ms. Options are:
- "DEF" (Default) = 2 ms
- "MIN" = 0.5 ms
- "MAX" = 300 s
- floating point number of seconds

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

__init__(*max_current=0.1*, *max_voltage=10*, *number_of_points=11*, *min_r_squared=0.9999*)

The constructor for ContactCheckOptimizedParameters class.

Args:

max_current(float or str):

- A 'not to exceed' output current value for the auto algorithm to use. Options are:
- "MIN" = 1 uA
- "MAX" = 100 mA
- "DEF" (Default) = 100 mA
- floating point number of amps

max_voltage(float or str):

- A 'not to exceed' output voltage value for the auto algorithm to use. Options are:
- "MIN" = 1 V
- "MAX" = 10 V
- "DEF" (Default) = 10 V
- floating point number of volts

number_of_points(int or str):

- The number of points to measure between the excitation start and end. Options are:
- "MIN" = 2
- "MAX" = 100
- "DEF" (Default) = 11
- integer number of points

min_r_squared(float): The minimum R^2 desired. Default is 0.9999.

class lakeshore.fast_hall_controller.**FastHallManualParameters**(*excitation_type*, *excitation_value*, *user_defined_field*, *compliance_limit*, *excitation_range='AUTO'*, *excitation_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

__init__(*excitation_type*, *excitation_value*, *user_defined_field*, *compliance_limit*, *excitation_range='AUTO'*, *excitation_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*)

The constructor for FastHallManualParameters class

Args:

excitation_type (str):

- The excitation type used for the measurement. Options are:
- "CURRENT"
- "VOLTAGE"

excitation_value(float): For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_range (float or str):

- Excitation range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V for voltage excitation,
  - amps in the range of -100e-3 to 100e-3 A for current excitation

excitation_measurement_range (float or str):

- Excitation measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V: voltage excitation
  - amps in the range of -100e-3 to 100e-3 A: current excitation

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

compliance_limit (float): For voltage excitation, specify the current limit 100e-9 to 100e-3 A For current excitation, specify the voltage compliance 1.00 to 10.0 V

user_defined_field (float): The field, in units of Tesla, the sample is being subjected to. Used for calculations.

max_samples(int): When minimumSnr is omitted or Infinity ('INF'), the total number of samples to average 1 - 1000 When minimumSnr is specified, the maximum number of samples to average 10 - 1000. Default is 100

resistivity (float): The resistivity of the sample in units of Ohm*Meters (bulk) of Ohms Per Square (sheet). Used for calculations. Measure this value using the RESistivity SCPI subsystem. Defaults to 'Nan' (not a number) which will propagate through calculated values

blanking_time (float or str):

- The time in seconds to wait for hardware to settle before gathering readings. Range of time is 0.5 ms - 300 s with a resolution of 0.1 ms. Options are:
- "DEF" (Default) = 2 ms
- "MIN" = 0.5 ms
- "MAX" = 300 s
- floating point number in seconds

averaging_samples (int): The number of voltage compensation samples to average. Only applied for excitation type voltage. 1 - 120. Default is 60.

sample_thickness (float): Thickness of the sample in meters. 0 to 10E-3 m Default is 0 m

min_hall_voltage_snr (float or str):

- The desired signal to noise ratio of the measurement calculated using average hall voltage / error of mean 1 - 1000. Options are:
- "INF" (Infinity)
- "DEF" (Default) = 30
- floating point number to represent the ratio

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

__init__(*user_defined_field*, *measurement_range='AUTO'*, *max_samples=100*, *min_hall_voltage_snr=30*, *averaging_samples=60*, *sample_thickness='DEF'*)

The constructor for FastHallLinkParameters class

Args:

user_defined_field (float): The field, in units of Tesla, the sample is being subjected to. Used for calculations.

measurement_range (float or str): For voltage excitation, specify the current measurement range 0 to 100e-3 A For current excitation, specify the voltage measurement range 0 to 10.0 V. Defaults to 'AUTO'

max_samples(int): When minimumSnr is omitted or Infinity ('INF'), the total number of samples to average 1 - 1000 When minimumSnr is specified, the maximum number of samples to average 10 - 1000 Defaults to 100

min_hall_voltage_snr (float or str):

- The desired signal to noise ratio of the measurement calculated using average hall voltage / error of mean 1 - 1000. Options are:
- "INF" (Infinity)
- "DEF" (Default) = 30
- floating point number to represent the ratio

averaging_samples (int): The number of voltage compensation samples to average. Only applied for excitation type voltage. 1 - 120. Defaults to 60

sample_thickness (float): Thickness of the sample in meters. 0 to 10E-3 m Default is the last run resistivity measurement's sample thickness.

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

__init__(*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*)

The constructor for FourWireParameter class.

Args:

contact_point1 (int): Excitation +. Valid contact points are: 1, 2, 3, 4, 5, or 6. Cannot be the same as Contact Point 2.

contact_point2 (int): Excitation - Valid contact points are: 1, 2, 3, 4, 5, or 6. Cannot be the same as Contact Point 1.

contact_point3 (int): Voltage Measure/Sense +. Valid contact points are: 1, 2, 3, 4, 5, or 6. Cannot be the same as Contact Point 4.

contact_point4 (int): Voltage Measure/Sense -. Valid contact points are: 1, 2, 3, 4, 5, or 6. Cannot be the same as Contact Point 3.

excitation_type (str):

- The excitation type used for the measurement. Options are:
- "CURRENT"
- "VOLTAGE"

excitation_value(float): For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_range (float or str):

- Excitation range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V for voltage excitation,
  - amps in the range of -100e-3 to 100e-3 A for current excitation

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

excitation_measurement_range (float or str):

- Excitation measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V: voltage excitation
  - amps in the range of -100e-3 to 100e-3 A: current excitation

compliance_limit (float): For voltage excitation, specify the current limit 100e-9 to 100e-3 A For current excitation, specify the voltage compliance 1.00 to 10.0 V

blanking_time (float or str):

- The time in seconds to wait for hardware to settle before gathering readings. Range of time is 0.5 ms - 300 s with a resolution of 0.1 ms. Options are:
- "DEF" (Default)= 2 ms
- "MIN" = 0.5 ms
- "MAX" = 300 s
- floating point number in seconds

max_samples(int): When minimumSnr is omitted or Infinity, the total number of samples to average 1 - 1000 When minimumSnr is specified, the maximum number of samples to average 10 - 1000. Default is 100

min_snr (float or str):

- The desired signal to noise ratio of the measurement resistance, calculated using measurement average / error of mean 1 - 1000. Options are:
- "INF" (Infinity)
- "DEF" (Default)= 30
- floating point number to represent the ratio

excitation_reversal (bool): True = Reverse the excitation to generate the resistance. False = no excitation reversal

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

__init__(*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*)

The constructor for DCHallParameters.

Args:

excitation_type (str):

- The excitation type used for the measurement. Options are:
- "CURRENT"
- "VOLTAGE"

excitation_value(float): For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_range (float or str):

- Excitation range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V for voltage excitation,
  - amps in the range of -100e-3 to 100e-3 A for current excitation

excitation_measurement_range (float or str):

- Excitation measurement range based on the excitation type. Options are:

- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V: voltage excitation
  - amps in the range of -100e-3 to 100e-3 A: current excitation

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

compliance_limit (float): For voltage excitation, specify the current limit 100e-9 to 100e-3 A For current excitation, specify the voltage compliance 1.00 to 10.0 V

averaging_samples(int): The number of samples to average 1-1000

user_defined_field(float): The field, in units of Tesla, the sample is being subjected to. Used for calculations.

with_field_reversal (bool): Specifies whether or not to apply reversal field. Default is true

resistivity(float): The resistivity of the sample in units of Ohm*Meters (bulk) of Ohms Per Square (sheet). Used for calculations. Measure this value using the RESistivity SCPI subsystem. Defaults to 'NaN' (not a number) which will propagate through calculated values.

blanking_time (float or str):

- The time in seconds to wait for hardware to settle before gathering readings. Range of time is 0.5 ms - 300 s with a resolution of 0.1 ms. Options are:
- "DEF" (Default) = 2 ms
- "MIN" = 0.5 ms
- "MAX" = 300 s
- floating point number in seconds

sample_thickness (float): Thickness of the sample in meters. 0 to 10e-3 m. Default is 0m

class lakeshore.fast_hall_controller.**ResistivityManualParameters**(*excitation_type*, *excitation_value*, *compliance_limit*, *excitation_range='AUTO'*, *excitation_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

__init__(*excitation_type*, *excitation_value*, *compliance_limit*, *excitation_range='AUTO'*, *excitation_measurement_range='AUTO'*, *measurement_range='AUTO'*, *max_samples=100*, *blanking_time=0.002*, *sample_thickness=0*, *min_snr=30*, ***kwargs*)

The constructor for ResistivityManualParameters class.

Args:

excitation_type (str):

- The excitation type used for the measurement. Options are:
- "CURRENT"
- "VOLTAGE"

excitation_value(float): For voltage -10.0 to 10.0 V For current -100e-3 to 100e-3 A

excitation_range (float or str):

- Excitation range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V for voltage excitation,
  - amps in the range of -100e-3 to 100e-3 A for current excitation

excitation_measurement_range (float or str):

- Excitation measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - volts in the range of 0 to 10.0V: voltage excitation
  - amps in the range of -100e-3 to 100e-3 A: current excitation

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:

- "AUTO": sets the range to the best fit range for a given excitation value
- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

compliance_limit (float): For voltage excitation, specify the current limit 100e-9 to 100e-3 A For current excitation, specify the voltage compliance 1.00 to 10.0 V

max_samples(int): When minimumSnr is omitted or Infinity ('INF'), the total number of samples to average 1 - 1000 When minimumSnr is specified, the maximum number of samples to average 10 - 1000. Default is 100

blanking_time (float or str):

- The time in seconds to wait for hardware to settle before gathering readings. Range of time is 0.5 ms - 300 s with a resolution of 0.1 ms. Options are:
- "DEF" (Default) = 2 ms
- "MIN" = 0.5 ms
- "MAX" = 300 s
- floating point number in seconds

averaging_samples (int): The number of voltage compensation samples to average. Only applied for excitation type voltage. 1 - 120. Default is 60

sample_thickness (float): Thickness of the sample in meters. 0 to 10E-3 m. Default is 0 m

min_snr (float or str):

- The desired signal to noise ratio of the measurement calculated using average resistivity / error of mean 1 - 1000. Options are:
- "INF" (Infinity)
- "DEF" (Default) = 30
- floating point number to represent the ratio

Kwargs:

width(float): The width of the sample in meters. Greater than 0

separation(float): The distance between the sample's arms in meters. Greater than 0

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

__init__(*measurement_range='AUTO'*, *sample_thickness=0*, *min_snr=30*, *max_samples=100*)

The constructor for ResistivityLinkParameters class.

Args:

measurement_range (float or str):

- Measurement range based on the excitation type. Options are:
- "AUTO": sets the range to the best fit range for a given excitation value

- **floating point number of**
  - amps in the range of 0 to 100e-3A: voltage excitation
  - volts in the range of 0 to 10.0V: current excitation

sample_thickness (float): Thickness of the sample in meters. 0 to 10E-3 m. Default is 0 m

min_snr (float or str):

- The desired signal to noise ratio of the measurement calculated using average resistivity / error of mean 1 - 1000. Options are:
- "INF" (Infinity)
- "DEF" (Default)= 30
- floating point number to represent the ratio

max_samples(int): When minimumSnr is omitted or Infinity ('INF'), the total number of samples to average 1 - 1000 When minimumSnr is specified, the maximum number of samples to average 10 - 1000 Default is 100

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

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

Initialize self. See help(type(self)) for accurate signature.

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

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

Initialize self. See help(type(self)) for accurate signature.

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

# 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_i_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_i_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.")
```

SSMS instrument methods

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

Class for interaction with the M81 instrument

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*)
Gets a list of values corresponding to the input data sources.

Args: data_sources (str, int): Variable length list of pairs of (DATASOURCE_MNEMONIC, CHANNEL_INDEX).

Returns: Tuple of values corresponding to the given data sources

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 (str, int): Variable length list of pairs of (DATASOURCE_MNEMONIC, 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 (str, int): Variable length list of pairs of (DATASOURCE_MNEMONIC, 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 (str, int): Pairs of (DATASOURCE_MNEMONIC, 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_head_self_cal_status()
Returns the status 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)

channel_index
alias of `builtins.int`

command(**commands*, *check_errors=True*)
Send a SCPI command or multiple commands to the instrument

Args:

commands (str): Any number of SCPI commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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*)
Send a SCPI query or multiple queries to the instrument and return the response(s)

Args:

queries (str): Any number of SCPI queries or commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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.

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

Args: data_sources str: Variable length list of DATASOURCE_MNEMONIC.

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_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*)
Set 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_excitation_mode()
Returns the excitation mode of the module. 'CURRENT' or 'VOLTAGE'.

set_excitation_mode(*excitation_mode*)
Sets the excitation mode of the module

Args:

excitation_mode (str): 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 channel will follow the frequency set for the specifed channel.

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'

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_cmr_source()
Returns the Common Mode Reduction (CMR) source. 'INTernal', or 'EXTernal'.

set_cmr_source(*cmr_source*)
Sets the Common Mode Reduction (CMR) source.

Args:

cmr_source (str): The new CMR source ('INTernal', or 'EXTernal')

get_cmr_state()
Returns the Common Mode Reduction (CMR) state of the module

set_cmr_state(*cmr_state*)
Sets the Common Mode Reduction (CMR) state of the module

Args:

cmr_state (bool): The new CMR state (True to enable CMR, False to disable CMR)

enable_cmr()
Sets the CMR state of the module to True

disable_cmr()
Sets the CMR state of the module to False

configure_cmr(*cmr_source*, *cmr_state=True*)
Configure Common Mode Reduction (CMR)

Args:

cmr_source (str): The new CMR source ('INTernal', or 'EXTernal')

cmr_state (bool): The new CMR state (True to enable CMR, False to disable CMR)

get_i_range()
Returns the present current range of the module in Amps

get_i_ac_range()
Returns the present AC current range of the module in Amps

get_i_dc_range()
Returns the present DC current range of the module in Amps

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

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.

get_i_amplitude()
Returns the current amplitude for the module in Amps

set_i_amplitude(*amplitude*)
Sets the current amplitude for the module

Args:

amplitude (float): The new current amplitude in Amps

get_i_offset()
Returns the current offset for the module in Amps

set_i_offset(*offset*)
Sets the current offset for the module

Args:

offset (float): The new current offset in Amps

apply_dc_current(*level*, *enable_output=True*)
Apply DC current

Args:

level (float): DC current level in Amps

enable_output (bool): Set the enable state of the module to True

apply_ac_current(*frequency*, *amplitude*, *offset=0.0*, *enable_output=True*)
Apply AC current

Args:

frequency (float): Excitation frequency in Hz

amplitude (float): Current amplitude in Amps

offset (float): Current offset in Amps

enable_output (bool): Set the enable state of the module to True

get_i_limit()
Returns the current limit enforced by the module 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

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

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*, *enable_output=True*)
Apply DC voltage

Args:

level (float): DC voltage level in Volts

enable_output (bool): Set the enable state of the module to True

apply_ac_voltage(*frequency*, *amplitude*, *offset=0.0*, *enable_output=True*)
Apply AC voltage

Args:

frequency (float): Excitation frequency in Hz

amplitude (float): Voltage amplitude in Volts

offset (float): Voltage offset in Volts

enable_output (bool): Set the enable state of the module to True

get_voltage_limit()
Returns the voltage limit enforced by the module in Volts

set_voltage_limit(*v_limit*)
Sets the voltage limit enforced by the module

Args:

v_limit (float): The new limit to apply in Volts

get_voltage_limit_status()
Returns whether the voltage limit circuitry is presently engaged and limiting 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*)
Returns 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

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_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.

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')

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 low pass filter cuttoff frequency. 'NONE', 'F10', 'F30', 'F100', 'F300', 'F1000', 'F3000', or 'F10000'.

get_lowpass_rolloff()
Returns the low pass filter roll-off. 'R6' or 'R12'.

get_highpass_corner_frequency()
Returns the high pass filter cuttoff 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 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 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_i_range()
Returns the current range in Amps

get_i_autorange_status()
Returns whether autoranging is enabled for the module

configure_i_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.

get_voltage_range()
Returns the voltage range in Volts

get_voltage_autorange_status()
Returns whether autoranging is enabled for the module

configure_voltage_range(*autorange*, *max_level*)
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 such that the present phase indication is zero. Coming in 0.3.

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

set_lock_in_fir_state(*state*)
Sets the state of the lock-in PSD output FIR filter

Args:

state (bool): The new state of the PSD output FIR filter

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

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

setup_dc_measurement(*nplc=1*)
Setup the module for DC measurement

Args:

nplc (float): The new number of power line cycles to average

setup_ac_measurement(*nplc=1*)
Setup the module for DC 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*)
Setup 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_fir (bool): Enable or disable the PSD output FIR filter

get_multiple(**data_sources*)
Gets a list of values corresponding to the input data sources for this module.

Args: data_sources (str): Variable length list of DATASOURCE_MNEMONIC.

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

get_dc()
Returns the DC indication in module units

get_rms()
Returns the RMS indication in module units

get_peak_to_peak()
Returns the peak to peak indication in module units

get_positive_peak()
Returns the positive peak indication in module units

get_negative_peak()
Returns the negative peak indication in module units

get_lock_in_x()
Returns the present X indication from the lock-in

get_lock_in_y()
Returns the present Y indication from the lock-in

get_lock_in_r()
Returns the present magnitude indication from the lock-in

get_lock_in_theta()
Returns the present angle indication from the lock-in

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

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_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 operaiton event enable register bits. These values determine which operaiton bits propagate to the operaiton 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*)
Returns 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

Settings Profiles methods

class lakeshore.ssm_settings_profiles.**SettingsProfiles**(*device*)
Class for interaction with settings profiles

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*)
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

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
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

# 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", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.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(Model224SoftCalSensorTypes.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)
```

Recording data with the Model 335

```
from lakeshore.model_335 import *

# 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(Model335InputSensorType.DIODE, True,␣
↪False,
                                               Model335InputSensorUnits.KELVIN,
                                               Model335DiodeRange.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", Model335DiodeCurrent.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 lakeshore.model_335 import Model335DisplaySetup, Model335HeaterResistance, \
    Model335HeaterOutputDisplay, Model335HeaterRange, Model335AutoTuneMode,␣
↪Model335HeaterError
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(Model335DisplaySetup.TWO_INPUT_A)
```

(continues on next page)

(continued from previous page)

```
# Configure heater output 1 using the HeaterSetup class and set_heater_setup method
my_model_335.set_heater_setup_one(Model335HeaterResistance.HEATER_50_OHM, 1.0,␣
↪Model335HeaterOutputDisplay.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, Model335HeaterRange.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 Model335HeaterError.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, Model335AutoTuneMode.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)

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

Return:

(float):

- Analog voltage heater output percentage

set_autotune(*output*, *mode*)
Initiates autotuning of the heater control loop.

Args:

output (int):

- Specifies the output associated with the loop to be Autotuned

mode (IntEnum):

- Specifies the Autotune mode
- Member of instrument's AutoTuneMode IntEnum class

set_brightness(*brightness*)
Method to set the front display brightness

Args:

brightness (IntEnum)

- A member of the instrument's BrightnessLevel IntEnum class

get_brightness()
Method to query the front display brightness

Return:

(IntEnum):

- A member of the instrument's BrightnessLevel IntEnum class

get_operation_condition()
Returns 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.

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.

get_thermocouple_junction_temp()
This method returns the temperature of the ceramic thermocouple block used in the room temperature compensation calculation

Return:

(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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

set_diode_excitation_current (*channel*, *excitation_current*)

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 (IntEnum):

- 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"

Return:

(IntEnum):

- A member of the instrument's DiodeCurrent IntEnum class
- Diode excitation current

get_tuning_control_status()

If initial conditions are not met when starting the autotune procedure, causing the autotuning process to never actually begin, then the error status will be set to 1 and the stage status will be stage 00

Return:

(dict):

- Keys:
- **active_tuning_enable (bool):**
  - False = no active tuning, True = active tuning
- **output (int):**
  - Heater output of the control loop being tuned
- **tuning_error (bool):**
  - False = no tuning error, True = tuning error
- **stage_status (int):**
  - 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

Return:

(dict)

- Keys:
- **"filter_enable": bool**
  - Specified whether the filtering function is enabled or not
- **"data_points": int**
  - Specifies how many points the filter function uses
- **"reset_threshold": int**
  - 1% - 10%
  - 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.

set_monitor_output_heater(*channel*, *high_value*, *low_value*, *units=<Model335MonitorOutUnits.KELVIN: 1>*, *polarity=<Polarity.UNIPOLAR: 0>*)

Configures output 2. Use the set_heater_output_mode command to set the output mode to Monitor Out.

Args:

channel (Model335InputSensor):

- 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% outputif unipolar. Entered in the units designated by the <units> argument

units (Model335MonitorOutUnits):

- Specifies the units on which to base the output voltage

polarity (Model335Polarity):

- Specifies output voltage is unipolar or bipolar

get_monitor_output_heater()

Used to obtain all monitor out parameters for output 2.

Return:

(dict):

- See set_monitor_output_heater method arguments
- **Keys:**
  - "channel": Model335InputSensor
  - "units": Model335MonitorOutUnits
  - "high_value": float
  - "low_value": float
  - "polarity": Model335Polarity

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 (Model335DisplaySetup):

- Specifies the front panel display mode
- See Model335DisplaySetup IntEnum class

get_display_setup()
Returns the display mode

Return:

(Model335DisplaySetup):

- Specifies the front panel display mode
- See Model335DisplaySetup IntEnum class

set_heater_setup_one(*heater_resistance*, *max_current*, *output_display_mode*)
Method to configure heater output one.

Args:

heater_resistance (Model335HeaterResistance):

- See Model335HeaterResistance IntEnum class

max_current (float):

- Specifies the maximum current for the heater

output_display_mode (Model335HeaterOutputDisplay):

- Specifies how the heater output is displayed
- See Model335HeaterOutType IntEnum class

set_heater_setup_two(*output_type*, *heater_resistance*, *max_current*, *display_mode*)
Method to configure the heater output 2.

Args:

output_type (Model335HeaterOutType):

- Specifies wheter the heater output is in constant current or voltage mode
- See Model335HeaterOutType IntEnum class

heater_resistance (Model335HeaterResistance):

- See Model335HeaterResistance IntEnum class

max_current (float):

- Specifies the maximum current for the heater

display_mode (Model335HeaterOutType):

- Specifies how the heater output is displayed
- Required only if output_type is set to CURRENT
- See Model335HeaterOutType 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
- **Keys:**
  - "output_type": Model335HeaterOutType
  - "heater_resistance": Model335HeaterResistance
  - "max_current": float
  - "output_display_mode": Model335HeaterOutputDisplay

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 (Model335InputSensorSettings):

- See Model335InputSensorSettings class

get_input_sensor (*channel*)

Returns the sensor type and associated parameters.

Args:

channel (str):

- Specifies sensor input to configure

- "A" or "B"

Return:

(Model335InputSensorSettings):

- See Model335InputSensor 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 (Model335HeaterOutputMode):

- Member of Model335HeaterOutputMode IntEnum class
- Specifies the control mode

channel (Model335InputSensor)

- 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):

- **Keys:**
  - "mode": Model335HeaterOutputMode
  - "channel": Model335InputSensor
  - "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 (Model335Polarity)

- Specifies whether output voltage is UNIPOLAR or BIPOLAR

get_output_2_polarity()
Returns the polarity of output 2

Return:

(Model335Polarity)

- 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:**
  - Model335HeaterRange IntEnum member
- **For Output 2 in Voltage mode:**
  - Model335HeaterVoltageRange 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:**
  - Model335HeaterRange IntEnum member
- **For Output 2 in Voltage mode:**
  - Model335HeaterVoltageRange 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 (Model335WarmupControl):

- 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):

- **Keys:**
  - "control": Model335WarmupControl
  - "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:

(Model335ControlLoopZone):

- See Model335ControlLoopZone class

clear_interface_command()
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.

command(**commands*, *check_errors=True*)
Send a SCPI command or multiple commands to the instrument

Args:

commands (str):

- A serial command

Kwargs:

check_errors (bool):

- Chooses whether to check for and raise errors after sending a command. True by default.

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

Return:

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.

Return:

(dict)

- Keys:
- **"high_state": bool**
  - True if high state is on, False if high state is off
- **"low_state" bool**
  - True if low state is on, False if low state is off

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)

Return:

value (float):

- The value for the setpoint (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

Return:

data_points (list: tuple):

- **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

Return:

curve_point (tuple)

- (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

Return:

(dict):

- See set_display_field_settings method
- Keys:
- "input_channel": IntEnum
- "display_units": IntEnum

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

Return:

(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

Return:

(dict):

- Keys:
- **"gain": float**
  - Proportional term in PID control.
- **"integral": float**
  - Integral term in PID control.
- **"ramp_rate": float**

- 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)

Return:

(IntEnum):

- Object of instrument's HeaterError type

get_ieee_488()
Returns the IEEE address set

Return:

address (int):

- 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

Return:

curve_number (int):

- 0-59

get_kelvin_reading(*input_channel*)
Returns the temperature value in kelvin of the given channel

Args:

input_channel:

- Selects the channel to retrieve measurement

get_keypad_lock()
Returns the state of the keypad lock and the lock-out code.

Return:

(dict):

- **Keys:**
  - "state": bool
  - "code": int

get_led_state()
Returns whether or not front panel LEDs are enabled.

Return:

(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

Return:

(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

Return:

(dict):

- **keys:**
  - "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

Return:

(dict):

- Keys:
- "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:

(IntEnum):

- 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

Return:

(IntEnum):

- A member of the instrument's InterfaceMode IntEnum class

get_self_test()
Instrument self test result completed at power up

Return:

(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.

Returns:

reading (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

Return:

(dict):

- Keys:
- "ramp_enable": bool
- "rate_value": float

get_setpoint_ramp_status(*output*)
"Returns whether or not the setpoint is ramping

Args:

output (int):

- Specifies which output's control loop to query

Return:

(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_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

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

Return:

- 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*)
Control settings, that is, P, I, D, and Setpoint, 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 setpoint (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):

- **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)

- Optional argument
- Specify only if the point is part of a cubic spindle curve
- The curvature value scale used to calculate spindle coefficients to 6 digits

set_curve_header(*curve_number*, *curve_header*)
Configures the user curve header

Args:

curve_number:

- 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 (IntEnum)

- Defines which input to display.
- A member of the instrument's InputChannel IntEnum class

display_units (IntEnum)

- Defines which units to display reading in.
- A member of the instrument's DisplayUnits 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 setpoint will transition to a new setpoint.
- The ramp rate is configured in field units per second.

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 (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 (IntEnum):

- 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):

- 0.1 to 100
- Specifies setpoint 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 setpoint ramping were off

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): An 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

Settings classes

class lakeshore.model_335.**Model335InputSensorSettings**(*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 (Model335InputSensorType):

- Specifies input sensor type

autorange_enable (bool):

- Specifies autoranging
- False = off and True = on

compensation (bool):

- Specifies input compensation
- False = off and True = on

units (Model335InputSensorUnits):

- Specifies the preferred units parameter for sensor readings and for the control setpoint

input_range (IntEnum)

- Specifies input range if autorange_enable is false
- **See IntEnum classes:**
  - Model335DiodeRange
  - Model335RTDRange
  - Model335ThermocoupleRange

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

__init__(*upper_bound*, *proportional*, *integral*, *derivative*, *manual_output_value*, *heater_range*, *channel*, *ramp_rate*)

Constructor

Args:

upper_bound (float):

- Specifies the upper Setpoint 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_output_value (float):

- Specifies the manual output for this zone
- 0 to 100 %

heater_range (Model335HeaterRange):

- Specifies the heater range for this zone
- See Model335HeaterRange IntEnum class

channel (Model335InputSensor):

- See Model335InputSensor IntEnum class

ramp_rate (float):

- Specifies the ramp rate for this zone
- 0 - 100 K/min

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.

lakeshore.model_335.**Model335RelayControlMode**
alias of *lakeshore.temperature_controllers.RelayControlMode*

class lakeshore.temperature_controllers.**RelayControlMode**
Relay operating mode enumeration

ALARMS = 2

RELAY_OFF = 0

RELAY_ON = 1

lakeshore.model_335.**Model335RelayControlAlarm**
alias of *lakeshore.temperature_controllers.RelayControlAlarm*

class lakeshore.temperature_controllers.**RelayControlAlarm**
Enumeration of the setting determining which alarm(s) cause a relay to close in alarm mode.

BOTH_ALARMS = 2

HIGH_ALARM = 1

LOW_ALARM = 0

lakeshore.model_335.**Model335InterfaceMode**
alias of *lakeshore.temperature_controllers.InterfaceMode*

class lakeshore.temperature_controllers.**InterfaceMode**
Enumeration for the mode of the remote interface

LOCAL = 0

REMOTE = 1

REMOTE_LOCAL_LOCK = 2

lakeshore.model_335.**Model335HeaterError**
alias of *lakeshore.temperature_controllers.HeaterError*

class lakeshore.temperature_controllers.**HeaterError**
Enumeration for possible errors flagged by the heater

HEATER_OPEN_LOAD = 1

HEATER_SHORT = 2

NO_ERROR = 0

lakeshore.model_335.**Model335CurveFormat**
alias of *lakeshore.temperature_controllers.CurveFormat*

class lakeshore.temperature_controllers.**CurveFormat**
Enumerations specify formats for temperature sensor curves

LOG_OHMS_PER_KELVIN = 4

MILLIVOLT_PER_KELVIN = 1

OHMS_PER_KELVIN = 3

VOLTS_PER_KELVIN = 2

lakeshore.model_335.**Model335CurveTemperatureCoefficient**
alias of *lakeshore.temperature_controllers.CurveTemperatureCoefficient*

class lakeshore.temperature_controllers.**CurveTemperatureCoefficient**
Enumerations specify positive/negative temperature sensor curve coefficients

NEGATIVE = 1

POSITIVE = 2

lakeshore.model_335.**Model335AutoTuneMode**
alias of *lakeshore.temperature_controllers.AutotuneMode*

class lakeshore.temperature_controllers.**AutotuneMode**
Enumerator used to represent the different autotune control modes

P_I = 1

P_I_D = 2

P_ONLY = 0

lakeshore.model_335.**Model335HeaterResistance**
alias of *lakeshore.temperature_controllers.HeaterResistance*

class lakeshore.temperature_controllers.**HeaterResistance**
Enumerator used to represent the different heater resistances

HEATER_25_OHM = 1

HEATER_50_OHM = 2

lakeshore.model_335.**Model335HeaterOutputUnits**
alias of *lakeshore.temperature_controllers.HeaterOutputUnits*

class lakeshore.temperature_controllers.**HeaterOutputUnits**
Enumerator used to represent heater output unit settings

CURRENT = 1

POWER = 2

class lakeshore.model_335.**Model335InputSensor**
Enumeration when "NONE" is an option for sensor input

CHANNEL_A = 1

CHANNEL_B = 2

NONE = 0

class lakeshore.model_335.**Model335MonitorOutUnits**
Units associated with a sensor channel

CELSIUS = 2

KELVIN = 1

SENSOR = 3

lakeshore.model_335.**Model335Polarity**
alias of *lakeshore.temperature_controllers.Polarity*

class lakeshore.temperature_controllers.**Polarity**
Enumerator for unipolar or bipolar output operation

BIPOLAR = 1

UNIPOLAR = 0

class lakeshore.model_335.**Model335InputSensorType**
Sensor type enumeration

DIODE = 1

DISABLED = 0

NTC_RTD = 3

PLATINUM_RTD = 2

THERMOCOUPLE = 4

lakeshore.model_335.**Model335InputSensorUnits**
alias of *lakeshore.temperature_controllers.InputSensorUnits*

class lakeshore.temperature_controllers.**InputSensorUnits**
Enumerator used to represent temperature sensor unit options

CELSIUS = 2

KELVIN = 1

SENSOR = 3

class lakeshore.model_335.**Model335DiodeRange**
Diode voltage range enumeration

TEN_VOLTS = 1

TWO_POINT_FIVE_VOLTS = 0

class lakeshore.model_335.**Model335RTDRange**
RTD resistance range enumeration

HUNDRED_OHM = 2

ONE_HUNDRED_THOUSAND_OHM = 8

ONE_THOUSAND_OHM = 4

TEN_OHM = 0

TEN_THOUSAND_OHM = 6

THIRTY_OHM = 1

THIRTY_THOUSAND_OHM = 7

THREE_HUNDRED_OHM = 3

THREE_THOUSAND_OHM = 5

class lakeshore.model_335.**Model335ThermocoupleRange**
Thermocouple range enumeration

FIFTY_MILLIVOLT = 0

lakeshore.model_335.**Model335BrightnessLevel**
alias of *lakeshore.temperature_controllers.BrightnessLevel*

class lakeshore.temperature_controllers.**BrightnessLevel**
Enumerator to specify the brightness level of an instrument display

FULL = 3

HALF = 1

QUARTER = 0

THREE_QUARTERS = 2

class lakeshore.model_335.**Model335HeaterOutType**
Heater output 2 enumeration

CURRENT = 0

VOLTAGE = 1

class lakeshore.model_335.**Model335HeaterOutputDisplay**
Heater output display units enumeration

CURRENT = 1

POWER = 2

class lakeshore.model_335.**Model335HeaterOutputMode**
Control loop enumeration

CLOSED_LOOP = 1

MONITOR_OUT = 4

OFF = 0

OPEN_LOOP = 3

WARMUP_SUPPLY = 5

ZONE = 2

class lakeshore.model_335.**Model335WarmupControl**
Heater output 2 voltage mode warmup enumerations

AUTO_OFF = 0

CONTINUOUS = 1

class lakeshore.model_335.**Model335HeaterRange**
Control loop heater range enumeration

HIGH = 3

LOW = 1

MEDIUM = 2

OFF = 0

lakeshore.model_335.**Model335ControlTypes**
alias of *lakeshore.temperature_controllers.ControlTypes*

class lakeshore.temperature_controllers.**ControlTypes**
Enumerator used to represent the control type settings

AUTO_OFF = 0

CONTINUOUS = 1

lakeshore.model_335.**Model335DiodeCurrent**
alias of *lakeshore.temperature_controllers.DiodeCurrent*

class lakeshore.temperature_controllers.**DiodeCurrent**
Enumerator used to represent diode current ranges

ONE_MILLIAMP = 1

TEN_MICROAMPS = 0

class lakeshore.model_335.**Model335DisplaySetup**
Panel display setup enumeration

CUSTOM = 6

INPUT_A = 0

INPUT_A_MAX_MIN = 1

INPUT_B = 3

INPUT_B_MAX_MIN = 4

TWO_INPUT_A = 2

TWO_INPUT_B = 5

TWO_LOOP = 7

class lakeshore.model_335.**Model335HeaterVoltageRange**
Voltage mode heater enumerations

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class lakeshore.model_335.**Model335DisplayInputChannel**
Panel display information enumeration

INPUT_A = 1

INPUT_B = 2

NONE = 0

OUTPUT_1 = 5

OUTPUT_2 = 6

SETPOINT_1 = 3

SETPOINT_2 = 4

class lakeshore.model_335.**Model335DisplayFieldUnits**
Panel display units enumeration

CELSIUS = 2

KELVIN = 1

MAXIMUM_DATA = 5

MINIMUM_DATA = 4

SENSOR_NAME = 6

SENSOR_UNITS = 3

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 *lakeshore.temperature_controllers.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 *lakeshore.temperature_controllers.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_overrange*, *sensor_units_zero*, *sensor_units_overrange*)

Class object representing the input status flag bits

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
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

# 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", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.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(Model224SoftCalSensorTypes.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, Model336HeaterResistance, Model336HeaterOutputUnits,␣
↪Model336InputChannel, \
                      Model336InputSensorUnits, Model336Polarity, Model336HeaterRange,
↪ Model336HeaterVoltageRange, \
                      Model336HeaterOutputMode

# 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, Model336HeaterResistance.HEATER_50_OHM, 0.75,␣
↪Model336HeaterOutputUnits.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, Model336InputChannel.CHANNEL_A,␣
↪Model336InputSensorUnits.KELVIN, 3.65, 1.02,
                                       Model336Polarity.UNIPOLAR)

# Set closed loop output mode for heater 1
my_model_336.set_heater_output_mode(1, Model336HeaterOutputMode.CLOSED_LOOP,␣
↪Model336InputChannel.CHANNEL_A)

# Set closed loop output mode for heater 3
my_model_336.set_heater_output_mode(3, Model336HeaterOutputMode.CLOSED_LOOP,␣
↪Model336InputChannel.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, Model336HeaterRange.MEDIUM)
my_model_336.set_heater_range(3, Model336HeaterVoltageRange.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

Return:

(float):

- Analog voltage heater output percentage

set_autotune(*output*, *mode*)

Initiates autotuning of the heater control loop.

Args:

output (int):

- Specifies the output associated with the loop to be Autotuned

mode (IntEnum):

- 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

get_operation_condition()

Returns 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.

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.

get_thermocouple_junction_temp()
This method returns the temperature of the ceramic thermocouple block used in the room temperature compensation calculation

Return:

(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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

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):

- Tuple of two floats in the form (temperature_value, sensor_value)
- Optional parameter

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

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

Return:

(dict)

- Keys:
- **"filter_enable": bool**
  - Specified whether the filtering function is enabled or not
- **"data_points": int**
  - Specifies how many points the filter function uses
- **"reset_threshold": int**
  - 1% - 10%
  - 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.

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

Return:

(dict):

- See set_network_settings arguments

get_network_configuration()
Method to return the configured ethernet parameters

Return:

(dict)

- Keys:
- **"lan_status": LanStatus**
  - Current status of the ethernet connection
  - Member of the instrument's LanStatus IntEnum class
- **"ip_address": str**
  - Configured IP address
- **"sub_mask": str**
  - Configured subnet mask
- **"gateway": str**
  - Configured gateway address

- **"primary_dns": str**
  - Configured primary DNS address
- **"secondary_dns": str**
  - Configured secondary DNS address
- **"hostname" str**
  - Assigned hostname
- **"domain": str**
  - Assigned domain
- **"mac_address": str**
  - 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.

Return:

website_login (dict):

- A dictionary containing 15 character string type items
- Keys:
- "username": str
- "password": str

get_celsius_reading(*channel*)
Returns the temperature in celsius of any channel

Args:

channel (str)

- "A" - "D" (in addition, "D1" - "D5" for 3062 option)

set_interface(*interface*)
Selects the remote interface for the instrument

Args:

interface (IntEnum):

- Member of instrument's Interface IntEnum class

get_interface()
Returns the remote interface for the instrument

Return:

(IntEnum):

- Member of instrument's Interface IntEnum class

get_tuning_control_status()

If initial conditions are not met when starting the autotune procedure, causing the autotuning process to never actually begin, then the error status will be set to 1 and the stage status will be stage 00

Return:

(dict):

- Keys:
- **active_tuning_enable (bool):**
  - False = no active tuning, True = active tuning
- **output (int):**
  - Heater output of the control loop being tuned
- **tuning_error (bool):**
  - False = no tuning error, True = tuning error
- **stage_status (int):**
  - 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*)

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 (IntEnum):

- 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"

Return:

(IntEnum):

- 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 (Model336InputChannel):

- Specifies which sensor input to monitor
- A member of the Model336InputChannel IntEnum class

units (Model336InputSensorUnits):

- Specifies the units on which to base the output voltage
- A member of the Model336InputSensorUnits 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% outputif unipolar. Entered in the units designated by the <units> argument

polarity (Model336Polarity):

- Specifies whether the output voltage is unipolar or bipolar
- Member of the Model336Polarity 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

Return:

(dict):

- See set_monitor_output_heater arguments

set_display_setup(*mode*, *num_fields=''*, *displayed_output=''*)
Sets the display mode

Args:

mode (Model336DisplaySetupMode):

- Member of Model336DisplaySetupMode 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 Model336DisplayFields
- **When mode is set to all inputs, specifies size of readings**
  - Member of Model336DisplayFieldsSize

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

Return:

(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 (Model336HeaterResistance):

- Member of Model336HeaterResistance IntEnum class

max_current (float):

- User defined maximum output current (see table 4-11 for max current and resistance relationships)

heater_output (Model336HeaterOutputUnits):

- Specifies whether the heater output displays in current or power
- Member of Model336HeaterOutputUnits 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

Return:

(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"

Return:

(Model336InputSensorSettings):

- See Model336InputSensorSettings class

get_all_kelvin_reading()
Returns the temperature value in kelvin of all channels.

Return:

(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 (Model336HeaterOutputMode):

- Member of Model336HeaterOutputMode IntEnum class

- Specifies the control mode

channel (Model336InputChannel):

- Model336InputChannel 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

Return:

(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 Model336HeaterRange IntEnum class
- **For Outputs 3 and 4:**
  - Model336HeaterVoltageRange 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)

Return:

(IntEnum):

- **For Outputs 1 and 2:**
  - Member of Model336HeaterRange IntEnum class
- **For Outputs 3 and 4:**
  - Member of Model336HeaterVoltageRange IntEnum class

all_heaters_off()
Recreates the front panel safety feature of shutting off all heaters

get_input_reading_status(*channel*)
Retruns the state of the input status flag bits.

Args:

channel (str):

- Specifies which channel to query
- "A" - "D"
- "D1" - "D5" for 3062 option

Return:

(Model336InputReadingStatus):

- Boolean representation of each bit in the input status flag register

get_all_sensor_reading()
Returns the sensor unit reading of all channels

Return:

(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 (Model336ControlTypes):

- Member of the Model336ControlTypes 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

Return:

(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

Return:

(Model336ControlLoopZoneSettings):

- See Model336ControlLoopZoneSettings class

clear_interface_command()
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.

command(**commands*, *check_errors=True*)
Send a SCPI command or multiple commands to the instrument

Args:

commands (str):

- A serial command

Kwargs:

check_errors (bool):

- Chooses whether to check for and raise errors after sending a command. True by default.

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

Return:

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.

Return:

(dict)

- Keys:
- **"high_state": bool**
  - True if high state is on, False if high state is off
- **"low_state" bool**
  - True if low state is on, False if low state is off

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)

Return:

value (float):

- The value for the setpoint (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

Return:

data_points (list: tuple):

- **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

Return:

curve_point (tuple)

- (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

Return:

(dict):

- See set_display_field_settings method
- Keys:
- "input_channel": IntEnum
- "display_units": IntEnum

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

Return:

(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

Return:

(dict):

- Keys:
- **"gain": float**
  - Proportional term in PID control.
- **"integral": float**
  - Integral term in PID control.
- **"ramp_rate": float**
  - 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)

Return:

(IntEnum):

- Object of instrument's HeaterError type

get_ieee_488()
Returns the IEEE address set

Return:

address (int):

- 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

Return:

curve_number (int):

- 0-59

get_kelvin_reading(*input_channel*)
Returns the temperature value in kelvin of the given channel

Args:

input_channel:

- Selects the channel to retrieve measurement

get_keypad_lock()
Returns the state of the keypad lock and the lock-out code.

Return:

(dict):

- **Keys:**
  - "state": bool
  - "code": int

get_led_state()
Returns whether or not front panel LEDs are enabled.

Return:

(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

Return:

(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

Return:

(dict):

- **keys:**
  - "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

Return:

(dict):

- Keys:
- "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:

(IntEnum):

- 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

Return:

(IntEnum):

- A member of the instrument's InterfaceMode IntEnum class

get_self_test()
Instrument self test result completed at power up

Return:

(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.

Returns:

reading (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

Return:

(dict):

- Keys:
- "ramp_enable": bool
- "rate_value": float

get_setpoint_ramp_status(*output*)
"Returns whether or not the setpoint is ramping

Args:

output (int):

- Specifies which output's control loop to query

Return:

(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_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

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

Return:

- 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*)
Control settings, that is, P, I, D, and Setpoint, 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 setpoint (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):

- **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)

- Optional argument
- Specify only if the point is part of a cubic spindle curve
- The curvature value scale used to calculate spindle coefficients to 6 digits

set_curve_header(*curve_number*, *curve_header*)
Configures the user curve header

Args:

curve_number:

- 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 (IntEnum)

- Defines which input to display.
- A member of the instrument's InputChannel IntEnum class

display_units (IntEnum)

- Defines which units to display reading in.
- A member of the instrument's DisplayUnits 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 setpoint will transition to a new setpoint.
- The ramp rate is configured in field units per second.

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 (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 (IntEnum):

- 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):

- 0.1 to 100
- Specifies setpoint 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 setpoint ramping were off

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): An 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

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 (Model336InputSensorType):

- Specifies input sensor type

autorange_enable (bool):

- Specifies autoranging
- False = off and True = on

compensation (bool):

- Specifies input compensation

- False = off and True = on

units (Model336InputSensorUnits):

- Specifies the preferred units parameter for sensor readings and for the control setpoint

input_range (IntEnum)

- Specifies input range if autorange_enable is false
- **See IntEnum classes:**
  - Model336DiodeRange
  - Model336RTDRange
  - Model336ThermocoupleRange

class lakeshore.model_336.**Model336ControlLoopZoneSettings**(*upper_bound*, *proportional*, *integral*, *derivative*, *manual_out_value*, *heater_range*, *channel*, *rate*)

Control loop configuarion 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 Setpoint 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 (Model336HeaterRange):

- Specifies the heater range for this zone
- See Model336HeaterRange IntEnum class

channel (Model336InputChannel):

- See Model336InputChannel 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 *lakeshore.temperature_controllers.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 of 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 *lakeshore.temperature_controllers.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 (IntEnum):

- Member of the instrument's CurveFormat IntEnum class
- Specifies the curve data format

temperature_limit (float):

- Specifies the curve temperature limit in Kelvin

coefficient (IntEnum):

- Member of instrument's CurveTemperatureCoefficient IntEnum class
- Specifies the curve temperature coefficient

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.model_336.**Model336InputChannel**

Enumeration where "NONE" is an option for sensor input

CHANNEL_A = 1

CHANNEL_B = 2

CHANNEL_C = 3

CHANNEL_D = 4

CHANNEL_D2 = 5

CHANNEL_D3 = 6

CHANNEL_D4 = 7

CHANNEL_D5 = 8

NONE = 0

class lakeshore.model_336.**Model336DisplaySetupMode**

Front panel display setup enum

ALL_INPUTS = 6

CUSTOM = 4

FOUR_LOOP = 5

INPUT_A = 0

INPUT_B = 1

INPUT_C = 2

INPUT_D = 3

INPUT_D2 = 7

INPUT_D3 = 8

INPUT_D4 = 9

INPUT_D5 = 10

class lakeshore.model_336.**Model336InputSensorType**
Sensor type enumeration. THERMOCOUPLE is only valid with the 3060 option, CAPACITANCE is only valid with the 3061 option

CAPACITANCE = 5

DIODE = 1

DISABLED = 0

NTC_RTD = 3

PLATINUM_RTD = 2

THERMOCOUPLE = 4

class lakeshore.model_336.**Model336DiodeRange**
Diode voltage range enumeration

TEN_VOLTS = 1

TWO_POINT_FIVE_VOLTS = 0

class lakeshore.model_336.**Model336RTDRange**
RTD resistance range enumeration. THIRTY_THOUSAND_OHM and ONE_HUNDRED_THOUSAND_OHM are only valid for NTC RTDs

HUNDRED_OHM = 2

ONE_HUNDRED_THOUSAND_OHM = 8

ONE_THOUSAND_OHM = 4

TEN_OHM = 0

TEN_THOUSAND_OHM = 6

THIRTY_OHM = 1

THIRTY_THOUSAND_OHM = 7

THREE_HUNDRED_OHM = 3

THREE_THOUSAND_OHM = 5

class lakeshore.model_336.**Model336ThermocoupleRange**
Thermocouple range enumeration

FIFTY_MILLIVOLT = 0

class lakeshore.model_336.**Model336HeaterOutputMode**
Control loop enumeration

CLOSED_LOOP = 1

MONITOR_OUT = 4

OFF = 0

OPEN_LOOP = 3

WARMUP_SUPPLY = 5

ZONE = 2

class lakeshore.model_336.**Model336HeaterRange**
Current mode heater enumerations

HIGH = 3

LOW = 1

MEDIUM = 2

OFF = 0

class lakeshore.model_336.**Model336HeaterVoltageRange**
Voltage mode heater enumerations

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class lakeshore.model_336.**Model336DisplayUnits**
Panel display units enumeration

CELSIUS = 2

KELVIN = 1

MAXIMUM_DATA = 5

MINIMUM_DATA = 4

SENSOR_NAME = 6

SENSOR_UNITS = 3

lakeshore.model_336.**Model336RelayControlMode**
alias of *lakeshore.temperature_controllers.RelayControlMode*

lakeshore.model_336.**Model336RelayControlAlarm**
alias of *lakeshore.temperature_controllers.RelayControlAlarm*

lakeshore.model_336.**Model336InterfaceMode**
alias of *lakeshore.temperature_controllers.InterfaceMode*

lakeshore.model_336.**Model336HeaterError**
alias of *lakeshore.temperature_controllers.HeaterError*

lakeshore.model_336.**Model336CurveFormat**
alias of *lakeshore.temperature_controllers.CurveFormat*

lakeshore.model_336.**Model336CurveTemperatureCoefficients**
alias of *lakeshore.temperature_controllers.CurveTemperatureCoefficient*

lakeshore.model_336.**Model336AutoTuneMode**
alias of *lakeshore.temperature_controllers.AutotuneMode*

lakeshore.model_336.**Model336HeaterResistance**
alias of *lakeshore.temperature_controllers.HeaterResistance*

lakeshore.model_336.**Model336Polarity**
alias of *lakeshore.temperature_controllers.Polarity*

lakeshore.model_336.**Model336DiodeCurrent**
alias of *lakeshore.temperature_controllers.DiodeCurrent*

lakeshore.model_336.**Model336HeaterOutputUnits**
alias of *lakeshore.temperature_controllers.HeaterOutputUnits*

lakeshore.model_336.**Model336InputSensorUnits**
alias of *lakeshore.temperature_controllers.InputSensorUnits*

lakeshore.model_336.**Model336ControlTypes**
alias of *lakeshore.temperature_controllers.ControlTypes*

lakeshore.model_336.**Model336LanStatus**
alias of *lakeshore.temperature_controllers.LanStatus*

class lakeshore.temperature_controllers.**LanStatus**
Represents the different status states for the lan connection

ACQUIRING_ADDRESS = 8

ADDRESS_NOT_ACQUIRED_ERROR = 3

AUTO_IP = 2

CABLE_UNPLUGGED = 6

DHCP = 1

DUPLICATE_INITIAL_IP_ERROR = 4

DUPLICATE_ONGOING_IP_ERROR = 5

ETHERNET_DISABLED = 9

MODULE_ERROR = 7

STATIC_IP = 0

lakeshore.model_336.**Model336Interface**
alias of *lakeshore.temperature_controllers.Interface*

class lakeshore.temperature_controllers.**Interface**
Enumerator used to represent remote interface communication methods

ETHERNET = 1

IEEE488 = 2

USB = 0

lakeshore.model_336.**Model336DisplayFields**
alias of *lakeshore.temperature_controllers.DisplayFields*

class lakeshore.temperature_controllers.**DisplayFields**
Enumeration of the possible number of fields to include in a custom display mode.

LARGE_2 = 0

LARGE_4 = 1

SMALL_8 = 2

lakeshore.model_336.**Model336DisplayFieldsSize**
alias of *lakeshore.temperature_controllers.DisplayFieldsSize*

class lakeshore.temperature_controllers.**DisplayFieldsSize**
Enumeration of the display fields when mode is set to all inputs.

LARGE = 1

SMALL = 0

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 *lakeshore.temperature_controllers.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_bi

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_bi

lakeshore.model_336.**Model336OperationEvent**

alias of *lakeshore.temperature_controllers.OperationEvent*

class lakeshore.model_336.**Model336InputReadingStatus**(*invalid_reading*, *temp_underrange*, *temp_overrange*, *sensor_units_zero*, *sensor_units_overrange*)

Class object representing the input staus flag bits

bit_names = ['invalid_reading', '', '', '', 'temp_underrange', 'temp_overrange', 'sens

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.

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
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

# 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", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.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(Model224SoftCalSensorTypes.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)
```

Using enums to configure an input sensor

```
from lakeshore import Model372
from lakeshore import Model372SensorExcitationMode,␣
↪Model372MeasurementInputCurrentRange, \
    Model372AutoRangeMode, Model372InputSensorUnits,␣
↪Model372MeasurementInputResistance, 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(Model372SensorExcitationMode.CURRENT,
                                             Model372MeasurementInputCurrentRange.
↪RANGE_31_POINT_6_MICRO_AMPS,
                                             Model372AutoRangeMode.CURRENT, False,␣
↪Model372InputSensorUnits.KELVIN,
                                             Model372MeasurementInputResistance.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
from lakeshore import Model372HeaterOutputSettings, Model372OutputMode,␣
↪Model372InputChannel, 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(Model372OutputMode.ZONE,␣
↪Model372InputChannel.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)
# 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 interace, such as 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 (Model372DisplayMode): Sets the input to monitor on the display, or configures display for custom.

number_of_fields (Model372DisplayFields): Configures the number of display fields to include in a custom display.

displayed_info (Model372DisplayInfo): 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:

(Model372DisplayMode): 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: Model372DisplayMode, number_of_fields: Model372DisplayFields, displayed_info: Model372DisplayInfo

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
  - "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
  - "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
  - "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
  - "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
- "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
- "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
  - "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)
  - 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)
  - 2: output 2 (analog heater)

heater_range (Enum or bool)

- Specifies the range of the output. Options:

  Sample Heater (Enum):

- Object of type Model372SampleHeaterOutputRange

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)
  - 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 Model372SampleHeaterOutputRange

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
  - "A" (control input)

state (bool)

- **Specifies whether to turn filter on or off. Options are:**
  - False for off, 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
- "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
- "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)

- 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)
  - 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 or not CMR is set for measurement channels

Returns:

- False (boolean) if CMR is disabled
- 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) or 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 autoscan in on, False if autoscan 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)

- Returns True is beep is enabled.
- Returns False is 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 or not to turn on auto control. Options are:**
  - False for auto off, 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 or not to turn on auto control. Returns:**
  - False for auto off, True for continuous

current (float)

- Specifies percent of full current to apply to external output.

set_setpoint_kelvin(*output_channel*, *setpoint*)
Sets the control setpoint in Kelvin. Changes input parameters so preferred units are Kelvin.

Args:

output_channel (int)

- **Specifies which heater to set a setpoint. Options are:**
  - 0: sample heater
  - 1: output 1 (warm up heater)

setpoint (float)

- Specifies the setpoint the heater ramps to, in Kelvin.

set_setpoint_ohms(*output_channel*, *setpoint*)
Sets the control setpoint in Ohms. Changes input parameters so preferred units are Ohms.

Args:

output_channel (int)

- **Specifies which heater to set a setpoint. Options are:**
  - 0: sample heater
  - 1: output 1 (warm up heater)

setpoint (float)

- Specifies the setpoint the heater ramps to, in Kelvin.

get_setpoint_kelvin(*output_channel*)
Returns the setpoint 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 setpoint. Options are:**
  - 0: sample heater
  - 1: output 1 (warm up heater)

Returns:

setpoint (float)

- Setpoint of the output in Kelvin.

get_setpoint_ohms(*output_channel*)
Returns the setpoint 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 setpoint. Options are:**
  - 0: sample heater
  - 1: output 1 (warm up heater)

Returns:

setpoint (float)

- Setpoint 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
  - "A" : control input

Returns:

frequency (Enum)

- The excitation frequency in Hz, returned as an object of Model372InputFrequency 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
  - "A" : control input

frequency (Enum)

- The excitation frequency in Hz (if float), represented as an object of type Model372InputFrequency

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 (Model372Interface)

- selects the interface based on the values as defined in the Model372Interface enum class

get_interface()
Returns the interface connected to the instrument.

Returns:

interface (Model372Interface)

- returns the interface as an object of the Model372Interface 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
  - "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
- "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 (Model372InterfaceMode)

- returns the mode as an enum type of class Model372InterfaceMode

set_ieee_interface_mode(*mode*)
Sets the IEEE interface mode of the instrument.

Args:

mode (Model372InterfaceMode)

- 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 (Model372MonitorOutputSource)

- Defines the source to run the monitor output off of.

get_monitor_output_source()
Returns the source for the monitor output.

Returns:

source (Model372MonitorOutputSource)

- returns the source as an object of the Model372MonitorOutputSource 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: Model372HeaterOutputUnits}

get_sample_heater_setup()
Returns the setup of the sample heater (channel 0).

Returns:

(dict): {resistance: float, units: Model372HeaterOutputUnits}

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 (Model372HeaterResistance):

- Heater load in ohms, as an object of the enum type Model372HeaterResistance

max_current (float):

- User specified max current in A.

units (Model372HeaterOutputUnits):

- 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 (Model372HeaterOutputUnits):

- 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 (Model372InputSensorUnits) 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 reaches 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: Model372InputSensorUnits, 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)
- 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
  - 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
  - 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
  - "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.

clear_interface_command()
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.

command(**commands*, *check_errors=True*)
Send a SCPI command or multiple commands to the instrument
Args:

commands (str):

- A serial command

Kwargs:

check_errors (bool):

- Chooses whether to check for and raise errors after sending a command. True by default.

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.

Return:

(dict)

- Keys:
- **"high_state": bool**
  - True if high state is on, False if high state is off
- **"low_state" bool**
  - True if low state is on, False if low state is off

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)

Return:

value (float):

- The value for the setpoint (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

Return:

data_points (list: tuple):

- **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

Return:

curve_point (tuple)

- (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

Return:

(dict):

- See set_display_field_settings method
- Keys:
- "input_channel": IntEnum

- "display_units": IntEnum

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

Return:

(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

Return:

(dict):

- Keys:
- **"gain": float**
  - Proportional term in PID control.
- **"integral": float**
  - Integral term in PID control.
- **"ramp_rate": float**
  - 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)

Return:

(IntEnum):

- Object of instrument's HeaterError type

get_ieee_488()

Returns the IEEE address set

Return:

address (int):

- 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

Return:

curve_number (int):

- 0-59

get_kelvin_reading(*input_channel*)
Returns the temperature value in kelvin of the given channel
Args:

input_channel:

- Selects the channel to retrieve measurement

get_keypad_lock()
Returns the state of the keypad lock and the lock-out code.
Return:

(dict):

- **Keys:**
  - "state": bool
  - "code": int

get_led_state()
Returns whether or not front panel LEDs are enabled.
Return:

(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

Return:

(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

Return:

(dict):

- **keys:**
  - "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

Return:

(dict):

- Keys:
- "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:

(IntEnum):

- 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

Return:

(IntEnum):

- A member of the instrument's InterfaceMode IntEnum class

get_self_test()
Instrument self test result completed at power up

Return:

(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.
Returns:

reading (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

Return:

(dict):

- Keys:
- "ramp_enable": bool
- "rate_value": float

get_setpoint_ramp_status(*output*)
"Returns whether or not the setpoint is ramping
Args:

output (int):

- Specifies which output's control loop to query

Return:

(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_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

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

Return:

- 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*)
Control settings, that is, P, I, D, and Setpoint, 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 setpoint (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):

- **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)

- Optional argument
- Specify only if the point is part of a cubic spindle curve
- The curvature value scale used to calculate spindle coefficients to 6 digits

set_curve_header(*curve_number*, *curve_header*)

Configures the user curve header

Args:

curve_number:

- 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 (IntEnum)

- Defines which input to display.
- A member of the instrument's InputChannel IntEnum class

display_units (IntEnum)

- Defines which units to display reading in.
- A member of the instrument's DisplayUnits 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 setpoint will transition to a new setpoint.
- The ramp rate is configured in field units per second.

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 (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 (IntEnum):

- 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):

- 0.1 to 100
- Specifies setpoint 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 setpoint ramping were off

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): An 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

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 Model372InputChannel

__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 autoscanning 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), 1 - 59

temperature_coefficient (Model372CurveTemperatureCoefficient)

- 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 Model372InputChannel

__init__(*mode*, *excitation_range*, *auto_range*, *current_source_shunted*, *units*, *resistance_range=None*)

The constructor for Model372InputSetupSettings class.

Args:

mode (Model372SensorExcitationMode):

- 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, Autoranging Current, or in ROX 102B mode.

current_source_shunted (bool):

- **Specifies whether or not the current source is shunted. If current source is shunted,** excitation is off. If current source is not shunted, excitation is on.

units (Model372InputSensorUnits)

- 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 (Model372OutputMode):

- The control or output mode to configure the heater for. Defines how the output is controlled.

input_channel (Model372InputChannel):

- 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 setpoint during AutoScanning. Options are:
- 1 - 255

polarity (Model372Polarity):

- 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 of 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 *lakeshore.temperature_controllers.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 (IntEnum):

- Member of the instrument's CurveFormat IntEnum class
- Specifies the curve data format

temperature_limit (float):

- Specifies the curve temperature limit in Kelvin

coefficient (IntEnum):

- Member of instrument's CurveTemperatureCoefficient IntEnum class
- Specifies the curve temperature coefficient

lakeshore.model_372.**Model372StandardEventRegister**
alias of *lakeshore.temperature_controllers.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.model_372.**Model372OutputMode**

Enumeration of the different modes for heater output setup.

CLOSED_LOOP = 5

MONITOR_OUT = 1

OFF = 0

OPEN_LOOP = 2

STILL = 4

WARMUP = 6

ZONE = 3

class lakeshore.model_372.**Model372InputChannel**

Enumeration of the input channels of the Model 372

CONTROL = 'A'

EIGHT = 8

ELEVEN = 11

FIFTEEN = 15

FIVE = 5

FOUR = 4

FOURTEEN = 14

NINE = 9

NONE = 0

ONE = 1

SEVEN = 7

SIX = 6

SIXTEEN = 16

TEN = 10

THIRTEEN = 13

THREE = 3

TWELVE = 12

TWO = 2

class lakeshore.model_372.**Model372SensorExcitationMode**
Enumeration of the possible excitation modes for an input sensor.

CURRENT = 1

VOLTAGE = 0

class lakeshore.model_372.**Model372AutoRangeMode**
Enumeration for the possible modes of the auto ranging feature. ROX102B mode is a special autoranging mode that applies only to Lake Shore ROX-102B sensor.

CURRENT = 1

OFF = 0

ROX102B = 2

class lakeshore.model_372.**Model372InputSensorUnits**
Enumeration of the units to handle input readings and display in.

KELVIN = 1

OHMS = 2

class lakeshore.model_372.**Model372MonitorOutputSource**
Enumeration of the source for an output to monitor.

CS_NEG = 1

CS_POS = 2

OFF = 0

VAD_CONTROL = 7

VAD_MEASUREMENT = 6

VCM_NEG = 3

VCM_POS = 4

VDIF = 5

class lakeshore.model_372.**Model372RelayControlMode**
Enumeration of the control modes of the configurable relays of the 372

ALARMS = 2

RELAY_OFF = 0

RELAY_ON = 1

SAMPLE_HEATER_ZONE = 3

WARMUP_HEATER_ZONE = 4

class lakeshore.model_372.**Model372DisplayMode**
Enumeration of the possible information to display

CONTROL_INPUT = 1

CUSTOM = 2

MEASUREMENT_INPUT = 0

class lakeshore.model_372.**Model372DisplayInfo**
Enumeration of the information to a display in the bottom left of the custom display mode

ACTIVE_SCAN_CHANNEL = 3

NONE = 0

SAMPLE_HEATER = 1

WARMUP_HEATER = 2

class lakeshore.model_372.**Model372CurveFormat**
Enumeration of the units to use in a calibration curve

LOGOHM_PER_KELVIN = 4

OHM_PER_KELVIN = 3

OHM_PER_KELVIN_CUBIC_SPLINE = 7

class lakeshore.model_372.**Model372DisplayFieldUnits**
Enumeration for the possible units to display in a single display field.

KELVIN = 1

MAXIMUM_DATA = 5

MINIMUM_DATA = 4

OHMS = 2

QUADRATURE = 3

SENSOR_NAME = 6

class lakeshore.model_372.**Model372SampleHeaterOutputRange**
Enumeration of the output range of the sample heater (output 0).

OFF = 0

RANGE_100_MICRO_AMPS = 2

RANGE_100_MILLI_AMPS = 8

RANGE_10_MILLI_AMPS = 6

RANGE_1_MILLI_AMP = 4

RANGE_316_MICRO_AMPS = 3

RANGE_31_POINT_6_MICRO_AMPS = 1

RANGE_31_POINT_6_MILLI_AMPS = 7

RANGE_3_POINT_16_MILLI_AMPS = 5

class lakeshore.model_372.**Model372InputFrequency**
Defines the enumeration of the excitation frequency of an input.

FREQUENCY_11_POINT_6_HZ = 4

FREQUENCY_13_POINT_7_HZ = 2

FREQUENCY_16_POINT_2_HZ = 3

FREQUENCY_18_POINT_2_HZ = 5

FREQUENCY_9_POINT_8_HZ = 1

class lakeshore.model_372.**Model372MeasurementInputVoltageRange**
Enumerates the possible voltage ranges for a measurement input.

RANGE_200_MICRO_VOLTS = 5

RANGE_200_MILLI_VOLTS = 11

RANGE_20_MICRO_VOLTS = 3

RANGE_20_MILLI_VOLTS = 9

RANGE_2_MICRO_VOLTS = 1

RANGE_2_MILLI_VOLTS = 7

RANGE_632_MICRO_VOLTS = 6

RANGE_632_MILLI_VOLTS = 12

RANGE_63_POINT_2_MICRO_VOLTS = 4

RANGE_63_POINT_2_MILLI_VOLTS = 10

RANGE_6_POINT_32_MICRO_VOLTS = 2

RANGE_6_POINT_32_MILLI_VOLTS = 8

class lakeshore.model_372.**Model372MeasurementInputCurrentRange**
Enumeration of the current range of a measurement input.

RANGE_100_MICRO_AMPS = 17

RANGE_100_NANO_AMPS = 11

RANGE_100_PICO_AMPS = 5

RANGE_10_MICRO_AMPS = 15

RANGE_10_MILLI_AMPS = 21

RANGE_10_NANO_AMPS = 9

RANGE_10_PICO_AMPS = 3

RANGE_1_MICRO_AMP = 13

RANGE_1_MILLI_AMP = 19

RANGE_1_NANO_AMP = 7

RANGE_1_PICO_AMP = 1

RANGE_316_MICRO_AMPS = 18

RANGE_316_NANO_AMPS = 12

RANGE_316_PICO_AMPS = 6

RANGE_31_POINT_6_MICRO_AMPS = 16

RANGE_31_POINT_6_MILLI_AMPS = 22

RANGE_31_POINT_6_NANO_AMPS = 10

RANGE_31_POINT_6_PICO_AMPS = 4

RANGE_3_POINT_16_MICRO_AMPS = 14

RANGE_3_POINT_16_MILLI_AMPS = 20

RANGE_3_POINT_16_NANO_AMPS = 8

RANGE_3_POINT_16_PICO_AMPS = 2

class lakeshore.model_372.**Model372ControlInputCurrentRange**
Enumeration of the current range of the control input.

RANGE_100_NANO_AMPS = 6

RANGE_10_NANO_AMPS = 4

RANGE_1_NANO_AMP = 2

RANGE_316_PICO_AMPS = 1

RANGE_31_POINT_6_NANO_AMPS = 5

RANGE_3_POINT_16_NANO_AMPS = 3

class lakeshore.model_372.**Model372MeasurementInputResistance**
Enumeration of the resistance range of a measurement input.

RANGE_200_KIL_OHMS = 17

RANGE_200_MILLI_OHMS = 5

RANGE_200_OHMS = 11

RANGE_20_KIL_OHMS = 15

RANGE_20_MEGA_OHMS = 21

RANGE_20_MILLI_OHMS = 3

RANGE_20_OHMS = 9

RANGE_2_KIL_OHMS = 13

RANGE_2_MEGA_OHMS = 19

RANGE_2_MILLI_OHMS = 1

RANGE_2_OHMS = 7

RANGE_632_KIL_OHMS = 18

RANGE_632_MILLI_OHMS = 6

RANGE_632_OHMS = 12

RANGE_63_POINT_2_KIL_OHMS = 16

RANGE_63_POINT_2_MEGA_OHMS = 22

RANGE_63_POINT_2_MILLI_OHMS = 4

RANGE_63_POINT_2_OHMS = 10

RANGE_6_POINT_32_KIL_OHMS = 14

RANGE_6_POINT_32_MEGA_OHMS = 20

RANGE_6_POINT_32_MILLI_OHMS = 2

RANGE_6_POINT_32_OHMS = 8

lakeshore.model_372.**Model372CurveTemperatureCoefficient**
alias of *lakeshore.temperature_controllers.CurveTemperatureCoefficient*

lakeshore.model_372.**Model372InterfaceMode**
alias of *lakeshore.temperature_controllers.InterfaceMode*

lakeshore.model_372.**Model372DisplayFields**
alias of *lakeshore.temperature_controllers.DisplayFields*

lakeshore.model_372.**Model372Polarity**
alias of *lakeshore.temperature_controllers.Polarity*

lakeshore.model_372.**Model372HeaterOutputUnits**
alias of *lakeshore.temperature_controllers.HeaterOutputUnits*

lakeshore.model_372.**Model372BrightnessLevel**
alias of *lakeshore.temperature_controllers.BrightnessLevel*

lakeshore.model_372.**Model372HeaterError**
alias of *lakeshore.temperature_controllers.HeaterError*

lakeshore.model_372.**Model372HeaterResistance**
alias of *lakeshore.temperature_controllers.HeaterResistance*

lakeshore.model_372.**Model372Interface**
alias of *lakeshore.temperature_controllers.Interface*

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
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

# 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", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.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
```

(continues on next page)

(continued from previous page)

```
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(Model224SoftCalSensorTypes.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*)

Send a SCPI command or multiple commands to the instrument

Args:

commands (str):

- A serial command

Kwargs:

check_errors (bool):

- Chooses whether to check for and raise errors after sending a command. True by default.

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

Return:

- 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 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

Return:

test_errors (bool):

- True = errors found
- False = 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:**
  - 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.

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, input_c2_reading: float, input_c3_reading: float, input_c4_reading: float, input_c5_reading: float, input_d1_reading: float, input_d2_reading: float, input_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
Return:

(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
Return:

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 or True for on

get_led_state()
Returns whether or not 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.
Return:

(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
Return:

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
Return:

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
Return:

(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}

- **high_state (bool)**
  - True if high state is on, False if high state is off
- **low_state (bool)**
  - True if low state is on, False if low state is off

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

Return:

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: Model224DisplayMode, number_of_fields: Model224NumberOfFields}

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

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

__init__(*high_value*, *low_value*, *deadband*, *latch_enable*, *audible=None*, *visible=None*)
Constructor for Model224AlarmParameters 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 of 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) Optional parameter.

visible (bool): Specifies if the Alarm LED on the instrument front panel will blink when an alarm condition occurs (False = off, True = on) Optional parameter.

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.

__init__(*sensor_type*, *preferred_units*, *sensor_range=None*, *autorange_enabled=False*, *compensation=False*)
Constructor for the Model224InputSensorSettings class.
Args:

sensor_type (Model224InputSensorType or int):

- Specifies what type of sensor is being used at the input.

preferred_units (Model224InputSensorUnits or int):

- Specifies the preferred units used for sensor readings and alarm setpoints when displayed.

sensor_range (IntEnum):

- Specifies the range of the sensor.
- Optional if auto range is enabled

autorange_enabled (bool):

- Defines if autorange is enabled.
- Not applicable for diode sensors
- Defaults to false

compensation (bool):

- Defines if thermal input compensation is on or off.
- Not applicable for diode sensors
- Defaults to false

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

__init__(*curve_name*, *serial_number*, *curve_data_format*, *temperature_limit*, *coefficient*)

Constructor for Model224CurveHeader 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 (Model224CurveFormat): Specifies the curve data format

temperature_limit (float):

- Specifies the curve temperature limit in Kelvin

coefficient (Model224CurveTemperatureCoefficients):

- Specifies the curve temperature coefficient

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.**Model224InputSensorType**

Enumeration for the type of sensor being used for a given input.

DIODE = 1

INPUT_DISABLED = 0

NTC_RTD = 3

PLATINUM_RTD = 2

class lakeshore.model_224.**Model224InputSensorUnits**

Enumeration for the preferred units of an input sensor.

CELSIUS = 2

KELVIN = 1

SENSOR = 3

class lakeshore.model_224.**Model224DiodeExcitationCurrent**
Enum type representing the different excitation currents available for a diode sensor.

ONE_MILLI_AMP = 1

TEN_MICRO_AMPS = 0

class lakeshore.model_224.**Model224DiodeSensorRange**
Enumeration for the voltage range of a diode sensor.

RANGE_10_VOLTS = 1

RANGE_2_POINT_5_VOLTS = 0

class lakeshore.model_224.**Model224PlatinumRTDSensorResistanceRange**
Enumeration of the resistance range of a platinum RTD input sensor.

ONE_HUNDRED_OHMS = 2

ONE_KILOHM = 4

TEN_KILOHMS = 6

TEN_OHMS = 0

THIRTY_OHMS = 1

THREE_HUNDRED_OHMS = 3

THREE_KILOHMS = 5

class lakeshore.model_224.**Model224NTCRTDSensorResistanceRange**
Enumeration of the resistance range of a NTC RTD input sensor.

ONE_HUNDRED_KILOHMS = 8

ONE_HUNDRED_OHMS = 2

ONE_KILOHM = 4

TEN_KILOHMS = 6

TEN_OHMS = 0

THIRTY_KILOHMS = 7

THIRTY_OHMS = 1

THREE_HUNDRED_OHMS = 3

THREE_KILOHMS = 5

class lakeshore.model_224.**Model224InterfaceMode**
Enumeration for the mode of the remote interface

LOCAL = 0

REMOTE = 1

REMOTE_LOCAL_LOCK = 2

class lakeshore.model_224.**Model224RemoteInterface**
Enumeration for the remote interface being used to communicate with the instrument.

ETHERNET = 1

IEEE_488 = 2

USB = 0

class lakeshore.model_224.**Model224DisplayFieldUnits**
Enumerated type defining how units are enumerated for settings and using Display Fields.

CELSIUS = 2

KELVIN = 1

MAXIMUM_DATA = 5

MINIMUM_DATA = 4

SENSOR = 3

class lakeshore.model_224.**Model224InputChannel**
Enumerated type defining which input channels correspond to ints for setting and using Display Fields.

INPUT_A = 1

INPUT_B = 2

INPUT_C = 3

INPUT_C2 = 9

INPUT_C3 = 10

INPUT_C4 = 11

INPUT_C5 = 12

INPUT_D1 = 4

INPUT_D2 = 5

INPUT_D3 = 6

INPUT_D4 = 7

INPUT_D5 = 8

NO_INPUT = 0

class lakeshore.model_224.**Model224DisplayMode**
Enumeration defining what input or information is shown on the front panel display.

ALL_INPUTS = 5

CUSTOM = 4

INPUT_A = 0

INPUT_B = 1

INPUT_C = 2

INPUT_C2 = 10

INPUT_C3 = 11

INPUT_C4 = 12

INPUT_C5 = 13

INPUT_D1 = 3

INPUT_D2 = 6

INPUT_D3 = 7

INPUT_D4 = 8

INPUT_D5 = 9

class lakeshore.model_224.**Model224NumberOfFields**
Enumerated type specifying the number of display fields to configure in the Custom display mode.

LARGE_4 = 0

LARGE_4_SMALL_8 = 2

LARGE_8 = 1

SMALL_16 = 3

class lakeshore.model_224.**Model224RelayControlAlarm**
Enumeration of the setting determining which alarm(s) cause a relay to activate in alarm mode.

BOTH_ALARMS = 2

HIGH_ALARM = 1

LOW_ALARM = 0

class lakeshore.model_224.**Model224RelayControlMode**
Enumeration of the configured mode of a relay.

ALARMS = 2

RELAY_OFF = 0

RELAY_ON = 1

class lakeshore.model_224.**Model224CurveFormat**
Enumerations specify formats for temperature sensor curves

LOG_OHMS_PER_KELVIN = 4

MILLIVOLT_PER_KELVIN = 1

OHMS_PER_KELVIN = 3

VOLTS_PER_KELVIN = 2

class lakeshore.model_224.**Model224CurveTemperatureCoefficients**
Enumerations specify positive/negative temperature sensor curve coefficients

NEGATIVE = 1

POSITIVE = 2

class lakeshore.model_224.**Model224SoftCalSensorTypes**
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 *lakeshore.temperature_controllers.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_

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_sta

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 conveninet to represent this functionality as a register.

bit_names = ['invalid_reading', '', '', '', 'temperature_under_range', 'temperature_ov

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
from lakeshore.model_240 import Model240InputParameter, Model240SensorTypes,␣
↪Model240Units, Model240InputRange
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
```

(continues on next page)

(continued from previous page)

```
rtd_config = Model240InputParameter(Model240SensorTypes.NTC_RTD, False, False,␣
↪Model240Units.SENSOR, True,
                                    Model240InputRange.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, Model240Units, 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())

# 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, Model240Units.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 adress: " + 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 Farenheit 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*)
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.

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

__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 (Model240CurveFormat):

- Specifies the curve data format

temperature_limit (float):

- Specifies the curve temperature limit in Kelvin

coefficient (Model240TemperatureCoefficient):

- Specifies the curve temperature coefficient

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.

__init__(*sensor*, *auto_range_enable*, *current_reversal_enable*, *units*, *input_enable*, *input_range=None*)

The constructor for InputParameter class.

Args:

sensor (Model240SensorTypes):

- Specifies the type of sensor configured at the input
- Member of the Model240SensorTypes IntEnum class

auto_range_enable (bool): Specifies if autoranging is enabled

current_reversal_enable (bool):

- Specifies channel current reversal
- Current reversal is used to remove thermal EMF errors on resistive sensors.
- Always False if channel is a diode
- False = OFF, True = ON

units (Model240Units):

- Member of the Model240Units IntEnum class
- Specifies the preferred units parameter.

input_enable (bool): Specifies whether the channel is disabled or enabled

input_range (Model240InputRange): Specifies channel range when autorange is off

class lakeshore.model_240.**Model240ProfiSlot**(*channel*, *temp_unit*)
Class used to configure and retrieve data for given PROFIBUS slot.

__init__(*channel*, *temp_unit*)
The constructor for Model240ProfiSlot class.
Args:

channel (int): Specifies which slot to configure (1-8)

temp_unit (Model240Units):

- Member of Model240Units IntEnum class
- Specifies the units to use for the data in this slot

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.**Model240Units**
Enumerations that specify temperature units

CELSIUS = 2

FAHRENHEIT = 4

KELVIN = 1

SENSOR = 3

class lakeshore.model_240.**Model240CurveFormat**
Enumerations that specify temperature sensor curve format units

LOG_OHMS_PER_KELVIN = 4

OHMS_PER_KELVIN = 3

VOLTS_PER_KELVIN = 2

class lakeshore.model_240.**Model240Coefficients**
Enumerations that specify a positive or negative coefficient

NEGATIVE = 1

POSITIVE = 2

class lakeshore.model_240.**Model240SensorTypes**
Enumerations specify types of temperature sensors

DIODE = 1

NTC_RTD = 3

PLATINUM_RTD = 2

class lakeshore.model_240.**Model240BrightnessLevel**
Enumerations for the screen brightness levels

HIGH = 100

LOW = 25

MED_HIGH = 75

MED_LOW = 50

OFF = 0

class lakeshore.model_240.**Model240TemperatureCoefficient**
Enumerations specify positive/negative temperature sensor curve coefficients

NEGATIVE = 1

POSITIVE = 2

class lakeshore.model_240.**Model240InputRange**
Enumerations to specify the input range when autorange is off

RANGE_DIODE = 0

RANGE_NTCRTD_100_KIL_OHMS = 8

RANGE_NTCRTD_100_OHMS = 2

RANGE_NTCRTD_10_KIL_OHMS = 6

RANGE_NTCRTD_10_OHMS = 0

RANGE_NTCRTD_1_KIL_OHMS = 4

RANGE_NTCRTD_30_KIL_OHMS = 7

RANGE_NTCRTD_30_OHMS = 1

RANGE_NTCRTD_3_KIL_OHMS = 5

RANGE_PTRTD_1_KIL_OHMS = 0

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.

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*)
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.

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
```

(continues on next page)

(continued from previous page)

```
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.
# 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)
- "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"
- "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"
- "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 milliamps.

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 a SCPI command or multiple commands to the instrument
Args:

commands (str): Any number of SCPI commands.

Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.

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*)
Send a SCPI query or multiple queries to the instrument and return the response(s)
Args:

queries (str): Any number of SCPI queries or commands.
Kwargs:

check_errors (bool): Chooses whether to query the SCPI error queue and raise errors as exceptions. True by default.
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.

Python Module Index

l

lakeshore.fast_hall_controller, 23
lakeshore.model_121, 192
lakeshore.model_155, 194
lakeshore.model_224, 165
lakeshore.model_240, 186
lakeshore.model_335, 63
lakeshore.model_336, 95
lakeshore.model_350, 126
lakeshore.model_372, 129
lakeshore.model_425, 19
lakeshore.model_643, 20
lakeshore.model_648, 21
lakeshore.ssm_measure_module, 53
lakeshore.ssm_settings_profiles, 59
lakeshore.ssm_source_module, 47
lakeshore.ssm_system, 42
lakeshore.teslameter, 10

Index

Symbols

__init__() (*lakeshore.fast_hall_controller.ContactCheckManualParameters method*), 30
__init__() (*lakeshore.fast_hall_controller.ContactCheckOptimizedParameters method*), 31
__init__() (*lakeshore.fast_hall_controller.DCHallParameters method*), 36
__init__() (*lakeshore.fast_hall_controller.FastHallLinkParameters method*), 33
__init__() (*lakeshore.fast_hall_controller.FastHallManualParameters method*), 32
__init__() (*lakeshore.fast_hall_controller.FourWireParameters method*), 34
__init__() (*lakeshore.fast_hall_controller.ResistivityLinkParameters method*), 39
__init__() (*lakeshore.fast_hall_controller.ResistivityManualParameters method*), 38
__init__() (*lakeshore.fast_hall_controller.StandardEventRegister method*), 40
__init__() (*lakeshore.fast_hall_controller.StatusByteRegister method*), 40
__init__() (*lakeshore.model_224.Model224AlarmParameters method*), 179
__init__() (*lakeshore.model_224.Model224CurveHeader method*), 180
__init__() (*lakeshore.model_224.Model224InputSensorSettings method*), 179
__init__() (*lakeshore.model_240.Model240CurveHeader method*), 190
__init__() (*lakeshore.model_240.Model240InputParameter method*), 190
__init__() (*lakeshore.model_240.Model240ProfiSlot method*), 191
__init__() (*lakeshore.model_335.Model335ControlLoopZoneSettings method*), 86
__init__() (*lakeshore.model_335.Model335InputSensorSettings method*), 85
__init__() (*lakeshore.model_336.Model336ControlLoopZoneSettings method*), 120
__init__() (*lakeshore.model_336.Model336InputSensorSettings method*), 119
__init__() (*lakeshore.model_372.CurveHeader method*), 157
__init__() (*lakeshore.model_372.Model372AlarmParameters method*), 157
__init__() (*lakeshore.model_372.Model372ControlLoopZoneSettings method*), 156
__init__() (*lakeshore.model_372.Model372HeaterOutputSettings method*), 155
__init__() (*lakeshore.model_372.Model372InputChannelSettings method*), 154
__init__() (*lakeshore.model_372.Model372InputSetupSettings method*), 155
__init__() (*lakeshore.temperature_controllers.AlarmSettings method*), 121
__init__() (*lakeshore.temperature_controllers.CurveHeader method*), 121
__init__() (*lakeshore.teslameter.StandardEventRegister method*), 19
__init__() (*lakeshore.teslameter.StatusByteRegister method*), 19

A

ACQUIRING_ADDRESS (*lakeshore.temperature_controllers.LanStatus attribute*), 125
ACTIVE_SCAN_CHANNEL (*lakeshore.model_372.Model372DisplayInfo attribute*), 161
ADDRESS_NOT_ACQUIRED_ERROR (*lakeshore.temperature_controllers.LanStatus attribute*), 125
ALARMS (*lakeshore.model_224.Model224RelayControlMode attribute*), 183
ALARMS (*lakeshore.model_372.Model372RelayControlMode attribute*), 160
ALARMS (*lakeshore.temperature_controllers.RelayControlMode attribute*), 87

AlarmSettings (*class in lakeshore.temperature_controllers*), 121
all_heaters_off() (*lakeshore.model_335.Model335 method*), 72
all_heaters_off() (*lakeshore.model_336.Model336 method*), 106
ALL_INPUTS (*lakeshore.model_224.Model224DisplayMode attribute*), 182
ALL_INPUTS (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
all_off() (*lakeshore.model_372.Model372 method*), 132
apply_ac_current() (*lakeshore.ssm_source_module.SourceModule method*), 50
apply_ac_voltage() (*lakeshore.ssm_source_module.SourceModule method*), 52
apply_dc_current() (*lakeshore.ssm_source_module.SourceModule method*), 50
apply_dc_voltage() (*lakeshore.ssm_source_module.SourceModule method*), 52
AUTO_IP (*lakeshore.temperature_controllers.LanStatus attribute*), 125
AUTO_OFF (*lakeshore.model_335.Model335WarmupControl attribute*), 90
AUTO_OFF (*lakeshore.temperature_controllers.ControlTypes attribute*), 91
auto_phase() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
AutotuneMode (*class in lakeshore.temperature_controllers*), 88

B

BIPOLAR (*lakeshore.temperature_controllers.Polarity attribute*), 89
bit_names (*lakeshore.model_224.Model224ReadingStatusRegister attribute*), 184
bit_names (*lakeshore.model_224.Model224ServiceRequestEnable attribute*), 184
bit_names (*lakeshore.model_224.Model224StatusByteRegister attribute*), 184
bit_names (*lakeshore.model_336.Model336InputReadingStatus attribute*), 126
bit_names (*lakeshore.model_336.Model336ServiceRequestEnable attribute*), 126
bit_names (*lakeshore.model_336.Model336StatusByteRegister attribute*), 126
BOTH_ALARMS (*lakeshore.model_224.Model224RelayControlAlarm attribute*), 183
BOTH_ALARMS (*lakeshore.temperature_controllers.RelayControlAlarm attribute*), 87
BrightnessLevel (*class in lakeshore.temperature_controllers*), 90

C

CABLE_UNPLUGGED (*lakeshore.temperature_controllers.LanStatus attribute*), 125
CAPACITANCE (*lakeshore.model_336.Model336InputSensorType attribute*), 123
CELSIUS (*lakeshore.model_224.Model224DisplayFieldUnits attribute*), 182
CELSIUS (*lakeshore.model_224.Model224InputSensorUnits attribute*), 180
CELSIUS (*lakeshore.model_240.Model240Units attribute*), 191
CELSIUS (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 91
CELSIUS (*lakeshore.model_335.Model335MonitorOutUnits attribute*), 89
CELSIUS (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
CELSIUS (*lakeshore.temperature_controllers.InputSensorUnits attribute*), 89
CHANNEL_A (*lakeshore.model_335.Model335InputSensor attribute*), 88
CHANNEL_A (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_B (*lakeshore.model_335.Model335InputSensor attribute*), 89
CHANNEL_B (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_C (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_D (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_D2 (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_D3 (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_D4 (*lakeshore.model_336.Model336InputChannel attribute*), 122
CHANNEL_D5 (*lakeshore.model_336.Model336InputChannel attribute*), 122
channel_index (*lakeshore.ssm_system.SSMSystem attribute*), 44
clear_interface() (*lakeshore.model_372.Model372 method*), 129
clear_interface_command() (*lakeshore.model_224.Model224 method*), 166
clear_interface_command() (*lakeshore.model_335.Model335 method*),

74
clear_interface_command() (*lakeshore.model_336.Model336 method*), 107
clear_interface_command() (*lakeshore.model_372.Model372 method*), 143
CLOSED_LOOP (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
CLOSED_LOOP (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123
CLOSED_LOOP (*lakeshore.model_372.Model372OutputMode attribute*), 159
command() (*lakeshore.fast_hall_controller.FastHall method*), 26
command() (*lakeshore.model_121.Model121 method*), 192
command() (*lakeshore.model_155.PrecisionSource method*), 196
command() (*lakeshore.model_224.Model224 method*), 165
command() (*lakeshore.model_240.Model240 method*), 189
command() (*lakeshore.model_335.Model335 method*), 74
command() (*lakeshore.model_336.Model336 method*), 107
command() (*lakeshore.model_350.Model350 method*), 126
command() (*lakeshore.model_372.Model372 method*), 144
command() (*lakeshore.model_425.Model425 method*), 19
command() (*lakeshore.model_643.Model643 method*), 20
command() (*lakeshore.model_648.Model648 method*), 21
command() (*lakeshore.ssm_system.SSMSystem method*), 44
command() (*lakeshore.teslameter.Teslameter method*), 16
configure_analog_heater() (*lakeshore.model_372.Model372 method*), 141
configure_analog_monitor_output_heater() (*lakeshore.model_372.Model372 method*), 141
configure_cmr() (*lakeshore.ssm_source_module.SourceModule method*), 49
configure_corrected_analog_output_scaling() (*lakeshore.teslameter.Teslameter method*), 14
configure_display() (*lakeshore.model_224.Model224 method*), 177
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*), 12
configure_heater() (*lakeshore.model_372.Model372 method*), 135
configure_i_range() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
configure_i_range() (*lakeshore.ssm_source_module.SourceModule method*), 50
configure_input() (*lakeshore.model_224.Model224 method*), 175
configure_input() (*lakeshore.model_372.Model372 method*), 131
configure_input_highpass_filter() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
configure_input_lowpass_filter() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
configure_mon_out() (*lakeshore.ssm_system.SSMSystem method*), 44
configure_mon_out_manual_mode() (*lakeshore.ssm_system.SSMSystem method*), 44
configure_qualifier() (*lakeshore.teslameter.Teslameter method*), 15
configure_ref_out() (*lakeshore.ssm_system.SSMSystem method*), 43
configure_sync() (*lakeshore.ssm_source_module.SourceModule method*), 48
configure_temperature_compensation() (*lakeshore.teslameter.Teslameter method*), 12
configure_voltage_range() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
configure_voltage_range() (*lakeshore.ssm_source_module.SourceModule method*), 51
connect_tcp() (*lakeshore.fast_hall_controller.FastHall method*), 27

connect_tcp() (*lakeshore.model_121.Model121 method*), 192
connect_tcp() (*lakeshore.model_155.PrecisionSource method*), 197
connect_tcp() (*lakeshore.model_224.Model224 method*), 178
connect_tcp() (*lakeshore.model_240.Model240 method*), 189
connect_tcp() (*lakeshore.model_335.Model335 method*), 74
connect_tcp() (*lakeshore.model_336.Model336 method*), 108
connect_tcp() (*lakeshore.model_350.Model350 method*), 126
connect_tcp() (*lakeshore.model_372.Model372 method*), 144
connect_tcp() (*lakeshore.model_425.Model425 method*), 19
connect_tcp() (*lakeshore.model_643.Model643 method*), 20
connect_tcp() (*lakeshore.model_648.Model648 method*), 21
connect_tcp() (*lakeshore.ssm_system.SSMSystem method*), 44
connect_tcp() (*lakeshore.teslameter.Teslameter method*), 16
connect_usb() (*lakeshore.fast_hall_controller.FastHall method*), 27
connect_usb() (*lakeshore.model_121.Model121 method*), 193
connect_usb() (*lakeshore.model_155.PrecisionSource method*), 197
connect_usb() (*lakeshore.model_224.Model224 method*), 178
connect_usb() (*lakeshore.model_240.Model240 method*), 189
connect_usb() (*lakeshore.model_335.Model335 method*), 74
connect_usb() (*lakeshore.model_336.Model336 method*), 108
connect_usb() (*lakeshore.model_350.Model350 method*), 126
connect_usb() (*lakeshore.model_372.Model372 method*), 144
connect_usb() (*lakeshore.model_425.Model425 method*), 19
connect_usb() (*lakeshore.model_643.Model643 method*), 20
connect_usb() (*lakeshore.model_648.Model648 method*), 21
connect_usb() (*lakeshore.ssm_system.SSMSystem method*), 44
connect_usb() (*lakeshore.teslameter.Teslameter method*), 16
ContactCheckManualParameters (*class in lakeshore.fast_hall_controller*), 29
ContactCheckOptimizedParameters (*class in lakeshore.fast_hall_controller*), 31
continue_dc_hall() (*lakeshore.fast_hall_controller.FastHall method*), 24
CONTINUOUS (*lakeshore.model_335.Model335WarmupControl attribute*), 90
CONTINUOUS (*lakeshore.temperature_controllers.ControlTypes attribute*), 91
CONTROL (*lakeshore.model_372.Model372InputChannel attribute*), 159
CONTROL_INPUT (*lakeshore.model_372.Model372DisplayMode attribute*), 160
ControlTypes (*class in lakeshore.temperature_controllers*), 91
create() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
CS_NEG (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
CS_POS (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
CURRENT (*lakeshore.model_335.Model335HeaterOutputDisplay attribute*), 90
CURRENT (*lakeshore.model_335.Model335HeaterOutType attribute*), 90
CURRENT (*lakeshore.model_372.Model372AutoRangeMode attribute*), 160
CURRENT (*lakeshore.model_372.Model372SensorExcitationMode attribute*), 160
CURRENT (*lakeshore.temperature_controllers.HeaterOutputUnits attribute*), 88
CurveFormat (*class in lakeshore.temperature_controllers*), 88
CurveHeader (*class in lakeshore.model_372*), 157
CurveHeader (*class in lakeshore.temperature_controllers*), 121
CurveTemperatureCoefficient (*class in lakeshore.temperature_controllers*), 88
CUSTOM (*lakeshore.model_224.Model224DisplayMode attribute*), 182
CUSTOM (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
CUSTOM (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
CUSTOM (*lakeshore.model_372.Model372DisplayMode attribute*), 160

D

DCHallParameters (*class in lakeshore.fast_hall_controller*), 36
delete() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 60

delete_all() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 60
delete_curve() (*lakeshore.model_224.Model224 method*), 172
delete_curve() (*lakeshore.model_240.Model240 method*), 186
delete_curve() (*lakeshore.model_335.Model335 method*), 74
delete_curve() (*lakeshore.model_336.Model336 method*), 108
delete_curve() (*lakeshore.model_372.Model372 method*), 144
DHCP (*lakeshore.temperature_controllers.LanStatus attribute*), 125
DIODE (*lakeshore.model_224.Model224InputSensorType attribute*), 180
DIODE (*lakeshore.model_240.Model240SensorTypes attribute*), 191
DIODE (*lakeshore.model_335.Model335InputSensorType attribute*), 89
DIODE (*lakeshore.model_336.Model336InputSensorType attribute*), 123
DiodeCurrent (*class in lakeshore.temperature_controllers*), 91
disable() (*lakeshore.ssm_source_module.SourceModule method*), 47
disable_ac_high_voltage_compliance() (*lakeshore.model_155.PrecisionSource method*), 196
disable_autorange() (*lakeshore.model_155.PrecisionSource method*), 195
disable_cmr() (*lakeshore.ssm_source_module.SourceModule method*), 49
disable_guards() (*lakeshore.ssm_source_module.SourceModule method*), 49
disable_high_frequency_filters() (*lakeshore.teslameter.Teslameter method*), 14
disable_input() (*lakeshore.model_224.Model224 method*), 175
disable_input() (*lakeshore.model_372.Model372 method*), 131
disable_input_filters() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
disable_lock_in_fir() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
disable_mon_out() (*lakeshore.ssm_system.SSMSystem method*), 44
disable_output() (*lakeshore.model_155.PrecisionSource method*), 194
disable_qualifier() (*lakeshore.teslameter.Teslameter method*), 15
disable_qualifier_latching() (*lakeshore.teslameter.Teslameter method*), 15
disable_ref_out() (*lakeshore.ssm_system.SSMSystem method*), 43
DISABLED (*lakeshore.model_335.Model335InputSensorType attribute*), 89
DISABLED (*lakeshore.model_336.Model336InputSensorType attribute*), 123
disconnect_tcp() (*lakeshore.fast_hall_controller.FastHall method*), 27
disconnect_tcp() (*lakeshore.model_121.Model121 method*), 193
disconnect_tcp() (*lakeshore.model_155.PrecisionSource method*), 197
disconnect_tcp() (*lakeshore.model_224.Model224 method*), 179
disconnect_tcp() (*lakeshore.model_240.Model240 method*), 189
disconnect_tcp() (*lakeshore.model_335.Model335 method*), 74
disconnect_tcp() (*lakeshore.model_336.Model336 method*), 108
disconnect_tcp() (*lakeshore.model_350.Model350 method*), 126
disconnect_tcp() (*lakeshore.model_372.Model372 method*), 144
disconnect_tcp() (*lakeshore.model_425.Model425 method*), 19
disconnect_tcp() (*lakeshore.model_643.Model643 method*), 20
disconnect_tcp() (*lakeshore.model_648.Model648 method*), 21
disconnect_tcp() (*lakeshore.ssm_system.SSMSystem method*), 45
disconnect_tcp() (*lakeshore.teslameter.Teslameter method*), 16
disconnect_usb() (*lakeshore.fast_hall_controller.FastHall method*), 27
disconnect_usb() (*lakeshore.model_121.Model121 method*), 193
disconnect_usb() (*lakeshore.model_155.PrecisionSource method*), 197
disconnect_usb() (*lakeshore.model_224.Model224 method*), 179
disconnect_usb() (*lakeshore.model_240.Model240 method*), 189
disconnect_usb() (*lakeshore.model_335.Model335 method*), 74
disconnect_usb() (*lakeshore.model_336.Model336 method*), 108
disconnect_usb() (*lakeshore.model_350.Model350*

method), 126
disconnect_usb() (*lakeshore.model_372.Model372 method*), 144
disconnect_usb() (*lakeshore.model_425.Model425 method*), 19
disconnect_usb() (*lakeshore.model_643.Model643 method*), 20
disconnect_usb() (*lakeshore.model_648.Model648 method*), 21
disconnect_usb() (*lakeshore.ssm_system.SSMSystem method*), 45
disconnect_usb() (*lakeshore.teslameter.Teslameter method*), 16
DisplayFields (*class in lakeshore.temperature_controllers*), 125
DisplayFieldsSize (*class in lakeshore.temperature_controllers*), 125
DT_400 (*lakeshore.model_224.Model224SoftCalSensorTypes attribute*), 183
DUPLICATE_INITIAL_IP_ERROR (*lakeshore.temperature_controllers.LanStatus attribute*), 125
DUPLICATE_ONGOING_IP_ERROR (*lakeshore.temperature_controllers.LanStatus attribute*), 125

E

EIGHT (*lakeshore.model_372.Model372InputChannel attribute*), 159
ELEVEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
enable() (*lakeshore.ssm_source_module.SourceModule method*), 47
enable_ac_high_voltage_compliance() (*lakeshore.model_155.PrecisionSource method*), 196
enable_autorange() (*lakeshore.model_155.PrecisionSource method*), 195
enable_cmr() (*lakeshore.ssm_source_module.SourceModule method*), 49
enable_guards() (*lakeshore.ssm_source_module.SourceModule method*), 49
enable_high_frequency_filters() (*lakeshore.teslameter.Teslameter method*), 14
enable_lock_in_fir() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
enable_mon_out() (*lakeshore.ssm_system.SSMSystem method*), 44
enable_output() (*lakeshore.model_155.PrecisionSource method*), 194
enable_qualifier() (*lakeshore.teslameter.Teslameter method*), 15
enable_qualifier_latching() (*lakeshore.teslameter.Teslameter method*), 15
enable_ref_out() (*lakeshore.ssm_system.SSMSystem method*), 43
ETHERNET (*lakeshore.model_224.Model224RemoteInterface attribute*), 181
ETHERNET (*lakeshore.temperature_controllers.Interface attribute*), 125
ETHERNET_DISABLED (*lakeshore.temperature_controllers.LanStatus attribute*), 125

F

factory_reset() (*lakeshore.fast_hall_controller.FastHall method*), 27
factory_reset() (*lakeshore.model_155.PrecisionSource method*), 197
factory_reset() (*lakeshore.ssm_system.SSMSystem method*), 45
factory_reset() (*lakeshore.teslameter.Teslameter method*), 16
FAHRENHEIT (*lakeshore.model_240.Model240Units attribute*), 191
FastHall (*class in lakeshore.fast_hall_controller*), 23
FastHallLinkParameters (*class in lakeshore.fast_hall_controller*), 33
FastHallManualParameters (*class in lakeshore.fast_hall_controller*), 31
FastHallOperationRegister (*class in lakeshore.fast_hall_controller*), 29
FastHallQuestionableRegister (*class in lakeshore.fast_hall_controller*), 29
FIFTEEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
FIFTY_MILLIVOLT (*lakeshore.model_335.Model335ThermocoupleRang attribute*), 90
FIFTY_MILLIVOLT (*lakeshore.model_336.Model336ThermocoupleRang attribute*), 123
FIVE (*lakeshore.model_372.Model372InputChannel attribute*), 159
FOUR (*lakeshore.model_372.Model372InputChannel attribute*), 159
FOUR_LOOP (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
FOURTEEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
FourWireParameters (*class in lakeshore.fast_hall_controller*), 34
FREQUENCY_11_POINT_6_HZ (*lakeshore.model_372.Model372InputFrequency attribute*), 161

FREQUENCY_13_POINT_7_HZ (*lakeshore.model_372.Model372InputFrequency attribute*), 161
FREQUENCY_16_POINT_2_HZ (*lakeshore.model_372.Model372InputFrequency attribute*), 161
FREQUENCY_18_POINT_2_HZ (*lakeshore.model_372.Model372InputFrequency attribute*), 161
FREQUENCY_9_POINT_8_HZ (*lakeshore.model_372.Model372InputFrequency attribute*), 161
FULL (*lakeshore.temperature_controllers.BrightnessLevel attribute*), 90

G

generate_and_apply_soft_cal_curve() (*lakeshore.model_224.Model224 method*), 173
get_alarm_beep_status() (*lakeshore.model_372.Model372 method*), 136
get_alarm_parameters() (*lakeshore.model_224.Model224 method*), 171
get_alarm_parameters() (*lakeshore.model_335.Model335 method*), 74
get_alarm_parameters() (*lakeshore.model_336.Model336 method*), 108
get_alarm_parameters() (*lakeshore.model_372.Model372 method*), 139
get_alarm_status() (*lakeshore.model_224.Model224 method*), 171
get_alarm_status() (*lakeshore.model_335.Model335 method*), 74
get_alarm_status() (*lakeshore.model_336.Model336 method*), 108
get_alarm_status() (*lakeshore.model_372.Model372 method*), 144
get_all_input_readings() (*lakeshore.model_372.Model372 method*), 130
get_all_inputs_celsius_reading() (*lakeshore.model_224.Model224 method*), 167
get_all_kelvin_reading() (*lakeshore.model_335.Model335 method*), 71
get_all_kelvin_reading() (*lakeshore.model_336.Model336 method*), 104
get_all_sensor_reading() (*lakeshore.model_336.Model336 method*), 106
get_analog_heater_output() (*lakeshore.model_372.Model372 method*), 132
get_analog_manual_value() (*lakeshore.model_372.Model372 method*), 142
get_analog_monitor_output_settings() (*lakeshore.model_372.Model372 method*), 141
get_analog_output() (*lakeshore.teslameter.Teslameter method*), 14
get_analog_output_percentage() (*lakeshore.model_335.Model335 method*), 63
get_analog_output_percentage() (*lakeshore.model_336.Model336 method*), 95
get_analog_output_signal() (*lakeshore.teslameter.Teslameter method*), 14
get_averaging_time() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_band_pass_filter_center() (*lakeshore.teslameter.Teslameter method*), 15
get_bias_voltage() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_brightness() (*lakeshore.model_240.Model240 method*), 186
get_brightness() (*lakeshore.model_335.Model335 method*), 64
get_buffered_data_points() (*lakeshore.teslameter.Teslameter method*), 11
get_celsius_reading() (*lakeshore.model_224.Model224 method*), 167
get_celsius_reading() (*lakeshore.model_240.Model240 method*), 186
get_celsius_reading() (*lakeshore.model_335.Model335 method*), 69
get_celsius_reading() (*lakeshore.model_336.Model336 method*), 100

get_channel_reading_status() (*lakeshore.model_240.Model240 method*), 189
get_cmr_source() (*lakeshore.ssm_source_module.SourceModule method*), 49
get_cmr_state() (*lakeshore.ssm_source_module.SourceModule method*), 49
get_common_mode_reduction() (*lakeshore.model_372.Model372 method*), 135
get_contact_check_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_contact_check_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 23
get_contact_check_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_contrast_level() (*lakeshore.model_336.Model336 method*), 95
get_control_loop_zone_parameters() (*lakeshore.model_372.Model372 method*), 142
get_control_loop_zone_table() (*lakeshore.model_335.Model335 method*), 73
get_control_loop_zone_table() (*lakeshore.model_336.Model336 method*), 107
get_control_setpoint() (*lakeshore.model_335.Model335 method*), 75
get_control_setpoint() (*lakeshore.model_336.Model336 method*), 108
get_control_setpoint() (*lakeshore.model_372.Model372 method*), 144
get_corrected_analog_output_scaling() (*lakeshore.teslameter.Teslameter method*), 14
get_coupling() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_coupling() (*lakeshore.ssm_source_module.SourceModule method*), 49
get_curve() (*lakeshore.model_224.Model224 method*), 173
get_curve() (*lakeshore.model_335.Model335 method*), 75
get_curve() (*lakeshore.model_336.Model336 method*), 109
get_curve() (*lakeshore.model_372.Model372 method*), 145
get_curve_data_point() (*lakeshore.model_224.Model224 method*), 172
get_curve_data_point() (*lakeshore.model_240.Model240 method*), 187
get_curve_data_point() (*lakeshore.model_335.Model335 method*), 75
get_curve_data_point() (*lakeshore.model_336.Model336 method*), 109
get_curve_data_point() (*lakeshore.model_372.Model372 method*), 145
get_curve_header() (*lakeshore.model_224.Model224 method*), 172
get_curve_header() (*lakeshore.model_240.Model240 method*), 187
get_curve_header() (*lakeshore.model_335.Model335 method*), 75
get_curve_header() (*lakeshore.model_336.Model336 method*), 109
get_curve_header() (*lakeshore.model_372.Model372 method*), 145
get_custom_display_settings() (*lakeshore.model_372.Model372 method*), 130
get_dark_mode_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 59
get_dark_mode_state() (*lakeshore.ssm_source_module.SourceModule method*), 53
get_data() (*lakeshore.ssm_system.SSMSystem method*), 42
get_dc() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
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*), 25
get_dc_hall_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 24
get_dc_hall_setup_results()

(*lakeshore.fast_hall_controller.FastHall method*), 25
get_dc_hall_waiting_status() (*lakeshore.fast_hall_controller.FastHall method*), 24
get_description() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
get_digital_output() (*lakeshore.model_372.Model372 method*), 138
get_diode_excitation_current() (*lakeshore.model_335.Model335 method*), 66
get_diode_excitation_current() (*lakeshore.model_336.Model336 method*), 101
get_display_configuration() (*lakeshore.model_224.Model224 method*), 177
get_display_contrast() (*lakeshore.model_224.Model224 method*), 169
get_display_field_settings() (*lakeshore.model_224.Model224 method*), 176
get_display_field_settings() (*lakeshore.model_335.Model335 method*), 76
get_display_field_settings() (*lakeshore.model_336.Model336 method*), 109
get_display_field_settings() (*lakeshore.model_372.Model372 method*), 145
get_display_mode() (*lakeshore.model_372.Model372 method*), 130
get_display_setup() (*lakeshore.model_335.Model335 method*), 69
get_display_setup() (*lakeshore.model_336.Model336 method*), 103
get_duty() (*lakeshore.ssm_source_module.SourceModule method*), 48
get_enable_state() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_excitation_frequency() (*lakeshore.model_372.Model372 method*), 138
get_excitation_mode() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_excitation_power() (*lakeshore.model_372.Model372 method*), 134
get_fahrenheit_reading() (*lakeshore.model_240.Model240 method*), 186
get_fasthall_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_fasthall_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 23
get_fasthall_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
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*), 175
get_filter() (*lakeshore.model_240.Model240 method*), 187
get_filter() (*lakeshore.model_335.Model335 method*), 68
get_filter() (*lakeshore.model_336.Model336 method*), 98
get_filter() (*lakeshore.model_372.Model372 method*), 133
get_filter_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_four_wire_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_four_wire_running_status() (*lakeshore.fast_hall_controller.FastHall*

method), 23
get_four_wire_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_frequency() (*lakeshore.ssm_source_module.SourceModule method*), 48
get_frequency() (*lakeshore.teslameter.Teslameter method*), 11
get_frequency_filter_type() (*lakeshore.teslameter.Teslameter method*), 15
get_gain_allocation_strategy() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_guard_state() (*lakeshore.ssm_source_module.SourceModule method*), 49
get_head_self_cal_status() (*lakeshore.ssm_system.SSMSystem method*), 44
get_heater_output() (*lakeshore.model_335.Model335 method*), 76
get_heater_output() (*lakeshore.model_336.Model336 method*), 110
get_heater_output() (*lakeshore.model_372.Model372 method*), 146
get_heater_output_mode() (*lakeshore.model_335.Model335 method*), 71
get_heater_output_mode() (*lakeshore.model_336.Model336 method*), 105
get_heater_output_range() (*lakeshore.model_372.Model372 method*), 133
get_heater_output_settings() (*lakeshore.model_372.Model372 method*), 134
get_heater_pid() (*lakeshore.model_335.Model335 method*), 76
get_heater_pid() (*lakeshore.model_336.Model336 method*), 110
get_heater_pid() (*lakeshore.model_372.Model372 method*), 146
get_heater_range() (*lakeshore.model_335.Model335 method*), 72
get_heater_range() (*lakeshore.model_336.Model336 method*), 105
get_heater_setup() (*lakeshore.model_335.Model335 method*), 70
get_heater_setup() (*lakeshore.model_336.Model336 method*), 103
get_heater_status() (*lakeshore.model_335.Model335 method*), 77
get_heater_status() (*lakeshore.model_336.Model336 method*), 110
get_heater_status() (*lakeshore.model_372.Model372 method*), 146
get_high_pass_filter_cutoff() (*lakeshore.teslameter.Teslameter method*), 15
get_highpass_corner_frequency() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_highpass_rolloff() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_hw_version() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_hw_version() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_i_ac_range() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_i_amplitude() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_i_autorange_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_i_autorange_status() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_i_dc_range() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_i_limit() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_i_limit_status() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_i_offset() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_i_range() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_i_range() (*lakeshore.ssm_source_module.SourceModule method*), 50
get_identification() (*lakeshore.model_240.Model240 method*), 186
get_identify_state()

(*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_identify_state() (*lakeshore.ssm_source_module.SourceModule method*), 53
get_ieee_488() (*lakeshore.model_224.Model224 method*), 169
get_ieee_488() (*lakeshore.model_335.Model335 method*), 77
get_ieee_488() (*lakeshore.model_336.Model336 method*), 110
get_ieee_488() (*lakeshore.model_372.Model372 method*), 146
get_ieee_interface_mode() (*lakeshore.model_372.Model372 method*), 140
get_ieee_interface_parameter() (*lakeshore.model_372.Model372 method*), 134
get_input_channel_parameters() (*lakeshore.model_372.Model372 method*), 131
get_input_configuration() (*lakeshore.model_224.Model224 method*), 175
get_input_configuration() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_input_curve() (*lakeshore.model_224.Model224 method*), 170
get_input_curve() (*lakeshore.model_335.Model335 method*), 77
get_input_curve() (*lakeshore.model_336.Model336 method*), 111
get_input_curve() (*lakeshore.model_372.Model372 method*), 146
get_input_diode_excitation_current() (*lakeshore.model_224.Model224 method*), 168
get_input_parameter() (*lakeshore.model_240.Model240 method*), 188
get_input_reading_status() (*lakeshore.model_335.Model335 method*), 72
get_input_reading_status() (*lakeshore.model_336.Model336 method*), 106
get_input_sensor() (*lakeshore.model_335.Model335 method*), 70
get_input_sensor() (*lakeshore.model_336.Model336 method*), 104
get_input_setup_parameters() (*lakeshore.model_372.Model372 method*), 131
get_interface() (*lakeshore.model_336.Model336 method*), 100
get_interface() (*lakeshore.model_372.Model372 method*), 139
get_interface_mode() (*lakeshore.model_224.Model224 method*), 176
get_json() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
get_kelvin_reading() (*lakeshore.model_224.Model224 method*), 167
get_kelvin_reading() (*lakeshore.model_240.Model240 method*), 186
get_kelvin_reading() (*lakeshore.model_335.Model335 method*), 77
get_kelvin_reading() (*lakeshore.model_336.Model336 method*), 111
get_kelvin_reading() (*lakeshore.model_372.Model372 method*), 147
get_keypad_lock() (*lakeshore.model_224.Model224 method*), 170
get_keypad_lock() (*lakeshore.model_335.Model335 method*), 77
get_keypad_lock() (*lakeshore.model_336.Model336 method*), 111
get_keypad_lock() (*lakeshore.model_372.Model372 method*), 147
get_led_state() (*lakeshore.model_224.Model224 method*), 169
get_led_state() (*lakeshore.model_335.Model335 method*), 77
get_led_state() (*lakeshore.model_336.Model336 method*), 111
get_led_state() (*lakeshore.model_372.Model372 method*), 147
get_list() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
get_lock_in_equivalent_noise_bandwidth() (*lakeshore.ssm_measure_module.MeasureModule*

method), 56
get_lock_in_fir_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
get_lock_in_frequency() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_lock_in_r() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_lock_in_rolloff() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_lock_in_settle_time() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_lock_in_theta() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_lock_in_time_constant() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_lock_in_x() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_lock_in_y() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_low_pass_filter_cutoff() (*lakeshore.teslameter.Teslameter method*), 15
get_lowpass_corner_frequency() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_lowpass_rolloff() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_manual_output() (*lakeshore.model_335.Model335 method*), 78
get_manual_output() (*lakeshore.model_336.Model336 method*), 111
get_manual_output() (*lakeshore.model_372.Model372 method*), 147
get_max_min() (*lakeshore.teslameter.Teslameter method*), 11
get_max_min_peaks() (*lakeshore.teslameter.Teslameter method*), 11
get_measure_module() (*lakeshore.ssm_system.SSMSystem method*), 42
get_measure_module_by_name() (*lakeshore.ssm_system.SSMSystem method*), 42
get_measure_pod() (*lakeshore.ssm_system.SSMSystem method*), 42
get_min_max_data() (*lakeshore.model_224.Model224 method*), 170
get_min_max_data() (*lakeshore.model_335.Model335 method*), 78
get_min_max_data() (*lakeshore.model_336.Model336 method*), 112
get_min_max_data() (*lakeshore.model_372.Model372 method*), 147
get_mode() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
get_model() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_model() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_modname() (*lakeshore.model_240.Model240 method*), 188
get_mon_out_manual_level() (*lakeshore.ssm_system.SSMSystem method*), 44
get_mon_out_mode() (*lakeshore.ssm_system.SSMSystem method*), 43
get_mon_out_state() (*lakeshore.ssm_system.SSMSystem method*), 43
get_monitor_output_heater() (*lakeshore.model_335.Model335 method*), 68
get_monitor_output_heater() (*lakeshore.model_336.Model336 method*), 102
get_monitor_output_source() (*lakeshore.model_372.Model372 method*), 140
get_multiple() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
get_multiple() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_multiple() (*lakeshore.ssm_system.SSMSystem method*), 42
get_name() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_name() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_negative_peak() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_network_configuration() (*lakeshore.model_336.Model336 method*), 99
get_network_settings() (*lakeshore.model_336.Model336 method*), 99

get_num_measure_channels() (*lakeshore.ssm_system.SSMSystem method*), 42
get_num_source_channels() (*lakeshore.ssm_system.SSMSystem method*), 42
get_operation_condition() (*lakeshore.model_335.Model335 method*), 64
get_operation_condition() (*lakeshore.model_336.Model336 method*), 95
get_operation_event() (*lakeshore.model_335.Model335 method*), 64
get_operation_event() (*lakeshore.model_336.Model336 method*), 96
get_operation_event_enable() (*lakeshore.model_335.Model335 method*), 64
get_operation_event_enable() (*lakeshore.model_336.Model336 method*), 95
get_operation_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_operation_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 197
get_operation_event_enable_mask() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_operation_event_enable_mask() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_operation_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 45
get_operation_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 16
get_operation_events() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_operation_events() (*lakeshore.model_155.PrecisionSource method*), 197
get_operation_events() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_operation_events() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_operation_events() (*lakeshore.ssm_system.SSMSystem method*), 45
get_operation_events() (*lakeshore.teslameter.Teslameter method*), 16
get_output_2_polarity() (*lakeshore.model_335.Model335 method*), 72
get_output_settings() (*lakeshore.model_155.PrecisionSource method*), 195
get_peak_to_peak() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
get_pll_lock_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_positive_peak() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_present_operation_status() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_present_operation_status() (*lakeshore.model_155.PrecisionSource method*), 197
get_present_operation_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_present_operation_status() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_present_operation_status() (*lakeshore.ssm_system.SSMSystem method*), 45
get_present_operation_status() (*lakeshore.teslameter.Teslameter method*), 16
get_present_questionable_status() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_present_questionable_status() (*lakeshore.model_155.PrecisionSource method*), 197
get_present_questionable_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_present_questionable_status() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_present_questionable_status() (*lakeshore.ssm_system.SSMSystem method*), 45
get_present_questionable_status() (*lakeshore.teslameter.Teslameter method*), 16
get_probe_information() (*lakeshore.teslameter.Teslameter method*), 11

get_profibus_address() (*lakeshore.model_240.Model240 method*), 188
get_profibus_connection_status() (*lakeshore.model_240.Model240 method*), 189
get_profibus_slot_configuration() (*lakeshore.model_240.Model240 method*), 189
get_profibus_slot_count() (*lakeshore.model_240.Model240 method*), 188
get_quadrature_reading() (*lakeshore.model_372.Model372 method*), 130
get_qualifier_configuration() (*lakeshore.teslameter.Teslameter method*), 15
get_qualifier_latching_setting() (*lakeshore.teslameter.Teslameter method*), 15
get_questionable_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_questionable_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 197
get_questionable_event_enable_mask() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_questionable_event_enable_mask() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_questionable_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 45
get_questionable_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 16
get_questionable_events() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_questionable_events() (*lakeshore.model_155.PrecisionSource method*), 197
get_questionable_events() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
get_questionable_events() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_questionable_events() (*lakeshore.ssm_system.SSMSystem method*), 45
get_questionable_events() (*lakeshore.teslameter.Teslameter method*), 16
get_reading_status() (*lakeshore.model_224.Model224 method*), 166
get_reading_status() (*lakeshore.model_372.Model372 method*), 143
get_ref_in_edge() (*lakeshore.ssm_system.SSMSystem method*), 42
get_ref_out_source() (*lakeshore.ssm_system.SSMSystem method*), 43
get_ref_out_state() (*lakeshore.ssm_system.SSMSystem method*), 43
get_reference_harmonic() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_reference_phase_shift() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_reference_source() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
get_relative_field() (*lakeshore.teslameter.Teslameter method*), 11
get_relative_field_baseline() (*lakeshore.teslameter.Teslameter method*), 12
get_relay_alarm_control_parameters() (*lakeshore.model_224.Model224 method*), 178
get_relay_alarm_control_parameters() (*lakeshore.model_335.Model335 method*), 78
get_relay_alarm_control_parameters() (*lakeshore.model_336.Model336 method*), 112
get_relay_alarm_control_parameters() (*lakeshore.model_372.Model372 method*), 147
get_relay_control_mode() (*lakeshore.model_224.Model224 method*), 178
get_relay_control_mode() (*lakeshore.model_335.Model335 method*), 78
get_relay_control_mode() (*lakeshore.model_336.Model336 method*), 112
get_relay_control_mode() (*lakeshore.model_372.Model372 method*), 148
get_relay_status() (*lakeshore.model_224.Model224 method*), 174
get_relay_status() (*lakeshore.model_335.Model335 method*), 79
get_relay_status()

(*lakeshore.model_336.Model336 method*), 112
get_relay_status() (*lakeshore.model_372.Model372 method*), 148
get_remote_interface() (*lakeshore.model_224.Model224 method*), 176
get_remote_interface_mode() (*lakeshore.model_335.Model335 method*), 79
get_remote_interface_mode() (*lakeshore.model_336.Model336 method*), 113
get_remote_interface_mode() (*lakeshore.model_372.Model372 method*), 148
get_resistance_reading() (*lakeshore.model_372.Model372 method*), 130
get_resistivity_measurement_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_resistivity_running_status() (*lakeshore.fast_hall_controller.FastHall method*), 24
get_resistivity_setup_results() (*lakeshore.fast_hall_controller.FastHall method*), 25
get_rms() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
get_rms_field() (*lakeshore.teslameter.Teslameter method*), 11
get_rms_field_xyz() (*lakeshore.teslameter.Teslameter method*), 11
get_sample_heater_setup() (*lakeshore.model_372.Model372 method*), 140
get_scanner_status() (*lakeshore.model_372.Model372 method*), 135
get_self_cal_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_self_cal_status() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_self_test() (*lakeshore.model_224.Model224 method*), 166
get_self_test() (*lakeshore.model_335.Model335 method*), 79
get_self_test() (*lakeshore.model_336.Model336 method*), 113
get_self_test() (*lakeshore.model_372.Model372 method*), 148
get_sensor_name() (*lakeshore.model_224.Model224 method*), 168
get_sensor_name() (*lakeshore.model_240.Model240 method*), 188
get_sensor_name() (*lakeshore.model_335.Model335 method*), 79
get_sensor_name() (*lakeshore.model_336.Model336 method*), 113
get_sensor_name() (*lakeshore.model_372.Model372 method*), 149
get_sensor_reading() (*lakeshore.model_224.Model224 method*), 167
get_sensor_reading() (*lakeshore.model_240.Model240 method*), 186
get_sensor_reading() (*lakeshore.model_335.Model335 method*), 79
get_sensor_reading() (*lakeshore.model_336.Model336 method*), 113
get_sensor_reading() (*lakeshore.model_372.Model372 method*), 149
get_sensor_units_channel_reading() (*lakeshore.model_240.Model240 method*), 189
get_serial() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
get_serial() (*lakeshore.ssm_source_module.SourceModule method*), 47
get_service_request() (*lakeshore.model_224.Model224 method*), 166
get_service_request() (*lakeshore.model_335.Model335 method*), 79
get_service_request() (*lakeshore.model_336.Model336 method*), 113
get_service_request() (*lakeshore.model_372.Model372 method*), 149
get_service_request_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_service_request_enable_mask()

(*lakeshore.model_155.PrecisionSource method*), 197
get_service_request_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 45
get_service_request_enable_mask() (*lakeshore.teslameter.Teslameter method*), 16
get_setpoint_kelvin() (*lakeshore.model_372.Model372 method*), 137
get_setpoint_ohms() (*lakeshore.model_372.Model372 method*), 137
get_setpoint_ramp_parameter() (*lakeshore.model_335.Model335 method*), 80
get_setpoint_ramp_parameter() (*lakeshore.model_336.Model336 method*), 113
get_setpoint_ramp_parameter() (*lakeshore.model_372.Model372 method*), 149
get_setpoint_ramp_status() (*lakeshore.model_335.Model335 method*), 80
get_setpoint_ramp_status() (*lakeshore.model_336.Model336 method*), 114
get_setpoint_ramp_status() (*lakeshore.model_372.Model372 method*), 149
get_shape() (*lakeshore.ssm_source_module.SourceModule method*), 48
get_source_module() (*lakeshore.ssm_system.SSMSystem method*), 42
get_source_module_by_name() (*lakeshore.ssm_system.SSMSystem method*), 42
get_source_pod() (*lakeshore.ssm_system.SSMSystem method*), 42
get_standard_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_standard_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 197
get_standard_event_enable_mask() (*lakeshore.model_224.Model224 method*), 165
get_standard_event_enable_mask() (*lakeshore.model_335.Model335 method*), 80
get_standard_event_enable_mask() (*lakeshore.model_336.Model336 method*), 114
get_standard_event_enable_mask() (*lakeshore.model_372.Model372 method*), 149
get_standard_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 45
get_standard_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 17
get_standard_events() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_standard_events() (*lakeshore.model_155.PrecisionSource method*), 197
get_standard_events() (*lakeshore.ssm_system.SSMSystem method*), 45
get_standard_events() (*lakeshore.teslameter.Teslameter method*), 17
get_status_byte() (*lakeshore.fast_hall_controller.FastHall method*), 27
get_status_byte() (*lakeshore.model_155.PrecisionSource method*), 197
get_status_byte() (*lakeshore.model_224.Model224 method*), 166
get_status_byte() (*lakeshore.model_335.Model335 method*), 80
get_status_byte() (*lakeshore.model_336.Model336 method*), 114
get_status_byte() (*lakeshore.model_372.Model372 method*), 149
get_status_byte() (*lakeshore.ssm_system.SSMSystem method*), 45
get_status_byte() (*lakeshore.teslameter.Teslameter method*), 17
get_still_output() (*lakeshore.model_372.Model372 method*), 136
get_sync_phase_shift() (*lakeshore.ssm_source_module.SourceModule method*), 48
get_sync_source() (*lakeshore.ssm_source_module.SourceModule method*), 48
get_sync_state() (*lakeshore.ssm_source_module.SourceModule method*), 48
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_335.Model335 method*), 80
get_temperature_limit() (*lakeshore.model_336.Model336 method*), 114
get_temperature_limit() (*lakeshore.model_372.Model372 method*), 150
get_thermocouple_junction_temp() (*lakeshore.model_335.Model335 method*), 64
get_thermocouple_junction_temp() (*lakeshore.model_336.Model336 method*), 96
get_tuning_control_status() (*lakeshore.model_335.Model335 method*), 67
get_tuning_control_status() (*lakeshore.model_336.Model336 method*), 101
get_valid_for_restore() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
get_voltage_ac_range() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_voltage_amplitude() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_voltage_autorange_status() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_voltage_autorange_status() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_voltage_dc_range() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_voltage_limit() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_voltage_limit_status() (*lakeshore.ssm_source_module.SourceModule method*), 52
get_voltage_offset() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_voltage_range() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
get_voltage_range() (*lakeshore.ssm_source_module.SourceModule method*), 51
get_warmup_heater_setup() (*lakeshore.model_372.Model372 method*), 140
get_warmup_output() (*lakeshore.model_372.Model372 method*), 136
get_warmup_supply() (*lakeshore.model_335.Model335 method*), 73
get_warmup_supply_parameter() (*lakeshore.model_336.Model336 method*), 106
get_website_login() (*lakeshore.model_224.Model224 method*), 171
get_website_login() (*lakeshore.model_336.Model336 method*), 100
get_website_login() (*lakeshore.model_372.Model372 method*), 142
go_to_current_mode() (*lakeshore.ssm_source_module.SourceModule method*), 48
go_to_voltage_mode() (*lakeshore.ssm_source_module.SourceModule method*), 48

H

HALF (*lakeshore.temperature_controllers.BrightnessLevel attribute*), 90
HEATER_25_OHM (*lakeshore.temperature_controllers.HeaterResistance attribute*), 88
HEATER_50_OHM (*lakeshore.temperature_controllers.HeaterResistance attribute*), 88
HEATER_OPEN_LOAD (*lakeshore.temperature_controllers.HeaterError attribute*), 88
HEATER_SHORT (*lakeshore.temperature_controllers.HeaterError attribute*), 88
HeaterError (*class in lakeshore.temperature_controllers*), 87
HeaterOutputUnits (*class in lakeshore.temperature_controllers*), 88
HeaterResistance (*class in lakeshore.temperature_controllers*), 88
HIGH (*lakeshore.model_240.Model240BrightnessLevel attribute*), 191

HIGH (*lakeshore.model_335.Model335HeaterRange attribute*), 90
HIGH (*lakeshore.model_336.Model336HeaterRange attribute*), 123
HIGH_ALARM (*lakeshore.model_224.Model224RelayControlAlarm attribute*), 183
HIGH_ALARM (*lakeshore.temperature_controllers.RelayControlAlarm attribute*), 87
HUNDRED_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
HUNDRED_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123

I

IEEE488 (*lakeshore.temperature_controllers.Interface attribute*), 125
IEEE_488 (*lakeshore.model_224.Model224RemoteInterface attribute*), 181
INPUT_A (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_A (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_A (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
INPUT_A (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
INPUT_A (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_A_MAX_MIN (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
INPUT_B (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_B (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_B (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
INPUT_B (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
INPUT_B (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_B_MAX_MIN (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
INPUT_C (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_C (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_C (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_C2 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_C2 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_C3 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_C3 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_C4 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_C4 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_C5 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_C5 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_D1 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_D1 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D2 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_D2 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D2 (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_D3 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_D3 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D3 (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_D4 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_D4 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D4 (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_D5 (*lakeshore.model_224.Model224DisplayMode attribute*), 182
INPUT_D5 (*lakeshore.model_224.Model224InputChannel attribute*), 182
INPUT_D5 (*lakeshore.model_336.Model336DisplaySetupMode attribute*), 122
INPUT_DISABLED (*lakeshore.model_224.Model224InputSensorType attribute*), 180
InputSensorUnits (*class in lakeshore.temperature_controllers*), 89
Interface (*class in lakeshore.temperature_controllers*), 125
InterfaceMode (*class in lakeshore.temperature_controllers*), 87
is_qualifier_condition_met() (*lakeshore.teslameter.Teslameter method*), 15

K

KELVIN (*lakeshore.model_224.Model224DisplayFieldUnits*

attribute), 182
KELVIN (*lakeshore.model_224.Model224InputSensorUnits attribute*), 180
KELVIN (*lakeshore.model_240.Model240Units attribute*), 191
KELVIN (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 92
KELVIN (*lakeshore.model_335.Model335MonitorOutUnits attribute*), 89
KELVIN (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
KELVIN (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
KELVIN (*lakeshore.model_372.Model372InputSensorUnits attribute*), 160
KELVIN (*lakeshore.temperature_controllers.InputSensorUnits attribute*), 89

L

lakeshore.fast_hall_controller (*module*), 23
lakeshore.model_121 (*module*), 192
lakeshore.model_155 (*module*), 194
lakeshore.model_224 (*module*), 165
lakeshore.model_240 (*module*), 186
lakeshore.model_335 (*module*), 63
lakeshore.model_336 (*module*), 95
lakeshore.model_350 (*module*), 126
lakeshore.model_372 (*module*), 129
lakeshore.model_425 (*module*), 19
lakeshore.model_643 (*module*), 20
lakeshore.model_648 (*module*), 21
lakeshore.ssm_measure_module (*module*), 53
lakeshore.ssm_settings_profiles (*module*), 59
lakeshore.ssm_source_module (*module*), 47
lakeshore.ssm_system (*module*), 42
lakeshore.teslameter (*module*), 10
LanStatus (*class in lakeshore.temperature_controllers*), 125
LARGE (*lakeshore.temperature_controllers.DisplayFieldsSize attribute*), 125
LARGE_2 (*lakeshore.temperature_controllers.DisplayFields attribute*), 125
LARGE_4 (*lakeshore.model_224.Model224NumberOfFields attribute*), 183
LARGE_4 (*lakeshore.temperature_controllers.DisplayFields attribute*), 125
LARGE_4_SMALL_8 (*lakeshore.model_224.Model224NumberOfFields attribute*), 183
LARGE_8 (*lakeshore.model_224.Model224NumberOfFields attribute*), 183
LOCAL (*lakeshore.model_224.Model224InterfaceMode attribute*), 181
LOCAL (*lakeshore.temperature_controllers.InterfaceMode attribute*), 87
log_buffered_data_to_file() (*lakeshore.teslameter.Teslameter method*), 11
log_data_to_csv_file() (*lakeshore.ssm_system.SSMSystem method*), 42
LOG_OHMS_PER_KELVIN (*lakeshore.model_224.Model224CurveFormat attribute*), 183
LOG_OHMS_PER_KELVIN (*lakeshore.model_240.Model240CurveFormat attribute*), 191
LOG_OHMS_PER_KELVIN (*lakeshore.temperature_controllers.CurveFormat attribute*), 88
LOGOHM_PER_KELVIN (*lakeshore.model_372.Model372CurveFormat attribute*), 161
LOW (*lakeshore.model_240.Model240BrightnessLevel attribute*), 191
LOW (*lakeshore.model_335.Model335HeaterRange attribute*), 90
LOW (*lakeshore.model_336.Model336HeaterRange attribute*), 123
LOW_ALARM (*lakeshore.model_224.Model224RelayControlAlarm attribute*), 183
LOW_ALARM (*lakeshore.temperature_controllers.RelayControlAlarm attribute*), 87

M

MAXIMUM_DATA (*lakeshore.model_224.Model224DisplayFieldUnits attribute*), 182
MAXIMUM_DATA (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 92
MAXIMUM_DATA (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
MAXIMUM_DATA (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
MEASUREMENT_INPUT (*lakeshore.model_372.Model372DisplayMode attribute*), 160
MeasureModule (*class in lakeshore.ssm_measure_module*), 53
MED_HIGH (*lakeshore.model_240.Model240BrightnessLevel attribute*), 192
MED_LOW (*lakeshore.model_240.Model240BrightnessLevel attribute*), 192
MEDIUM (*lakeshore.model_335.Model335HeaterRange attribute*), 90
MEDIUM (*lakeshore.model_336.Model336HeaterRange attribute*), 124
MILLIVOLT_PER_KELVIN (*lakeshore.model_224.Model224CurveFormat*

attribute), 183
MILLIVOLT_PER_KELVIN (*lakeshore.temperature_controllers.CurveFormat attribute*), 88
MINIMUM_DATA (*lakeshore.model_224.Model224DisplayFieldUnits attribute*), 182
MINIMUM_DATA (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 92
MINIMUM_DATA (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
MINIMUM_DATA (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
Model121 (*class in lakeshore.model_121*), 192
Model224 (*class in lakeshore.model_224*), 165
Model224AlarmParameters (*class in lakeshore.model_224*), 179
Model224CurveFormat (*class in lakeshore.model_224*), 183
Model224CurveHeader (*class in lakeshore.model_224*), 180
Model224CurveTemperatureCoefficients (*class in lakeshore.model_224*), 183
Model224DiodeExcitationCurrent (*class in lakeshore.model_224*), 180
Model224DiodeSensorRange (*class in lakeshore.model_224*), 181
Model224DisplayFieldUnits (*class in lakeshore.model_224*), 181
Model224DisplayMode (*class in lakeshore.model_224*), 182
Model224InputChannel (*class in lakeshore.model_224*), 182
Model224InputSensorSettings (*class in lakeshore.model_224*), 179
Model224InputSensorType (*class in lakeshore.model_224*), 180
Model224InputSensorUnits (*class in lakeshore.model_224*), 180
Model224InterfaceMode (*class in lakeshore.model_224*), 181
Model224NTCRTDSensorResistanceRange (*class in lakeshore.model_224*), 181
Model224NumberOfFields (*class in lakeshore.model_224*), 183
Model224PlatinumRTDSensorResistanceRange (*class in lakeshore.model_224*), 181
Model224ReadingStatusRegister (*class in lakeshore.model_224*), 184
Model224RelayControlAlarm (*class in lakeshore.model_224*), 183
Model224RelayControlMode (*class in lakeshore.model_224*), 183
Model224RemoteInterface (*class in lakeshore.model_224*), 181
Model224ServiceRequestRegister (*class in lakeshore.model_224*), 184
Model224SoftCalSensorTypes (*class in lakeshore.model_224*), 183
Model224StandardEventRegister (*in module lakeshore.model_224*), 183
Model224StatusByteRegister (*class in lakeshore.model_224*), 184
Model240 (*class in lakeshore.model_240*), 186
Model240BrightnessLevel (*class in lakeshore.model_240*), 191
Model240Coefficients (*class in lakeshore.model_240*), 191
Model240CurveFormat (*class in lakeshore.model_240*), 191
Model240CurveHeader (*class in lakeshore.model_240*), 190
Model240InputParameter (*class in lakeshore.model_240*), 190
Model240InputRange (*class in lakeshore.model_240*), 192
Model240ProfiSlot (*class in lakeshore.model_240*), 191
Model240SensorTypes (*class in lakeshore.model_240*), 191
Model240TemperatureCoefficient (*class in lakeshore.model_240*), 192
Model240Units (*class in lakeshore.model_240*), 191
Model335 (*class in lakeshore.model_335*), 63
Model335AutoTuneMode (*in module lakeshore.model_335*), 88
Model335BrightnessLevel (*in module lakeshore.model_335*), 90
Model335ControlLoopZoneSettings (*class in lakeshore.model_335*), 86
Model335ControlTypes (*in module lakeshore.model_335*), 91
Model335CurveFormat (*in module lakeshore.model_335*), 88
Model335CurveTemperatureCoefficient (*in module lakeshore.model_335*), 88
Model335DiodeCurrent (*in module lakeshore.model_335*), 91
Model335DiodeRange (*class in lakeshore.model_335*), 89
Model335DisplayFieldUnits (*class in lakeshore.model_335*), 91
Model335DisplayInputChannel (*class in lakeshore.model_335*), 91
Model335DisplaySetup (*class in lakeshore.model_335*), 91
Model335HeaterError (*in module lakeshore.model_335*), 87
Model335HeaterOutputDisplay (*class in*

lakeshore.model_335), 90
Model335HeaterOutputMode (*class in lakeshore.model_335*), 90
Model335HeaterOutputUnits (*in module lakeshore.model_335*), 88
Model335HeaterOutType (*class in lakeshore.model_335*), 90
Model335HeaterRange (*class in lakeshore.model_335*), 90
Model335HeaterResistance (*in module lakeshore.model_335*), 88
Model335HeaterVoltageRange (*class in lakeshore.model_335*), 91
Model335InputReadingStatus (*class in lakeshore.model_335*), 92
Model335InputSensor (*class in lakeshore.model_335*), 88
Model335InputSensorSettings (*class in lakeshore.model_335*), 85
Model335InputSensorType (*class in lakeshore.model_335*), 89
Model335InputSensorUnits (*in module lakeshore.model_335*), 89
Model335InterfaceMode (*in module lakeshore.model_335*), 87
Model335MonitorOutUnits (*class in lakeshore.model_335*), 89
Model335OperationEvent (*in module lakeshore.model_335*), 92
Model335Polarity (*in module lakeshore.model_335*), 89
Model335RelayControlAlarm (*in module lakeshore.model_335*), 87
Model335RelayControlMode (*in module lakeshore.model_335*), 87
Model335RTDRange (*class in lakeshore.model_335*), 89
Model335ServiceRequestEnable (*class in lakeshore.model_335*), 92
Model335StandardEventRegister (*in module lakeshore.model_335*), 92
Model335StatusByteRegister (*class in lakeshore.model_335*), 92
Model335ThermocoupleRange (*class in lakeshore.model_335*), 90
Model335WarmupControl (*class in lakeshore.model_335*), 90
Model336 (*class in lakeshore.model_336*), 95
Model336AlarmSettings (*in module lakeshore.model_336*), 121
Model336AutoTuneMode (*in module lakeshore.model_336*), 124
Model336ControlLoopZoneSettings (*class in lakeshore.model_336*), 120
Model336ControlTypes (*in module lakeshore.model_336*), 124
Model336CurveFormat (*in module lakeshore.model_336*), 124
Model336CurveHeader (*in module lakeshore.model_336*), 121
Model336CurveTemperatureCoefficients (*in module lakeshore.model_336*), 124
Model336DiodeCurrent (*in module lakeshore.model_336*), 124
Model336DiodeRange (*class in lakeshore.model_336*), 123
Model336DisplayFields (*in module lakeshore.model_336*), 125
Model336DisplayFieldsSize (*in module lakeshore.model_336*), 125
Model336DisplaySetupMode (*class in lakeshore.model_336*), 122
Model336DisplayUnits (*class in lakeshore.model_336*), 124
Model336HeaterError (*in module lakeshore.model_336*), 124
Model336HeaterOutputMode (*class in lakeshore.model_336*), 123
Model336HeaterOutputUnits (*in module lakeshore.model_336*), 124
Model336HeaterRange (*class in lakeshore.model_336*), 123
Model336HeaterResistance (*in module lakeshore.model_336*), 124
Model336HeaterVoltageRange (*class in lakeshore.model_336*), 124
Model336InputChannel (*class in lakeshore.model_336*), 122
Model336InputReadingStatus (*class in lakeshore.model_336*), 126
Model336InputSensorSettings (*class in lakeshore.model_336*), 119
Model336InputSensorType (*class in lakeshore.model_336*), 122
Model336InputSensorUnits (*in module lakeshore.model_336*), 124
Model336Interface (*in module lakeshore.model_336*), 125
Model336InterfaceMode (*in module lakeshore.model_336*), 124
Model336LanStatus (*in module lakeshore.model_336*), 124
Model336OperationEvent (*in module lakeshore.model_336*), 126
Model336Polarity (*in module lakeshore.model_336*), 124
Model336RelayControlAlarm (*in module lakeshore.model_336*), 124

Model336RelayControlMode (*in module lakeshore.model_336*), 124
Model336RTDRange (*class in lakeshore.model_336*), 123
Model336ServiceRequestEnable (*class in lakeshore.model_336*), 126
Model336StandardEventRegister (*in module lakeshore.model_336*), 125
Model336StatusByteRegister (*class in lakeshore.model_336*), 125
Model336ThermocoupleRange (*class in lakeshore.model_336*), 123
Model350 (*class in lakeshore.model_350*), 126
Model372 (*class in lakeshore.model_372*), 129
Model372AlarmParameters (*class in lakeshore.model_372*), 157
Model372AutoRangeMode (*class in lakeshore.model_372*), 160
Model372BrightnessLevel (*in module lakeshore.model_372*), 164
Model372ControlInputCurrentRange (*class in lakeshore.model_372*), 163
Model372ControlLoopZoneSettings (*class in lakeshore.model_372*), 156
Model372CurveFormat (*class in lakeshore.model_372*), 161
Model372CurveHeader (*in module lakeshore.model_372*), 157
Model372CurveTemperatureCoefficient (*in module lakeshore.model_372*), 163
Model372DisplayFields (*in module lakeshore.model_372*), 163
Model372DisplayFieldUnits (*class in lakeshore.model_372*), 161
Model372DisplayInfo (*class in lakeshore.model_372*), 160
Model372DisplayMode (*class in lakeshore.model_372*), 160
Model372HeaterError (*in module lakeshore.model_372*), 164
Model372HeaterOutputSettings (*class in lakeshore.model_372*), 155
Model372HeaterOutputUnits (*in module lakeshore.model_372*), 164
Model372HeaterResistance (*in module lakeshore.model_372*), 164
Model372InputChannel (*class in lakeshore.model_372*), 159
Model372InputChannelSettings (*class in lakeshore.model_372*), 154
Model372InputFrequency (*class in lakeshore.model_372*), 161
Model372InputSensorUnits (*class in lakeshore.model_372*), 160
Model372InputSetupSettings (*class in lakeshore.model_372*), 155
Model372Interface (*in module lakeshore.model_372*), 164
Model372InterfaceMode (*in module lakeshore.model_372*), 163
Model372MeasurementInputCurrentRange (*class in lakeshore.model_372*), 162
Model372MeasurementInputResistance (*class in lakeshore.model_372*), 163
Model372MeasurementInputVoltageRange (*class in lakeshore.model_372*), 161
Model372MonitorOutputSource (*class in lakeshore.model_372*), 160
Model372OutputMode (*class in lakeshore.model_372*), 159
Model372Polarity (*in module lakeshore.model_372*), 164
Model372ReadingStatusRegister (*class in lakeshore.model_372*), 158
Model372RelayControlMode (*class in lakeshore.model_372*), 160
Model372SampleHeaterOutputRange (*class in lakeshore.model_372*), 161
Model372SensorExcitationMode (*class in lakeshore.model_372*), 160
Model372ServiceRequestEnable (*class in lakeshore.model_372*), 158
Model372StandardEventRegister (*in module lakeshore.model_372*), 158
Model372StatusByteRegister (*class in lakeshore.model_372*), 158
Model425 (*class in lakeshore.model_425*), 19
Model643 (*class in lakeshore.model_643*), 20
Model648 (*class in lakeshore.model_648*), 21
modify_operation_register_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
modify_operation_register_mask() (*lakeshore.model_155.PrecisionSource method*), 197
modify_operation_register_mask() (*lakeshore.ssm_system.SSMSystem method*), 45
modify_operation_register_mask() (*lakeshore.teslameter.Teslameter method*), 17
modify_questionable_register_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
modify_questionable_register_mask() (*lakeshore.model_155.PrecisionSource method*), 198
modify_questionable_register_mask() (*lakeshore.ssm_system.SSMSystem method*), 45

modify_questionable_register_mask() (*lakeshore.teslameter.Teslameter method*), 17
modify_service_request_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
modify_service_request_mask() (*lakeshore.model_155.PrecisionSource method*), 198
modify_service_request_mask() (*lakeshore.ssm_system.SSMSystem method*), 46
modify_service_request_mask() (*lakeshore.teslameter.Teslameter method*), 17
modify_standard_event_register_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
modify_standard_event_register_mask() (*lakeshore.model_155.PrecisionSource method*), 198
modify_standard_event_register_mask() (*lakeshore.ssm_system.SSMSystem method*), 46
modify_standard_event_register_mask() (*lakeshore.teslameter.Teslameter method*), 17
MODULE_ERROR (*lakeshore.temperature_controllers.LanStatus attribute*), 125
MONITOR_OUT (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
MONITOR_OUT (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123
MONITOR_OUT (*lakeshore.model_372.Model372OutputMode attribute*), 159

N

NEGATIVE (*lakeshore.model_224.Model224CurveTemperatureCoefficient attribute*), 183
NEGATIVE (*lakeshore.model_240.Model240Coefficients attribute*), 191
NEGATIVE (*lakeshore.model_240.Model240TemperatureCoefficient attribute*), 192
NEGATIVE (*lakeshore.temperature_controllers.CurveTemperatureCoefficient attribute*), 88
NINE (*lakeshore.model_372.Model372InputChannel attribute*), 159
NO_ERROR (*lakeshore.temperature_controllers.HeaterError attribute*), 88
NO_INPUT (*lakeshore.model_224.Model224InputChannel attribute*), 182
NONE (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
NONE (*lakeshore.model_335.Model335InputSensor attribute*), 89
NONE (*lakeshore.model_336.Model336InputChannel attribute*), 122
NONE (*lakeshore.model_372.Model372DisplayInfo attribute*), 161
NONE (*lakeshore.model_372.Model372InputChannel attribute*), 159
NTC_RTD (*lakeshore.model_224.Model224InputSensorType attribute*), 180
NTC_RTD (*lakeshore.model_240.Model240SensorTypes attribute*), 191
NTC_RTD (*lakeshore.model_335.Model335InputSensorType attribute*), 89
NTC_RTD (*lakeshore.model_336.Model336InputSensorType attribute*), 123

O

OFF (*lakeshore.model_240.Model240BrightnessLevel attribute*), 192
OFF (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
OFF (*lakeshore.model_335.Model335HeaterRange attribute*), 91
OFF (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123
OFF (*lakeshore.model_336.Model336HeaterRange attribute*), 124
OFF (*lakeshore.model_372.Model372AutoRangeMode attribute*), 160
OFF (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
OFF (*lakeshore.model_372.Model372OutputMode attribute*), 159
OFF (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
OHM_PER_KELVIN (*lakeshore.model_372.Model372CurveFormat attribute*), 161
OHM_PER_KELVIN_CUBIC_SPLINE (*lakeshore.model_372.Model372CurveFormat attribute*), 161
OHMS (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
OHMS (*lakeshore.model_372.Model372InputSensorUnits attribute*), 160
OHMS_PER_KELVIN (*lakeshore.model_224.Model224CurveFormat attribute*), 183
OHMS_PER_KELVIN (*lakeshore.model_240.Model240CurveFormat attribute*), 191
OHMS_PER_KELVIN (*lakeshore.temperature_controllers.CurveFormat attribute*), 88
ONE (*lakeshore.model_372.Model372InputChannel attribute*), 159
ONE_HUNDRED_KILOHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181
ONE_HUNDRED_OHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181

ONE_HUNDRED_OHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResistanceRange attribute*), 181
ONE_HUNDRED_THOUSAND_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
ONE_HUNDRED_THOUSAND_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
ONE_KILOHM (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181
ONE_KILOHM (*lakeshore.model_224.Model224PlatinumRTDSensorResistanceRange attribute*), 181
ONE_MILLI_AMP (*lakeshore.model_224.Model224DiodeExcitationCurrent attribute*), 181
ONE_MILLIAMP (*lakeshore.temperature_controllers.DiodeCurrent attribute*), 91
ONE_THOUSAND_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
ONE_THOUSAND_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
OPEN_LOOP (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
OPEN_LOOP (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123
OPEN_LOOP (*lakeshore.model_372.Model372OutputMode attribute*), 159
OperationEvent (*class in lakeshore.temperature_controllers*), 92
OUTPUT_1 (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
OUTPUT_2 (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
output_dc_current() (*lakeshore.model_155.PrecisionSource method*), 195
output_dc_voltage() (*lakeshore.model_155.PrecisionSource method*), 195
output_sine_current() (*lakeshore.model_155.PrecisionSource method*), 195
output_sine_voltage() (*lakeshore.model_155.PrecisionSource method*), 195

P

P_I (*lakeshore.temperature_controllers.AutotuneMode attribute*), 88
P_I_D (*lakeshore.temperature_controllers.AutotuneMode attribute*), 88
P_ONLY (*lakeshore.temperature_controllers.AutotuneMode attribute*), 88
PLATINUM_RTD (*lakeshore.model_224.Model224InputSensorType attribute*), 180
PLATINUM_RTD (*lakeshore.model_240.Model240SensorTypes attribute*), 191
PLATINUM_RTD (*lakeshore.model_335.Model335InputSensorType attribute*), 89
PLATINUM_RTD (*lakeshore.model_336.Model336InputSensorType attribute*), 123
Polarity (*class in lakeshore.temperature_controllers*), 89
POSITIVE (*lakeshore.model_224.Model224CurveTemperatureCoefficients attribute*), 183
POSITIVE (*lakeshore.model_240.Model240Coefficients attribute*), 191
POSITIVE (*lakeshore.model_240.Model240TemperatureCoefficient attribute*), 192
POSITIVE (*lakeshore.temperature_controllers.CurveTemperatureCoefficie attribute*), 88
PowerRange (*lakeshore.model_335.Model335HeaterOutputDisplay attribute*), 90
PowerRange (*lakeshore.temperature_controllers.HeaterOutputUnits attribute*), 88
PrecisionSource (*class in lakeshore.model_155*), 194
PT100 (*lakeshore.model_224.Model224SoftCalSensorTypes attribute*), 183
PT_1000 (*lakeshore.model_224.Model224SoftCalSensorTypes attribute*), 183

Q

QUADRATURE (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
QUARTER (*lakeshore.temperature_controllers.BrightnessLevel attribute*), 90
query() (*lakeshore.fast_hall_controller.FastHall method*), 28
query() (*lakeshore.model_121.Model121 method*), 193
query() (*lakeshore.model_155.PrecisionSource method*), 198
query() (*lakeshore.model_224.Model224 method*), 165
query() (*lakeshore.model_240.Model240 method*), 189
query() (*lakeshore.model_335.Model335 method*), 80
query() (*lakeshore.model_336.Model336 method*), 114
query() (*lakeshore.model_350.Model350 method*), 127
query() (*lakeshore.model_372.Model372 method*), 150
query() (*lakeshore.model_425.Model425 method*), 19
query() (*lakeshore.model_643.Model643 method*), 20
query() (*lakeshore.model_648.Model648 method*), 21
query() (*lakeshore.ssm_system.SSMSystem method*), 46

query() (*lakeshore.teslameter.Teslameter method*), 17

R

RANGE_100_MICRO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_100_MICRO_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_100_MILLI_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_100_NANO_AMPS (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_100_NANO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_100_PICO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_10_MICRO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_10_MILLI_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_10_MILLI_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_10_NANO_AMPS (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_10_NANO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_10_PICO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_10_VOLTS (*lakeshore.model_224.Model224DiodeSensorRange attribute*), 181
RANGE_1_MICRO_AMP (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_1_MILLI_AMP (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_1_MILLI_AMP (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_1_NANO_AMP (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_1_NANO_AMP (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_1_PICO_AMP (*lakeshore.model_372.Model372MeasurementInput attribute*), 162
RANGE_200_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_200_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_200_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_200_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_200_OHMS (*lakeshore.model_372.Model372MeasurementInputRe attribute*), 163
RANGE_20_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_20_MEGA_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_20_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_20_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_20_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_20_OHMS (*lakeshore.model_372.Model372MeasurementInputRes attribute*), 163
RANGE_2_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInpu attribute*), 163
RANGE_2_MEGA_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_2_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_2_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_2_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_2_OHMS (*lakeshore.model_372.Model372MeasurementInputResis attribute*), 163
RANGE_2_POINT_5_VOLTS (*lakeshore.model_224.Model224DiodeSensorRange attribute*), 181
RANGE_316_MICRO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange*

attribute), 162
RANGE_316_MICRO_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_316_NANO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_316_PICO_AMPS (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_316_PICO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_31_POINT_6_MICRO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_31_POINT_6_MICRO_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_31_POINT_6_MILLI_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_31_POINT_6_MILLI_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_31_POINT_6_NANO_AMPS (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_31_POINT_6_NANO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_31_POINT_6_PICO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_3_POINT_16_MICRO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_3_POINT_16_MILLI_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_3_POINT_16_MILLI_AMPS (*lakeshore.model_372.Model372SampleHeaterOutputRange attribute*), 161
RANGE_3_POINT_16_NANO_AMPS (*lakeshore.model_372.Model372ControlInputCurrentRange attribute*), 163
RANGE_3_POINT_16_NANO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_3_POINT_16_PICO_AMPS (*lakeshore.model_372.Model372MeasurementInputCurrentRange attribute*), 162
RANGE_632_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInputResistanceRange attribute*), 163
RANGE_632_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_632_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_632_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_632_OHMS (*lakeshore.model_372.Model372MeasurementInputResistanceRange attribute*), 163
RANGE_63_POINT_2_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_63_POINT_2_MEGA_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_63_POINT_2_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_63_POINT_2_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_63_POINT_2_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_63_POINT_2_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_6_POINT_32_KIL_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_6_POINT_32_MEGA_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_6_POINT_32_MICRO_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_6_POINT_32_MILLI_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_6_POINT_32_MILLI_VOLTS (*lakeshore.model_372.Model372MeasurementInputVoltageRange attribute*), 162
RANGE_6_POINT_32_OHMS (*lakeshore.model_372.Model372MeasurementInputResistance attribute*), 163
RANGE_DIODE (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_PT100_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_PT1000_OHMS

(*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_10_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_10_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_1_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_30_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_30_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_NTCRTD_3_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RANGE_PTRTD_1_KIL_OHMS (*lakeshore.model_240.Model240InputRange attribute*), 192
RELAY_OFF (*lakeshore.model_224.Model224RelayControlMode attribute*), 183
RELAY_OFF (*lakeshore.model_372.Model372RelayControlMode attribute*), 160
RELAY_OFF (*lakeshore.temperature_controllers.RelayControlMode attribute*), 87
RELAY_ON (*lakeshore.model_224.Model224RelayControlMode attribute*), 183
RELAY_ON (*lakeshore.model_372.Model372RelayControlMode attribute*), 160
RELAY_ON (*lakeshore.temperature_controllers.RelayControlMode attribute*), 87
RelayControlAlarm (*class in lakeshore.temperature_controllers*), 87
RelayControlMode (*class in lakeshore.temperature_controllers*), 87
REMOTE (*lakeshore.model_224.Model224InterfaceMode attribute*), 181
REMOTE (*lakeshore.temperature_controllers.InterfaceMode attribute*), 87
REMOTE_LOCAL_LOCK (*lakeshore.model_224.Model224InterfaceMode attribute*), 181
REMOTE_LOCAL_LOCK (*lakeshore.temperature_controllers.InterfaceMode attribute*), 87
rename() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
reset_alarm_status() (*lakeshore.model_224.Model224 method*), 171
reset_alarm_status() (*lakeshore.model_335.Model335 method*), 80
reset_alarm_status() (*lakeshore.model_336.Model336 method*), 114
reset_alarm_status() (*lakeshore.model_372.Model372 method*), 150
reset_contact_check_measurement() (*lakeshore.fast_hall_controller.FastHall method*), 26
reset_dc_hall_measurement() (*lakeshore.fast_hall_controller.FastHall method*), 26
reset_fasthall_measurement() (*lakeshore.fast_hall_controller.FastHall method*), 26
reset_four_wire_measurement() (*lakeshore.fast_hall_controller.FastHall method*), 26
reset_head_self_calibration() (*lakeshore.ssm_system.SSMSystem method*), 44
reset_instrument() (*lakeshore.model_224.Model224 method*), 166
reset_instrument() (*lakeshore.model_335.Model335 method*), 81
reset_instrument() (*lakeshore.model_336.Model336 method*), 114
reset_instrument() (*lakeshore.model_372.Model372 method*), 129
reset_max_min() (*lakeshore.teslameter.Teslameter method*), 11
reset_measurement_settings() (*lakeshore.fast_hall_controller.FastHall method*), 28
reset_measurement_settings() (*lakeshore.model_155.PrecisionSource method*), 198
reset_measurement_settings() (*lakeshore.ssm_system.SSMSystem method*), 46
reset_measurement_settings() (*lakeshore.teslameter.Teslameter method*), 17
reset_min_max_data() (*lakeshore.model_224.Model224 method*), 170
reset_min_max_data() (*lakeshore.model_335.Model335 method*), 81

reset_min_max_data() (*lakeshore.model_336.Model336 method*), 114
reset_min_max_data() (*lakeshore.model_372.Model372 method*), 150
reset_qualifier_latch() (*lakeshore.teslameter.Teslameter method*), 15
reset_resistivity_measurement() (*lakeshore.fast_hall_controller.FastHall method*), 26
reset_self_cal() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
reset_self_cal() (*lakeshore.ssm_source_module.SourceModule method*), 47
reset_status_register_masks() (*lakeshore.fast_hall_controller.FastHall method*), 28
reset_status_register_masks() (*lakeshore.model_155.PrecisionSource method*), 198
reset_status_register_masks() (*lakeshore.ssm_system.SSMSystem method*), 46
reset_status_register_masks() (*lakeshore.teslameter.Teslameter method*), 17
ResistivityLinkParameters (*class in lakeshore.fast_hall_controller*), 39
ResistivityManualParameters (*class in lakeshore.fast_hall_controller*), 37
restore() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
route_terminals() (*lakeshore.model_155.PrecisionSource method*), 195
ROX102B (*lakeshore.model_372.Model372AutoRangeMode attribute*), 160
run_complete_contact_check_manual() (*lakeshore.fast_hall_controller.FastHall method*), 25
run_complete_contact_check_optimized() (*lakeshore.fast_hall_controller.FastHall method*), 25
run_complete_dc_hall() (*lakeshore.fast_hall_controller.FastHall method*), 26
run_complete_fasthall_link() (*lakeshore.fast_hall_controller.FastHall method*), 25
run_complete_fasthall_manual() (*lakeshore.fast_hall_controller.FastHall method*), 26
run_complete_four_wire() (*lakeshore.fast_hall_controller.FastHall method*), 26
run_complete_resistivity_link() (*lakeshore.fast_hall_controller.FastHall method*), 26
run_complete_resistivity_manual() (*lakeshore.fast_hall_controller.FastHall method*), 26
run_head_self_calibration() (*lakeshore.ssm_system.SSMSystem method*), 44
run_self_cal() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
run_self_cal() (*lakeshore.ssm_source_module.SourceModule method*), 47

S

SAMPLE_HEATER (*lakeshore.model_372.Model372DisplayInfo attribute*), 161
SAMPLE_HEATER_ZONE (*lakeshore.model_372.Model372RelayControlMode attribute*), 160
select_interface_mode() (*lakeshore.model_224.Model224 method*), 176
select_remote_interface() (*lakeshore.model_224.Model224 method*), 176
SENSOR (*lakeshore.model_224.Model224DisplayFieldUnits attribute*), 182
SENSOR (*lakeshore.model_224.Model224InputSensorUnits attribute*), 180
SENSOR (*lakeshore.model_240.Model240Units attribute*), 191
SENSOR (*lakeshore.model_335.Model335MonitorOutUnits attribute*), 89
SENSOR (*lakeshore.temperature_controllers.InputSensorUnits attribute*), 89
SENSOR_NAME (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 92
SENSOR_NAME (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
SENSOR_NAME (*lakeshore.model_372.Model372DisplayFieldUnits attribute*), 161
SENSOR_UNITS (*lakeshore.model_335.Model335DisplayFieldUnits attribute*), 92
SENSOR_UNITS (*lakeshore.model_336.Model336DisplayUnits attribute*), 124
service_request_enable (*lakeshore.model_336.Model336 attribute*), 95
set_alarm_beep() (*lakeshore.model_372.Model372 method*), 136
set_alarm_parameters() (*lakeshore.model_224.Model224 method*), 171

set_alarm_parameters() (*lakeshore.model_335.Model335 method*), 81
set_alarm_parameters() (*lakeshore.model_336.Model336 method*), 114
set_alarm_parameters() (*lakeshore.model_372.Model372 method*), 139
set_analog_output() (*lakeshore.teslameter.Teslameter method*), 14
set_analog_output_signal() (*lakeshore.teslameter.Teslameter method*), 14
set_autotune() (*lakeshore.model_335.Model335 method*), 64
set_autotune() (*lakeshore.model_336.Model336 method*), 95
set_averaging_time() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
set_band_pass_filter_center() (*lakeshore.teslameter.Teslameter method*), 15
set_bias_voltage() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
set_brightness() (*lakeshore.model_240.Model240 method*), 186
set_brightness() (*lakeshore.model_335.Model335 method*), 64
set_cmr_source() (*lakeshore.ssm_source_module.SourceModule method*), 49
set_cmr_state() (*lakeshore.ssm_source_module.SourceModule method*), 49
set_common_mode_reduction() (*lakeshore.model_372.Model372 method*), 135
set_contrast_level() (*lakeshore.model_336.Model336 method*), 95
set_control_loop_parameters() (*lakeshore.model_372.Model372 method*), 143
set_control_loop_zone_table() (*lakeshore.model_335.Model335 method*), 73
set_control_loop_zone_table() (*lakeshore.model_336.Model336 method*), 107
set_control_setpoint() (*lakeshore.model_335.Model335 method*), 81
set_control_setpoint() (*lakeshore.model_336.Model336 method*), 115
set_control_setpoint() (*lakeshore.model_372.Model372 method*), 150
set_coupling() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
set_coupling() (*lakeshore.ssm_source_module.SourceModule method*), 49
set_current_limit() (*lakeshore.model_155.PrecisionSource method*), 196
set_current_mode_voltage_protection() (*lakeshore.model_155.PrecisionSource method*), 196
set_current_range() (*lakeshore.model_155.PrecisionSource method*), 195
set_curve() (*lakeshore.model_224.Model224 method*), 173
set_curve() (*lakeshore.model_335.Model335 method*), 81
set_curve() (*lakeshore.model_336.Model336 method*), 115
set_curve() (*lakeshore.model_372.Model372 method*), 150
set_curve_data_point() (*lakeshore.model_224.Model224 method*), 172
set_curve_data_point() (*lakeshore.model_240.Model240 method*), 187
set_curve_data_point() (*lakeshore.model_335.Model335 method*), 81
set_curve_data_point() (*lakeshore.model_336.Model336 method*), 115
set_curve_data_point() (*lakeshore.model_372.Model372 method*), 150
set_curve_header() (*lakeshore.model_224.Model224 method*), 171
set_curve_header() (*lakeshore.model_240.Model240 method*), 186
set_curve_header() (*lakeshore.model_335.Model335 method*), 82
set_curve_header() (*lakeshore.model_336.Model336 method*), 115

set_curve_header() (*lakeshore.model_372.Model372 method*), 151
set_dark_mode_state() (*lakeshore.ssm_measure_module.MeasureModule method*), 59
set_dark_mode_state() (*lakeshore.ssm_source_module.SourceModule method*), 53
set_description() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
set_digital_output() (*lakeshore.model_372.Model372 method*), 138
set_diode_excitation_current() (*lakeshore.model_335.Model335 method*), 66
set_diode_excitation_current() (*lakeshore.model_336.Model336 method*), 101
set_display_contrast() (*lakeshore.model_224.Model224 method*), 168
set_display_field_settings() (*lakeshore.model_224.Model224 method*), 176
set_display_field_settings() (*lakeshore.model_335.Model335 method*), 82
set_display_field_settings() (*lakeshore.model_336.Model336 method*), 116
set_display_field_settings() (*lakeshore.model_372.Model372 method*), 151
set_display_settings() (*lakeshore.model_372.Model372 method*), 129
set_display_setup() (*lakeshore.model_335.Model335 method*), 69
set_display_setup() (*lakeshore.model_336.Model336 method*), 102
set_duty() (*lakeshore.ssm_source_module.SourceModule method*), 48
set_enable_state() (*lakeshore.ssm_source_module.SourceModule method*), 47
set_excitation_frequency() (*lakeshore.model_372.Model372 method*), 138
set_excitation_mode() (*lakeshore.ssm_source_module.SourceModule method*), 48
set_factory_defaults() (*lakeshore.model_240.Model240 method*), 186
set_field_control_open_loop_voltage() (*lakeshore.teslameter.Teslameter method*), 14
set_field_control_setpoint() (*lakeshore.teslameter.Teslameter method*), 13
set_filter() (*lakeshore.model_224.Model224 method*), 174
set_filter() (*lakeshore.model_240.Model240 method*), 187
set_filter() (*lakeshore.model_335.Model335 method*), 67
set_filter() (*lakeshore.model_336.Model336 method*), 97
set_filter() (*lakeshore.model_372.Model372 method*), 133
set_frequency() (*lakeshore.ssm_source_module.SourceModule method*), 48
set_frequency_filter_type() (*lakeshore.teslameter.Teslameter method*), 14
set_gain_allocation_strategy() (*lakeshore.ssm_measure_module.MeasureModule method*), 55
set_guard_state() (*lakeshore.ssm_source_module.SourceModule method*), 49
set_heater_output_mode() (*lakeshore.model_335.Model335 method*), 71
set_heater_output_mode() (*lakeshore.model_336.Model336 method*), 104
set_heater_output_range() (*lakeshore.model_372.Model372 method*), 132
set_heater_pid() (*lakeshore.model_335.Model335 method*), 82
set_heater_pid() (*lakeshore.model_336.Model336 method*), 116
set_heater_pid() (*lakeshore.model_372.Model372 method*), 151
set_heater_range() (*lakeshore.model_335.Model335 method*), 72
set_heater_range() (*lakeshore.model_336.Model336 method*), 105
set_heater_setup() (*lakeshore.model_336.Model336 method*),

103
set_heater_setup_one() (lakeshore.model_335.Model335 method), 69
set_heater_setup_two() (lakeshore.model_335.Model335 method), 69
set_high_pass_filter_cutoff() (lakeshore.teslameter.Teslameter method), 15
set_i_amplitude() (lakeshore.ssm_source_module.SourceModule method), 50
set_i_limit() (lakeshore.ssm_source_module.SourceModule method), 51
set_i_offset() (lakeshore.ssm_source_module.SourceModule method), 50
set_identify_state() (lakeshore.ssm_measure_module.MeasureModule method), 59
set_identify_state() (lakeshore.ssm_source_module.SourceModule method), 53
set_ieee_488() (lakeshore.model_224.Model224 method), 169
set_ieee_488() (lakeshore.model_335.Model335 method), 83
set_ieee_488() (lakeshore.model_336.Model336 method), 116
set_ieee_488() (lakeshore.model_372.Model372 method), 152
set_ieee_interface_mode() (lakeshore.model_372.Model372 method), 140
set_ieee_interface_parameter() (lakeshore.model_372.Model372 method), 134
set_input_channel_parameters() (lakeshore.model_372.Model372 method), 132
set_input_configuration() (lakeshore.ssm_measure_module.MeasureModule method), 54
set_input_curve() (lakeshore.model_224.Model224 method), 170
set_input_curve() (lakeshore.model_335.Model335 method), 83
set_input_curve() (lakeshore.model_336.Model336 method), 116
set_input_curve() (lakeshore.model_372.Model372 method), 152
set_input_diode_excitation_current() (lakeshore.model_224.Model224 method), 167
set_input_parameter() (lakeshore.model_240.Model240 method), 188
set_input_sensor() (lakeshore.model_335.Model335 method), 70
set_input_sensor() (lakeshore.model_336.Model336 method), 104
set_interface() (lakeshore.model_336.Model336 method), 100
set_interface() (lakeshore.model_372.Model372 method), 139
set_keypad_lock() (lakeshore.model_224.Model224 method), 169
set_keypad_lock() (lakeshore.model_335.Model335 method), 83
set_keypad_lock() (lakeshore.model_336.Model336 method), 117
set_keypad_lock() (lakeshore.model_372.Model372 method), 152
set_led_state() (lakeshore.model_224.Model224 method), 169
set_led_state() (lakeshore.model_335.Model335 method), 83
set_led_state() (lakeshore.model_336.Model336 method), 117
set_led_state() (lakeshore.model_372.Model372 method), 152
set_lock_in_fir_state() (lakeshore.ssm_measure_module.MeasureModule method), 57
set_lock_in_rolloff() (lakeshore.ssm_measure_module.MeasureModule method), 56
set_lock_in_time_constant() (lakeshore.ssm_measure_module.MeasureModule method), 56
set_low_pass_filter_cutoff() (lakeshore.teslameter.Teslameter method), 15
set_manual_output() (lakeshore.model_335.Model335 method), 83
set_manual_output() (lakeshore.model_336.Model336 method), 117

set_manual_output() (*lakeshore.model_372.Model372 method*), 152
set_mode() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
set_modname() (*lakeshore.model_240.Model240 method*), 188
set_mon_out_manual_level() (*lakeshore.ssm_system.SSMSystem method*), 44
set_mon_out_mode() (*lakeshore.ssm_system.SSMSystem method*), 43
set_mon_out_state() (*lakeshore.ssm_system.SSMSystem method*), 43
set_monitor_output_heater() (*lakeshore.model_335.Model335 method*), 68
set_monitor_output_heater() (*lakeshore.model_336.Model336 method*), 101
set_monitor_output_source() (*lakeshore.model_372.Model372 method*), 140
set_name() (*lakeshore.ssm_measure_module.MeasureModule method*), 53
set_name() (*lakeshore.ssm_source_module.SourceModule method*), 47
set_network_settings() (*lakeshore.model_336.Model336 method*), 98
set_operation_event_enable() (*lakeshore.model_335.Model335 method*), 64
set_operation_event_enable() (*lakeshore.model_336.Model336 method*), 95
set_operation_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
set_operation_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 198
set_operation_event_enable_mask() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
set_operation_event_enable_mask() (*lakeshore.ssm_source_module.SourceModule method*), 53
set_operation_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 46
set_operation_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 18
set_output() (*lakeshore.model_155.PrecisionSource method*), 194
set_output_two_polarity() (*lakeshore.model_335.Model335 method*), 71
set_profibus_address() (*lakeshore.model_240.Model240 method*), 188
set_profibus_slot_configuration() (*lakeshore.model_240.Model240 method*), 188
set_profibus_slot_count() (*lakeshore.model_240.Model240 method*), 188
set_qualifier_latching_setting() (*lakeshore.teslameter.Teslameter method*), 15
set_questionable_event_enable_mask() (*lakeshore.fast_hall_controller.FastHall method*), 28
set_questionable_event_enable_mask() (*lakeshore.model_155.PrecisionSource method*), 198
set_questionable_event_enable_mask() (*lakeshore.ssm_measure_module.MeasureModule method*), 58
set_questionable_event_enable_mask() (*lakeshore.ssm_source_module.SourceModule method*), 52
set_questionable_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 46
set_questionable_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 18
set_ref_in_edge() (*lakeshore.ssm_system.SSMSystem method*), 43
set_ref_out_source() (*lakeshore.ssm_system.SSMSystem method*), 43
set_ref_out_state() (*lakeshore.ssm_system.SSMSystem method*), 43
set_reference_harmonic() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
set_reference_phase_shift() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
set_reference_source() (*lakeshore.ssm_measure_module.MeasureModule method*), 56
set_relative_field_baseline() (*lakeshore.teslameter.Teslameter method*), 12
set_relay_alarms() (*lakeshore.model_224.Model224 method*), 177
set_relay_alarms() (*lakeshore.model_335.Model335 method*),

83
set_relay_alarms() (lakeshore.model_336.Model336 method), 117
set_relay_alarms() (lakeshore.model_372.Model372 method), 152
set_relay_for_sample_heater_control_zones() (lakeshore.model_372.Model372 method), 139
set_relay_for_warmup_heater_control_zone() (lakeshore.model_372.Model372 method), 140
set_remote_interface_mode() (lakeshore.model_335.Model335 method), 84
set_remote_interface_mode() (lakeshore.model_336.Model336 method), 118
set_remote_interface_mode() (lakeshore.model_372.Model372 method), 153
set_scanner_status() (lakeshore.model_372.Model372 method), 135
set_sensor_name() (lakeshore.model_224.Model224 method), 168
set_sensor_name() (lakeshore.model_240.Model240 method), 187
set_sensor_name() (lakeshore.model_335.Model335 method), 84
set_sensor_name() (lakeshore.model_336.Model336 method), 118
set_sensor_name() (lakeshore.model_372.Model372 method), 153
set_service_request() (lakeshore.model_224.Model224 method), 166
set_service_request() (lakeshore.model_335.Model335 method), 84
set_service_request() (lakeshore.model_336.Model336 method), 118
set_service_request() (lakeshore.model_372.Model372 method), 153
set_service_request_enable_mask() (lakeshore.fast_hall_controller.FastHall method), 29
set_service_request_enable_mask() (lakeshore.model_155.PrecisionSource method), 198
set_service_request_enable_mask() (lakeshore.ssm_system.SSMSystem method), 46
set_service_request_enable_mask() (lakeshore.teslameter.Teslameter method), 18
set_setpoint_kelvin() (lakeshore.model_372.Model372 method), 137
set_setpoint_ohms() (lakeshore.model_372.Model372 method), 137
set_setpoint_ramp_parameter() (lakeshore.model_335.Model335 method), 84
set_setpoint_ramp_parameter() (lakeshore.model_336.Model336 method), 118
set_setpoint_ramp_parameter() (lakeshore.model_372.Model372 method), 153
set_shape() (lakeshore.ssm_source_module.SourceModule method), 48
set_soft_cal_curve_dt_470() (lakeshore.model_335.Model335 method), 64
set_soft_cal_curve_dt_470() (lakeshore.model_336.Model336 method), 96
set_soft_cal_curve_pt_100() (lakeshore.model_335.Model335 method), 65
set_soft_cal_curve_pt_100() (lakeshore.model_336.Model336 method), 96
set_soft_cal_curve_pt_1000() (lakeshore.model_335.Model335 method), 66
set_soft_cal_curve_pt_1000() (lakeshore.model_336.Model336 method), 97
set_standard_event_enable_mask() (lakeshore.fast_hall_controller.FastHall method), 29
set_standard_event_enable_mask() (lakeshore.model_155.PrecisionSource method), 199
set_standard_event_enable_mask() (lakeshore.model_224.Model224 method), 165
set_standard_event_enable_mask() (lakeshore.model_335.Model335 method), 85

set_standard_event_enable_mask() (*lakeshore.model_336.Model336 method*), 118
set_standard_event_enable_mask() (*lakeshore.model_372.Model372 method*), 153
set_standard_event_enable_mask() (*lakeshore.ssm_system.SSMSystem method*), 47
set_standard_event_enable_mask() (*lakeshore.teslameter.Teslameter method*), 18
set_still_output() (*lakeshore.model_372.Model372 method*), 136
set_temperature_limit() (*lakeshore.model_335.Model335 method*), 85
set_temperature_limit() (*lakeshore.model_336.Model336 method*), 118
set_temperature_limit() (*lakeshore.model_372.Model372 method*), 154
set_to_factory_defaults() (*lakeshore.model_224.Model224 method*), 166
set_voltage_amplitude() (*lakeshore.ssm_source_module.SourceModule method*), 51
set_voltage_limit() (*lakeshore.model_155.PrecisionSource method*), 196
set_voltage_limit() (*lakeshore.ssm_source_module.SourceModule method*), 52
set_voltage_mode_current_protection() (*lakeshore.model_155.PrecisionSource method*), 196
set_voltage_offset() (*lakeshore.ssm_source_module.SourceModule method*), 51
set_voltage_range() (*lakeshore.model_155.PrecisionSource method*), 196
set_wait_to_continue() (*lakeshore.model_224.Model224 method*), 166
set_warmup_output() (*lakeshore.model_372.Model372 method*), 136
set_warmup_supply() (*lakeshore.model_335.Model335 method*), 73
set_warmup_supply_parameter() (*lakeshore.model_336.Model336 method*), 106
set_website_login() (*lakeshore.model_224.Model224 method*), 170
set_website_login() (*lakeshore.model_336.Model336 method*), 100
set_website_login() (*lakeshore.model_372.Model372 method*), 142
SETPOINT_1 (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
SETPOINT_2 (*lakeshore.model_335.Model335DisplayInputChannel attribute*), 91
SettingsProfiles (*class in lakeshore.ssm_settings_profiles*), 59
setup_ac_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
setup_dc_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
setup_lock_in_measurement() (*lakeshore.ssm_measure_module.MeasureModule method*), 57
setup_sample_heater() (*lakeshore.model_372.Model372 method*), 141
setup_warmup_heater() (*lakeshore.model_372.Model372 method*), 141
SEVEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
SIX (*lakeshore.model_372.Model372InputChannel attribute*), 159
SIXTEEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
SMALL (*lakeshore.temperature_controllers.DisplayFieldsSize attribute*), 125
SMALL_16 (*lakeshore.model_224.Model224NumberOfFields attribute*), 183
SMALL_8 (*lakeshore.temperature_controllers.DisplayFields attribute*), 125
SourceModule (*class in lakeshore.ssm_source_module*), 47
SSMSystem (*class in lakeshore.ssm_system*), 42
SSMSystemMeasureModuleOperationRegister (*class in lakeshore.ssm_measure_module*), 60
SSMSystemModuleQuestionableRegister (*class in lakeshore.ssm_base_module*), 60
SSMSystemOperationRegister (*class in lakeshore.ssm_system*), 60
SSMSystemQuestionableRegister (*class in*

lakeshore.ssm_system), 60
SSMSystemSourceModuleOperationRegister (*class in lakeshore.ssm_source_module*), 60
StandardEventRegister (*class in lakeshore.fast_hall_controller*), 40
StandardEventRegister (*class in lakeshore.model_372*), 158
StandardEventRegister (*class in lakeshore.temperature_controllers*), 92
start_contact_check_hbar() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_contact_check_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_contact_check_vdp_optimized() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_dc_hall_hbar() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_dc_hall_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_fasthall_link_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_fasthall_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_four_wire() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_resistivity_hbar() (*lakeshore.fast_hall_controller.FastHall method*), 25
start_resistivity_link_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
start_resistivity_vdp() (*lakeshore.fast_hall_controller.FastHall method*), 24
STATIC_IP (*lakeshore.temperature_controllers.LanStatus attribute*), 125
status_byte_register (*lakeshore.model_336.Model336 attribute*), 95
StatusByteRegister (*class in lakeshore.fast_hall_controller*), 40
STILL (*lakeshore.model_372.Model372OutputMode attribute*), 159
stream_buffered_data() (*lakeshore.teslameter.Teslameter method*), 10
stream_data() (*lakeshore.ssm_system.SSMSystem method*), 42
sweep_current() (*lakeshore.model_155.PrecisionSource method*), 194
sweep_voltage() (*lakeshore.model_155.PrecisionSource method*), 194

T

tare_relative_field() (*lakeshore.teslameter.Teslameter method*), 11
TEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
TEN_KILOHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanc attribute*), 181
TEN_KILOHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResis attribute*), 181
TEN_MICRO_AMPS (*lakeshore.model_224.Model224DiodeExcitationCurr attribute*), 181
TEN_MICROAMPS (*lakeshore.temperature_controllers.DiodeCurrent attribute*), 91
TEN_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
TEN_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
TEN_OHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRa attribute*), 181
TEN_OHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResistanc attribute*), 181
TEN_THOUSAND_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
TEN_THOUSAND_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
TEN_VOLTS (*lakeshore.model_335.Model335DiodeRange attribute*), 89
TEN_VOLTS (*lakeshore.model_336.Model336DiodeRange attribute*), 123
Teslameter (*class in lakeshore.teslameter*), 10
TeslameterOperationRegister (*class in lakeshore.teslameter*), 18
TeslameterQuestionableRegister (*class in lakeshore.teslameter*), 18
THERMOCOUPLE (*lakeshore.model_335.Model335InputSensorType attribute*), 89
THERMOCOUPLE (*lakeshore.model_336.Model336InputSensorType attribute*), 123
THIRTEEN (*lakeshore.model_372.Model372InputChannel attribute*), 159
THIRTY_KILOHMS (*lakeshore.model_224.Model224NTCRTDSensorResis attribute*), 181
THIRTY_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 89
THIRTY_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123

THIRTY_OHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181
THIRTY_OHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResistanceRange attribute*), 181
THIRTY_THOUSAND_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 90
THIRTY_THOUSAND_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
THREE (*lakeshore.model_372.Model372InputChannel attribute*), 159
THREE_HUNDRED_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 90
THREE_HUNDRED_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
THREE_HUNDRED_OHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181
THREE_HUNDRED_OHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResistanceRange attribute*), 181
THREE_KILOHMS (*lakeshore.model_224.Model224NTCRTDSensorResistanceRange attribute*), 181
THREE_KILOHMS (*lakeshore.model_224.Model224PlatinumRTDSensorResistanceRange attribute*), 181
THREE_QUARTERS (*lakeshore.temperature_controllers.BrightnessLevel attribute*), 90
THREE_THOUSAND_OHM (*lakeshore.model_335.Model335RTDRange attribute*), 90
THREE_THOUSAND_OHM (*lakeshore.model_336.Model336RTDRange attribute*), 123
turn_relay_off() (*lakeshore.model_224.Model224 method*), 177
turn_relay_off() (*lakeshore.model_335.Model335 method*), 85
turn_relay_off() (*lakeshore.model_336.Model336 method*), 119
turn_relay_off() (*lakeshore.model_372.Model372 method*), 154
turn_relay_on() (*lakeshore.model_224.Model224 method*), 177
turn_relay_on() (*lakeshore.model_335.Model335 method*), 85
turn_relay_on() (*lakeshore.model_336.Model336 method*), 119
turn_relay_on() (*lakeshore.model_372.Model372 method*), 154
TWELVE (*lakeshore.model_372.Model372InputChannel attribute*), 159
TWENTY_FIVE_OHMS (*lakeshore.model_372.Model372InputChannel attribute*), 160
TWENTY_FIVE_OHMS (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
TWO_INPUT_B (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
TWO_LOOP (*lakeshore.model_335.Model335DisplaySetup attribute*), 91
TWO_POINT_FIVE_VOLTS (*lakeshore.model_335.Model335DiodeRange attribute*), 89
TWO_POINT_FIVE_VOLTS (*lakeshore.model_336.Model336DiodeRange attribute*), 123

U

UNIPOLAR (*lakeshore.temperature_controllers.Polarity attribute*), 89
update() (*lakeshore.ssm_settings_profiles.SettingsProfiles method*), 59
USB (*lakeshore.model_224.Model224RemoteInterface attribute*), 181
USB (*lakeshore.temperature_controllers.Interface attribute*), 125
use_ac_coupling() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
use_ac_coupling() (*lakeshore.ssm_source_module.SourceModule method*), 49
use_dc_coupling() (*lakeshore.ssm_measure_module.MeasureModule method*), 54
use_dc_coupling() (*lakeshore.ssm_source_module.SourceModule method*), 49

V

VAD_CONTROL (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
VAD_MEASUREMENT (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
VCM_NEG (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
VCM_POS (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
VDIF (*lakeshore.model_372.Model372MonitorOutputSource attribute*), 160
VOLTAGE (*lakeshore.model_335.Model335HeaterOutType attribute*), 90
VOLTAGE (*lakeshore.model_372.Model372SensorExcitationMode attribute*), 160
VOLTAGE_OFF (*lakeshore.model_335.Model335HeaterVoltageRange attribute*), 91

VOLTAGE_OFF (*lakeshore.model_336.Model336HeaterVoltageRange attribute*), 124
VOLTAGE_ON (*lakeshore.model_335.Model335HeaterVoltageRange attribute*), 91
VOLTAGE_ON (*lakeshore.model_336.Model336HeaterVoltageRange attribute*), 124
VOLTS_PER_KELVIN (*lakeshore.model_224.Model224CurveFormat attribute*), 183
VOLTS_PER_KELVIN (*lakeshore.model_240.Model240CurveFormat attribute*), 191
VOLTS_PER_KELVIN (*lakeshore.temperature_controllers.CurveFormat attribute*), 88

W

WARMUP (*lakeshore.model_372.Model372OutputMode attribute*), 159
WARMUP_HEATER (*lakeshore.model_372.Model372DisplayInfo attribute*), 161
WARMUP_HEATER_ZONE (*lakeshore.model_372.Model372RelayControlMode attribute*), 160
WARMUP_SUPPLY (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
WARMUP_SUPPLY (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123

Z

ZONE (*lakeshore.model_335.Model335HeaterOutputMode attribute*), 90
ZONE (*lakeshore.model_336.Model336HeaterOutputMode attribute*), 123
ZONE (*lakeshore.model_372.Model372OutputMode attribute*), 159