Lake Shore Python Driver Documentation

Release 1.8.1

Lake Shore Cryotronics, Inc.

Jan 25, 2024

CONTENTS

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

ONE

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

TWO

TABLE OF CONTENTS

2.1 Installation

2.1.1 Python version

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

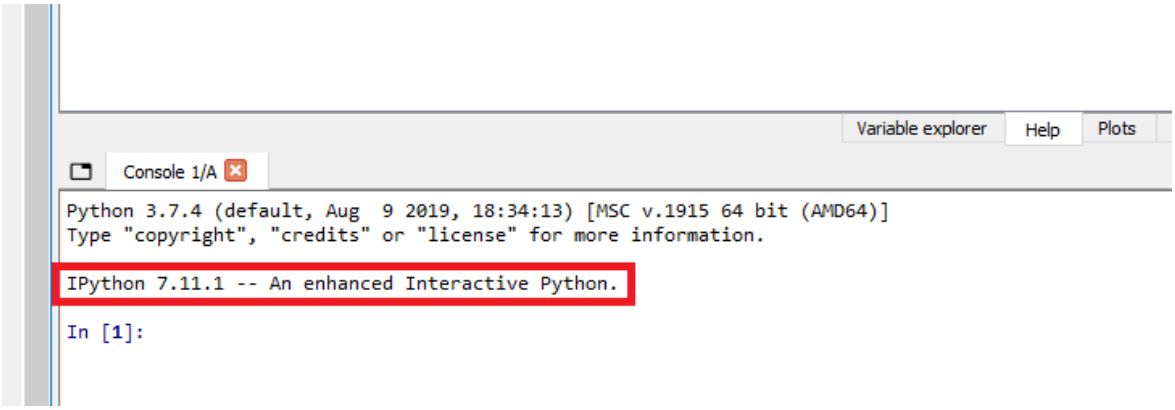
2.1.2 Install the Lake Shore Python driver

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

```
pip install lakeshore
```

2.1.3 Installing the driver through Spyder

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



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

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

Back in the Spyder console, type:

```
pip install lakeshore
```

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

Updating the driver

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

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

2.2 Getting Started

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

2.2.1 A simple example

```
from lakeshore import Model155

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

2.2.2 Making Connections

Connecting to a specific instrument

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

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

```
from lakeshore import Teslameter

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

or the COM port it is connected to:

```
from lakeshore import FastHall

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

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

```
from lakeshore import Model372

my_instrument = Model372(9600)
```

Connecting over TCP

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

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

```
from lakeshore import Model155

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

2.2.3 Commands and queries

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

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

```
from lakeshore import Model155

my_instrument = Model155()

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

2.2.4 SCPI commands and queries

Grouping multiple commands & queries

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

```
from lakeshore import 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 Alternative instrument connections

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

GPIB connection with PyVISA

```
import pyvisa
from lakeshore import SSMSystem

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

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

2.3.6 Python 2 compatibility

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

2.4 Supported Instruments

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

2.4.1 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')
```

(continues on next page)

(continued from previous page)

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

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

Instrument class methods

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

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

stream_buffered_data(*length_of_time_in_seconds*, *sample_rate_in_ms*)

Yield a generator object for the buffered field data.

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

Args:

length_of_time_in_seconds (float):
The period of time over which to stream the data.

sample_rate_in_ms (int):
The averaging window (sampling period) of the instrument.

Returns:
A generator object that returns the data as datapoint tuples.

get_buffered_data_points(*length_of_time_in_seconds*, *sample_rate_in_ms*)

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

Args:

length_of_time_in_seconds (float):
The period of time over which to collect the data.

sample_rate_in_ms (int):
The averaging window (sampling period) of the instrument.

Returns:
The data as a list of datapoint tuples.

log_buffered_data_to_file(*length_of_time_in_seconds*, *sample_rate_in_ms*, *file*)

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

Args:

length_of_time_in_seconds (float):
The period of time over which to collect the data.

sample_rate_in_ms (int):
The averaging window (sampling period) of the instrument.

file (file_object):
Field measurement data will be written to this file object in a CSV format.

get_dc_field()

Returns the DC field reading.

get_dc_field_xyz()

Returns the DC field reading.

get_rms_field()

Returns the RMS field reading.

get_rms_field_xyz()

Returns the RMS field reading.

get_frequency()

Returns the field frequency reading.

get_max_min()

Returns the maximum and minimum field readings respectively.

get_max_min_peaks()

Returns the maximum and minimum peak field readings respectively.

reset_max_min()

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

get_temperature()

Returns the temperature reading.

get_probe_information()

Returns a dictionary of probe data.

get_relative_field()

Returns the relative field value.

tare_relative_field()

Copies the current field reading to the relative baseline value.

get_relative_field_baseline()

Returns the relative field baseline value.

set_relative_field_baseline(*baseline_field*)

Configures the relative baseline value.

Args:

baseline_field (float):
A field units value that will act as the zero field for the relative measurement.

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

Configures the field measurement settings.

Args:

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

autorange (bool):
Chooses whether the instrument automatically selects the best range for the measured value.

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

averaging_samples (int):
The number of field samples to average. Each sample is 10 milliseconds of field information.

`get_field_measurement_setup()`

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

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

Configures how temperature compensation is applied to the field readings.

Args:

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

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

`get_temperature_compensation_source()`

Returns the source of temperature measurement for field compensation.

`get_temperature_compensation_manual_temp()`

Returns the manual temperature setting value in Celsius.

`configure_field_units(units='TESLA')`

Configures the field measurement units of the instrument.

Args:

units (str):
A unit of magnetic field. Options are: "TESLA", or "GAUSS".

`get_field_units()`

Returns the magnetic field units of the instrument.

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

Configures the limits of the field control output.

Args:

voltage_limit (float):
The maximum voltage permitted at the field control output. Must be between 0 and 10V.

slew_rate_limit (float):
The maximum rate of change of the field control output voltage in volts per second.

`get_field_control_limits()`

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

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

Configure the field control mode and state.

Args:

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

output_enabled (bool):
Turn the field control voltage output on or off.

`get_field_control_output_mode()`

Returns the mode and state of the field control output.

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

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

Args:

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

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

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

`get_field_control_pid()`

Returns the gain, integral, and ramp rate.

`set_field_control_setpoint(setpoint)`

Sets the field control setpoint value in field units.

`get_field_control_setpoint()`

Returns the field control setpoint.

`set_field_control_open_loop_voltage(output_voltage)`

Sets the field control open loop voltage.

`get_field_control_open_loop_voltage()`

Returns the field control open loop voltage.

`set_analog_output(analog_output_mode)`

Configures what signal is provided by the analog output BNC.

`set_analog_output_signal(analog_output_mode)`

Configures what signal is provided by the analog output BNC.

Args:

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

`configure_corrected_analog_output_scaling(scale_factor, baseline)`

Configures the conversion between field reading and analog output voltage.

Args:

scale_factor (float):
Scale factor in volts per unit field.

baseline (float):
The field value at which the analog output voltage is zero.

`get_corrected_analog_output_scaling()`

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

`get_analog_output()`

Returns what signal is being provided by the analog output.

`get_analog_output_signal()`

Returns what signal is being provided by the analog output.

`enable_high_frequency_filters()`

Applies filtering to the high frequency RMS measurements.

`disable_high_frequency_filters()`

Turns off filtering of the high frequency mode measurements.

`set_frequency_filter_type(`*filter_type*`)`

Configures which filter is applied to the high frequency measurements.

Args:

filter_type (str):
Options: "LPASS" (low pass filter), "HPASS" (high pass filter), or "BPASS" (band pass filter).

`get_frequency_filter_type()`

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

`get_low_pass_filter_cutoff()`

Returns the cutoff frequency setting of the low pass filter.

`set_low_pass_filter_cutoff(`*cutoff_frequency*`)`

Configures the low pass filter cutoff.

Args:

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

`get_high_pass_filter_cutoff()`

Returns the cutoff frequency setting of the low pass filter.

`set_high_pass_filter_cutoff(`*cutoff_frequency*`)`

Configures the high pass filter cutoff.

Args:

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

`get_band_pass_filter_center()`

Returns the center of the band pass filter.

`set_band_pass_filter_center(`*center_frequency*`)`

Configures the band pass filter parameters.

Args:

center_frequency (float):
The frequency at which the gain of the filter is 1.

`enable_qualifier()`

Enables the qualifier.

`disable_qualifier()`

Disables the qualifier.

`is_qualifier_condition_met()`

Returns whether the qualifier condition is met.

`enable_qualifier_latching()`

Enables the qualifier condition latching.

`disable_qualifier_latching()`

Disables the qualifier condition latching.

`get_qualifier_latching_setting()`

Returns whether the qualifier latches.

`set_qualifier_latching_setting(`*latching*`)`

Sets whether the qualifier latches.

Args:

latching (bool):
Determines whether the qualifier latches.

`reset_qualifier_latch()`

Resets the condition status of the qualifier.

`get_qualifier_configuration()`

Returns the threshold mode and field threshold values.

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

Sets the threshold condition of the qualifier.

Args:

mode (str):
The type of threshold condition used by the qualifier. Options: "OVER", "UNDER", "BETWEEN", "OUTSIDE", "ABSBETWEEN", or "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 an SCPI command or multiple commands to the instrument.

Args:

commands (str):
Any number of SCPI commands.

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

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

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

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

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

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

get_operation_event_enable_mask()

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

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

get_operation_events()

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

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

get_present_operation_status()

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

get_present_questionable_status()

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

get_questionable_event_enable_mask()

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

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

get_questionable_events()

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

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

get_service_request_enable_mask()

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

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

get_standard_event_enable_mask()

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

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

get_standard_events()

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

get_status_byte()

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

`modify_operation_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_questionable_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_service_request_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_standard_event_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`query`(**queries*, *check_errors=True*)

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

Args:

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

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

Returns:
The instrument query response as a string.

`reset_measurement_settings`()

Resets measurement settings to their default values.

reset_status_register_masks()

Resets status register masks to preset values.

set_operation_event_enable_mask(*register_mask*)

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

set_questionable_event_enable_mask(*register_mask*)

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

set_service_request_enable_mask(*register_mask*)

Configures values of the service request enable register bits.

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

Args:

register_mask (StatusByteRegister):
A StatusByteRegister class object with all bits configured true or false.

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

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

Args:

register_mask (StandardEventRegister):
A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

Status register classes

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

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

Class object representing the operation status register.

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

Class object representing the questionable status register.

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

Class object representing the status byte register.

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

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

Class object representing the standard event register.

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

Model 425 Gaussmeter

The Model 425 Gaussmeter provides field strength measurements.

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

Instrument methods

class lakeshore.model_425.**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.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

2.4.2 Magnet System Power Supplies

Model 643 & 648 Electromagnet Power Supplies

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

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

Example Scripts

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

Outputting current and measuring voltage on a Model 643

```
from lakeshore import Model643
from time import sleep

# Connect to 643 over USB
my_power_supply = Model643()

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

Setting up magnet water and outputting current from a Model 648

```
from lakeshore import Model648

# Connect to 648 over USB
my_power_supply = Model648()

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

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

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

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

Querying the hardware error status register from a Model 648

```
from lakeshore import Model648

# Connect to 648 over USB
my_power_supply = Model648()

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

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

# Unmask desired error bits
register_mask.output_over_current = True
```

(continues on next page)

(continued from previous page)

```
register_mask.output_over_voltage = True
register_mask.temperature_fault = True

# Set hardware errors mask
my_power_supply.set_hardware_error_enable_mask(register_mask)

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

Instrument class methods

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

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

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

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

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

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

Class object representing the status byte register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

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

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

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

class EMPowerSupplyOperationEventRegister(*compliance*, *ramp_done*, *power_limit*)

Class object representing the operation event register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

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

Class object representing the hardware errors register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

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

Class object representing the operational errors register LSB to MSB.

classmethod from_integer(*integer_representation*)

Creates the register object from an integer representation value.

to_integer()

Translates the register object to an integer representation value.

command(*command_string*, *check_errors=True*)

Send a command to the instrument.

Args:

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

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

Send a query to the instrument and return the response

Args:

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

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

Returns:

str: The instrument query response as a string.

`set_limits`(*max_current*, *max_ramp_rate*)

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

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

Args:

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

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

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

`get_limits`()

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

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

Returns:

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

`set_ramp_rate`(*ramp_rate*)

Sets the output current ramp rate.

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

Args:

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

(0.0001 - 50.000 A/s).

`get_ramp_rate`()

Returns the output current ramp rate.

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

Returns:

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

`set_ramp_segment`(*segment*, *current*, *ramp_rate*)

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

Args:

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

`get_ramp_segment`(*segment*)

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

Args:

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

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

`set_ramp_segments_enable`(*state*)

Specifies if ramp segments are to be used.

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

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

`get_ramp_segments_enable`()

Returns if ramp segments are to be used.

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

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

`set_current`(*current*)

Sets the output current setting.

The setting value is limited by set_limit.

Args:

current (float): The output current value that the output will ramp to at the present ramp rate. The Model
643 bounds are 0.0000 - +/-70.1000 A. The Model 648 bounds are 0.0000 - +/-135.1000 A.

`get_current`()

Returns the output current setting.

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

`get_measured_current`()

Returns actual measured output current.

Returns:
float: Measured output current.

`get_measured_voltage`()

Returns actual output voltage measured at the power supply terminals.

Returns:
float: Measured output voltage.

`stop_output_current_ramp`()

Stops the output current ramp.

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

`set_internal_water`(*mode*)

Configures the internal water mode.

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

`get_internal_water()`

Returns the internal water mode.

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

`set_magnet_water(mode)`

Configures the magnet water mode.

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

`get_magnet_water()`

Returns the magnet water mode.

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

`set_display_brightness(brightness_level)`

Specifies display brightness.

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

`get_display_brightness()`

Returns display brightness.

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

`set_front_panel_lock(lock_state, code)`

Sets the lock status of the front panel keypad.

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

`get_front_panel_status()`

Returns what lock state the front panel keypad is in.

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

`get_front_panel_lock_code()`

Returns the lock code for the front panel.

Returns:
int: Front panel lock code.

`set_programming_mode(mode)`

Sets the current programming mode of the instrument.

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

`get_programming_mode()`

Returns the current programming mode of the instrument.

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

`set_ieee_488`(*terminator*, *eoi_enable*, *address*)

Configures the IEEE-488 interface.

Args:

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

`get_iee_488`()

Returns IEEE-488 interface configuration.

Returns:

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

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

`set_ieee_interface_mode`(*mode*)

Sets the interface mode of the instrument.

Args:

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

`get_ieee_interface_mode`()

Returns the interface mode of the instrument.

Returns:

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

`set_factory_defaults`()

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

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

`reset_instrument`()

Sets the controller parameters to power-up settings.

Use the set_factory_defaults command to set factory-defaults.

`clear_interface`()

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

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

`get_self_test`()

Returns result of instrument self test completed at power up.

Returns:

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

`set_service_request_enable_mask`(*register_mask*)

Configures the Service Request Enable Register.

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

Args:

register_mask (ElectromagnetPowerSupply.EMPowerSupplyServiceRequestEnableRegister): Register configuration object.

`get_service_request_enable_mask()`

Returns Service Request Enable Register configuration.

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyServiceRequestEnableRegister: Register configuration object.

`get_status_byte()`

Returns state of the Status Byte Register.

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyStatusByteRegister: Register state object.

`set_standard_event_status_enable_mask(`*register_mask*`)`

Configures Standard Event Status Enable Register group.

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

Args:

register_mask (ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister): Register configuration
object.

`get_standard_event_status_enable_mask()`

Returns Standard Event Status Enable Register configuration.

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister: Register configuration object.

`get_standard_event_status_event()`

Returns state of the Standard Event Status Register.

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyStandardEventStatusRegister: Register state object.

set_operation_event_enable_mask(*register_mask*)

Configures the Operational Event Enable Register.

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

Args:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Register configuration object.

get_operation_event_enable_mask()

Returns Operational Event Enable Register configuration.

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyOperationEventRegister: Register configuration object.

get_operation_event_condition()

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

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

Returns:

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

get_operation_event_event()

Returns the latched state of the operation event bits.

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

Returns:

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

set_hardware_error_enable_mask(*register_mask*)

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

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

Args:

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

`get_hardware_error_enable_mask()`

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

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Register mask configuration object.

`get_hardware_error_condition()`

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

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

Returns:

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

`get_hardware_error_event()`

Returns the latched state of the hardware error bits.

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

Returns:

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

`set_operational_error_enable_mask(`*register_mask*`)`

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

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

Args:

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

`get_operational_error_enable_mask()`

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

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

Returns:

ElectromagnetPowerSupply.EMPowerSupplyHardwareErrorsRegister: Register mask configuration object.

`get_operational_error_condition()`

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

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

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

`get_operational_error_event()`

Returns the latched state of the operational error bits.

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

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

`connect_tcp(ip_address, tcp_port, timeout)`

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

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

Establish a serial USB connection.

`disconnect_tcp()`

Disconnect the TCP connection.

`disconnect_usb()`

Disconnect the USB connection.

`write(command_string)`

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

2.4.3 Materials Characterization

M91 Fast Hall Controller

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

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

Example Scripts

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

Fast Hall Full Sample Analysis

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

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

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

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

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

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

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

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

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

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

Fast Hall Record Contact Check Data

```
from lakeshore import FastHall, ContactCheckManualParameters

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

# Create a contact check parameters object with desired settings to run a manual Contact␣
↪Check measurement
ccheck_settings = ContactCheckManualParameters(excitation_type='CURRENT',
                                               excitation_start_value=-10e-6,
                                               excitation_end_value=10e-6,
                                               compliance_limit=1.5,
                                               number_of_points=20)
# Create a file to write data into
file = open("fasthall_data.csv", "w")

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

Instrument class methods

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

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

get_contact_check_running_status()

Indicates if the contact check measurement is running.

get_fasthall_running_status()

Indicates if the FastHall measurement is running.

get_four_wire_running_status()

Indicates if the four wire measurement is running.

get_resistivity_running_status()

Indicates if the resistivity measurement is running.

get_dc_hall_running_status()

Indicates if the DC Hall measurement is running.

get_dc_hall_waiting_status()

Indicates if the DC hall measurement is running.

continue_dc_hall()

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

start_contact_check_vdp_optimized(*settings*)

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

Args:

settings(ContactCheckOptimizedParameters):

start_contact_check_vdp(*settings*)

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

Args:

settings(ContactCheckManualParameters):

start_contact_check_hbar(*settings*)

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

Args:

settings(ContactCheckManualParameters):

start_fasthall_vdp(*settings*)

Performs a FastHall measurement.

Args:
settings (FastHallManualParameters):

start_fasthall_link_vdp(*settings*)

Starts a FastHall measurement with provided link parameters.

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

Args:
settings (FastHallLinkParameters):

start_four_wire(*settings*)

Performs a Four wire measurement.

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

Args:
settings(FourWireParameters):

start_dc_hall_vdp(*settings*)

Performs a DC hall measurement for a Hall Bar sample.

Args:
settings(DCHallParameters):

start_dc_hall_hbar(*settings*)

Performs a DC hall measurement for a Hall Bar sample.

Args:
settings(DCHallParameters):

start_resistivity_vdp(*settings*)

Performs a resistivity measurement on a Van der Pauw sample.

Args:
settings(ResistivityManualParameters):

start_resistivity_link_vdp(*settings*)

Performs a resistivity measurement with provided link settings.

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

Args:
settings(ResistivityLinkParameters):

start_resistivity_hbar(*settings*)

Performs a resistivity measurement on a hall bar sample.

Args:
settings(ResistivityManualParameters):

get_contact_check_setup_results()

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

`get_contact_check_measurement_results()`

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

`get_fasthall_setup_results()`

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

`get_fasthall_measurement_results()`

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

`get_four_wire_setup_results()`

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

`get_four_wire_measurement_results()`

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

`get_dc_hall_setup_results()`

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

`get_dc_hall_measurement_results()`

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

`get_resistivity_setup_results()`

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

`get_resistivity_measurement_results()`

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

`run_complete_contact_check_optimized(`*settings*`)`

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

Args:

settings(ContactCheckOptimizedParameters):

Returns:

The measurement results as a dictionary.

`run_complete_contact_check_manual(`*settings*, *sample_type*`)`

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

Args:

settings (ContactCheckManualParameters):
Object with settings for FastHall link setup.

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

Returns:

The measurement results as a dictionary.

`run_complete_fasthall_link(`*settings*`)`

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

Args:

settings(FastHallLinkParameters):
Object with settings for FastHall link setup.

Returns:

The measurement results as a dictionary.

run_complete_fasthall_manual(*settings*)

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

Args:

settings(FastHallManualParameters):
Object with settings for FastHall link setup.

Returns:
The measurement results as a dictionary.

run_complete_four_wire(*settings*)

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

Args:
settings(FourWireParameters):

Returns:
The measurement results as a dictionary.

run_complete_dc_hall(*settings*, *sample_type*)

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

Args:

settings(DCHallParameters):
Object with settings for FastHall link setup.

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

Returns:
The measurement results as a dictionary.

run_complete_resistivity_link(*settings*)

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

Args:
settings(ResistivityLinkParameters):

Returns:
The measurement results as a dictionary.

run_complete_resistivity_manual(*settings*, *sample_type*)

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

Args:

settings(ResistivityManualParameters):
Object with settings for manual resistivity setup.

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

Returns:
The measurement results as a dictionary.

reset_contact_check_measurement()

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

reset_fasthall_measurement()

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

reset_four_wire_measurement()

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

reset_dc_hall_measurement()

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

reset_resistivity_measurement()

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

command(**commands*, *check_errors=True*)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):
Any number of SCPI commands.

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

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

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

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

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

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

get_operation_event_enable_mask()

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

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

get_operation_events()

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

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

get_present_operation_status()

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

get_present_questionable_status()

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

get_questionable_event_enable_mask()

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

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

get_questionable_events()

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

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

get_service_request_enable_mask()

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

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

get_standard_event_enable_mask()

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

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

get_standard_events()

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

get_status_byte()

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

modify_operation_register_mask(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

modify_questionable_register_mask(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

modify_service_request_mask(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

modify_standard_event_register_mask(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`query`(**queries*, *check_errors=True*)

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

Args:

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

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

Returns:
The instrument query response as a string.

`reset_measurement_settings`()

Resets measurement settings to their default values.

`reset_status_register_masks`()

Resets status register masks to preset values.

`set_operation_event_enable_mask`(*register_mask*)

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

`set_questionable_event_enable_mask`(*register_mask*)

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

`set_service_request_enable_mask`(*register_mask*)

Configures values of the service request enable register bits.

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

Args:

register_mask (StatusByteRegister):
A StatusByteRegister class object with all bits configured true or false.

`set_standard_event_enable_mask`(*register_mask*)

Configures values of the standard event enable register bits.

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

Args:

register_mask (StandardEventRegister):
A StandardEventRegister class object with all bits configured true or false.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

Settings classes

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

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

Class object representing the operation status register.

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

Class object representing the questionable status register.

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

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

__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” and “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" which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3A for voltage excitation, or volts in the range of 0 to 10.0V for 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, or a 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, or 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, or 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, or an 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" or "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" which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3A for voltage excitation, or volts in the range of 0 to 10.0V for 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, or a 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, or a 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, or a 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: “CURRENT” or “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” which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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” which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3A for voltage excitation, or volts in the range of 0 to 10.0V for current excitation.

excitation_measurement_range (float or str):
Excitation measurement range based on the excitation type. Options are: “AUTO” which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or amps in the range of -100e-3 to 100e-3 A for 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, or a 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, or a 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: "CURRENT", or "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" for sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" for sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3A for voltage excitation, or volts in the range of 0 to 10.0V for 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 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, or 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" or "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" which sets the range to the best fit range for a given excitation value, or a floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or floating point number of either volts in the range of 0 to 10.0V for voltage excitation, or 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3A for voltage excitation, or volts in the range of 0 to 10.0V for 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, or a 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, or a 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" which sets the range to the best fit range for a given excitation value, or a floating point number of either amps in the range of 0 to 100e-3 A for voltage excitation, or volts in the range of 0 to 10.0V for 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, or a 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*)

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*)

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_current_amplitude(0.001)

# Set the voltage measure module to reference the source 1 module with a 100 ms time␣
↪constant
```

(continues on next page)

(continued from previous page)

```
voltage_measure.setup_lock_in_measurement('S1', 0.1)

# Enable the source output
balanced_current_source.enable()

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

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

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

List settings profiles and restore a profile

```
from lakeshore import SSMSystem

# Connect to instrument via USB
my_M81 = SSMSystem()

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

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

Stream data

```
from lakeshore import SSMSystem

# Connect to instrument via USB
my_M81 = SSMSystem()

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

(continues on next page)

(continued from previous page)

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

Sweep a BCS-10 from 0 mA to 100 mA

```
from lakeshore import SSMSystem

# Connect to instrument via USB
my_ssm = SSMSystem()

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

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

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

# stream 1,000 samples of synchronized data at 1,000 samples per second and␣
↪simultaneously start the sweep
s1_bcs.enable()
stream_data = my_ssm.get_data(1000, 1000,
                              [my_ssm.DataSourceMnemonic.SOURCE_AMPLITUDE, 1],
                              [my_ssm.DataSourceMnemonic.MEASURE_DC, 1])

print(stream_data)
```

SSMS instrument methods

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

Class for interaction with the M81 instrument.

load_modules()

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

get_num_measure_channels()

Returns the number of measure channels supported by the instrument.

get_num_source_channels()

Returns the number of source channels supported by the instrument

get_source_module(*port_number*)

Returns a SourceModule instance for the given port number.

get_source_pod(*port_number*)

Alias of get_source_module.

get_source_module_by_name(*module_name*)

Return the SourceModule instance that matches the specified name.

get_measure_module(*port_number*)

Returns a MeasureModule instance for the given port number.

get_measure_pod(*port_number*)

Alias of get_measure_module.

get_measure_module_by_name(*module_name*)

Return the MeasureModule instance that matches the specified name.

get_multiple(**data_sources*)

This function is deprecated. Use fetch_multiple() instead.

Deprecated since version 1.5.4: Use fetch_multiple instead.

get_multiple_min_max_values(**data_sources*)

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

Args:

data_sources (str, int):
Pairs of (DATASOURCE_MNEMONIC, CHANNEL_INDEX).

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

Generator object to stream data from the instrument.

Args:

rate (int):
Desired transfer rate in points/sec.

num_points (int):
Number of points to return. None to stream indefinitely.

data_sources (SSMSystemDataSourceMnemonic or str, int):
Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Yields:
A single row of stream data as a tuple.

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

Like stream_data, but returns a list.

Args:

rate (int):
Desired transfer rate in points/sec.

num_points (int):
Number of points to return.

data_sources (SSMSystemDataSourceMnemonic or str, int):
Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:
All available stream data as a list of tuples.

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

Like stream_data, but logs directly to a CSV file.

Args:

rate (int):
Desired transfer rate in points/sec.

file (IO):
File to log to.

num_points (int):
Number of points to log.

data_sources (SSMSystemDataSourceMnemonic or str, int):
Pairs of (DATA_SOURCE, CHANNEL_INDEX).

write_header (bool):
If true, a header row is written with column names.

`get_ref_in_edge`()

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

`set_ref_in_edge`(*edge*)

Sets the active edge of the reference input.

Args:

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

`get_ref_out_source`()

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

`set_ref_out_source`(*ref_out_source*)

Sets the channel used for the reference output.

Args:

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

`get_ref_out_state()`

Returns the enable state of reference out.

`set_ref_out_state(ref_out_state)`

Sets the enable state of reference out.

Args:

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

`enable_ref_out()`

Sets the enable state of reference out to True.

`disable_ref_out()`

Sets the enable state of reference out to False.

`configure_ref_out(ref_out_source, ref_out_state=True)`

Configure the reference output.

Args:

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

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

`get_mon_out_mode()`

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

`set_mon_out_mode(mon_out_source)`

Sets the channel used for the monitor output.

Args:

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

`get_mon_out_state()`

Returns the enable state of monitor out.

`set_mon_out_state(mon_out_state)`

Sets the enable state of monitor out.

Args:

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

`enable_mon_out()`

Sets the enable state of monitor out to True.

`disable_mon_out()`

Sets the enable state of monitor out to False.

`configure_mon_out(mon_out_source, mon_out_state=True)`

Configure the monitor output.

Args:

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

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

`get_mon_out_scale()`

Returns the monitor out scaling factor of the configured module.

`get_head_cal_datetime()`

Returns the date and time of the head calibration.

`get_head_cal_temperature()`

Returns the temperature of the head calibration.

`get_head_self_cal_status()`

Returns the status of the last head self calibration.

`get_head_self_cal_datetime()`

Returns the datetime of the last head self calibration.

`get_head_self_cal_temperature()`

Returns the temperature of the last head self calibration.

`run_head_self_calibration()`

Runs a self calibration for the head.

`reset_head_self_calibration()`

"Restore the factory self calibration.

`set_mon_out_manual_level(manual_level)`

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

Args:

manual_level (float):
The new monitor out manual level.

`get_mon_out_manual_level()`

Returns the manual level of monitor out.

`configure_mon_out_manual_mode(manual_level, mon_out_state=True)`

Configures the monitor output for manual mode.

Args:

manual_level (float):
The new monitor out manual level.

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

`get_line_frequency()`

Returns the line frequency in Hz.

`get_detected_line_frequency()`

Returns the detected line frequency in Hz.

`get_line_frequency_detection_error_status()`

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

fetch_multiple(**data_sources*)

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

Args:

data_sources (SSMSystemDataSourceMnemonic or str, int):
Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:
Tuple of values corresponding to the given data sources.

read_multiple(**data_sources*)

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

Returns values after the measurement is complete.

Args:

data_sources (SSMSystemReadDataSourceMnemonic or str, int):
Variable length list of pairs of (DATA_SOURCE, CHANNEL_INDEX).

Returns:
Tuple of values corresponding to the given data sources.

initiate_sweeps()

Initiates sweeps across all channels.

abort_sweeps()

Aborts in progress sweeps across all channels.

class DataSourceMnemonic(*value*)

Enumeration of M81 data source mnemonics.

class ExcitationType(*value*)

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

class ReadDataSourceMnemonic(*value*)

Enumeration of M81 read data source mnemonics.

class ReferenceModule(*value*)

Class object representing the available source modules

class ResistanceExcitationType(*value*)

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

class ResistanceMode(*value*)

Class object representing the possible resistance modes.

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

Class to configure a parameter sweep on a source module.

class Direction(*value*)

Class object representing the possible directions for sweeping.

class SweepSpacing(*value*)

Class object representing the possible types of sweep spacing.

class SourceSweepType(*value*)

Class representing the available sweep types for a source module.

command(**commands*, *check_errors=True*)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):
Any number of SCPI commands.

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

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

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

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

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

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

get_operation_event_enable_mask()

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

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

get_operation_events()

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

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

get_present_operation_status()

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

get_present_questionable_status()

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

get_questionable_event_enable_mask()

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

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

get_questionable_events()

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

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

get_service_request_enable_mask()

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

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

`get_standard_event_enable_mask()`

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

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

`get_standard_events()`

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

`get_status_byte()`

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

`modify_operation_register_mask(bit_name, value)`

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_questionable_register_mask(bit_name, value)`

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_service_request_mask(bit_name, value)`

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_standard_event_register_mask(bit_name, value)`

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

Args:

bit_name (str):
The name of the bit to modify.

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

`query(*queries, check_errors=True)`

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

Args:

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

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

Returns:
The instrument query response as a string.

`reset_measurement_settings()`

Resets measurement settings to their default values.

`reset_status_register_masks()`

Resets status register masks to preset values.

`set_operation_event_enable_mask(register_mask)`

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

`set_questionable_event_enable_mask(register_mask)`

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

`set_service_request_enable_mask(register_mask)`

Configures values of the service request enable register bits.

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

Args:

register_mask (StatusByteRegister):
A StatusByteRegister class object with all bits configured true or false.

`set_standard_event_enable_mask(register_mask)`

Configures values of the standard event enable register bits.

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

Args:

register_mask (StandardEventRegister):
A StandardEventRegister class object with all bits configured true or false.

`write(command_string)`

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

Source Module methods

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

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

get_multiple(**data_sources*)

This function is deprecated. Use fetch_multiple() instead.

fetch_multiple(**data_sources*)

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

Args:

data_sources (SSMSystemDataSourceMnemonic or str):
Variable length list of data sources.

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

get_name()

Returns the user-settable name of the module.

set_name(*new_name*)

Set the name of the module.

get_notes()

Returns the user-settable notes of the module.

set_notes(*new_note*)

Set the notes of the module.

get_model()

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

get_serial()

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

get_hw_version()

Returns the hardware version of the module.

get_self_cal_status()

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

run_self_cal()

Run a self calibration for the module.

reset_self_cal()

Restore factory self calibration for the module.

get_enable_state()

Returns the output state of the module.

set_enable_state(*state*)

Sets the enable state of the module.

Args:

state (bool):
The new output state.

`enable()`

Sets the enable state of the module to True.

`disable()`

Sets the enable state of the module to False.

`get_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 (SSMSystem.ExcitationType):
The new excitation mode ('CURRENT' or 'VOLTAGE').

`go_to_current_mode()`

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

`go_to_voltage_mode()`

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

`get_shape()`

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

`set_shape(`*shape*`)`

Sets the signal shape of the module.

Args:

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

`get_frequency()`

Returns the excitation frequency of the module.

`set_frequency(`*frequency*`)`

Sets the excitation frequency of the module.

Args:

frequency (float):
The new excitation frequency.

`get_sync_state()`

Returns whether the source channel synchronization feature is engaged

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

`get_sync_source()`

Returns the channel used for frequency synchronization.

`get_sync_phase_shift()`

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

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

Configure the source channel synchronization feature.

Args:

source (str):
The channel used for synchronization ('S1', 'S2', 'S3', or 'RIN'). This channel will follow the frequency set for the specified 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_current_range()

Returns the present current range of the module in Amps.

get_i_range()

Returns the present current range of the module in Amps

Deprecated since version 1.5.4: Use get_current_range instead.

get_current_ac_range()

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

get_i_ac_range()

Returns the present AC current range of the module in Amps

Deprecated since version 1.5.4: Use get_current_ac_range instead.

get_current_dc_range()

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

get_i_dc_range()

Returns the present DC current range of the module in Amps

Deprecated since version 1.5.4: Use get_current_dc_range instead.

`get_current_autorange_status()`

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

`get_i_autorange_status()`

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

Deprecated since version 1.5.4: Use get_current_autorange_status instead.

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

Sets up current ranging for this module.

Args:

autorange (bool):
True to enable automatic range selection. False for manual ranging.

max_level (float):
The largest current that needs to be sourced.

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

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

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

Sets up current ranging for this module

Args:

autorange (bool):
True to enable automatic range selection. False for manual ranging.

max_level (float):
The largest current that needs to be sourced.

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

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

Deprecated since version 1.5.4: Use configure_current_range instead.

`get_current_amplitude()`

Returns the current amplitude for the module in Amps.

`get_i_amplitude()`

Returns the current amplitude for the module in Amps

Deprecated since version 1.5.4: Use get_current_amplitude instead.

`set_current_amplitude(amplitude)`

Sets the current amplitude for the module.

Args:

amplitude (float):
The new current amplitude in Amps.

`set_i_amplitude`(*amplitude*)

Sets the current amplitude for the module

Args:

amplitude (float):
The new current amplitude in Amps

Deprecated since version 1.5.4: Use set_current_amplitude instead.

`get_current_offset`()

Returns the current offset for the module in Amps.

`get_i_offset`()

Returns the current offset for the module in Amps

Deprecated since version 1.5.4: Use get_current_offset instead.

`set_current_offset`(*offset*)

Sets the current offset for the module.

Args:

offset (float):
The new current offset in Amps.

`set_i_offset`(*offset*)

Sets the current offset for the module

Args:

offset (float):
The new current offset in Amps

Deprecated since version 1.5.4: Use set_current_offset instead.

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

Apply DC current.

Args:

level (float):
DC current level in Amps.

output_enable (bool):
Turns the module output on if true; off if false.

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

Apply AC current.

Args:

frequency (float):
Excitation frequency in Hz.

amplitude (float):
Current amplitude in Amps.

offset (float):
Current offset in Amps.

output_enable (bool):
Turns the module output on if true; off if false.

`get_current_limit()`

Returns the current limit enforced by the module in Amps.

`get_i_limit()`

Returns the current limit enforced by the module in Amps

Deprecated since version 1.5.4: Use get_current_limit instead.

`set_current_limit(current_limit)`

Sets the current limit enforced by the module.

Args:

current_limit (float):
The new limit to apply in Amps.

`set_i_limit(i_limit)`

Sets the current limit enforced by the module

Args:

i_limit (float):
The new limit to apply in Amps

Deprecated since version 1.5.4: Use set_current_limit instead.

`get_current_limit_status()`

Returns whether the current limit circuitry is presently engaged.

This limits the current sourced by the module.

`get_i_limit_status()`

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

Deprecated since version 1.5.4: Use get_current_limit_status instead.

`get_voltage_range()`

Returns the present voltage range of the module in Volts.

`get_voltage_ac_range()`

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

`get_voltage_dc_range()`

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

`get_voltage_autorange_status()`

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

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

Sets up voltage ranging for this module.

Args:

autorange (bool):
True to enable automatic range selection. False for manual ranging.

max_level (float):
The largest voltage that needs to be sourced.

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

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

`get_voltage_amplitude()`

Returns the voltage amplitude for the module in Volts.

`set_voltage_amplitude(amplitude)`

Sets the voltage amplitude for the module.

Args:

amplitude (float):
The new voltage amplitude in Volts.

`get_voltage_offset()`

Returns the voltage offset for the module in Volts.

`set_voltage_offset(offset)`

Sets the voltage offset for the module.

Args:

offset (float):
The new voltage offset in Volts.

`apply_dc_voltage(level, output_enable=True)`

Apply DC voltage.

Args:

level (float):
DC voltage level in Volts.

output_enable (bool):
Turns the module output on if true; off if false.

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

Apply AC voltage.

Args:

frequency (float):
Excitation frequency in Hz.

amplitude (float):
Voltage amplitude in Volts.

offset (float):
Voltage offset in Volts.

output_enable (bool):
Turns the module output on if true; off if false.

`get_voltage_limit()`

Returns the voltage limit enforced by the module in Volts.

`set_voltage_limit(voltage_limit)`

Sets the voltage limit enforced by the module.

Args:

voltage_limit (float):
The new limit to apply in Volts.

`get_voltage_limit_status()`

Returns whether the voltage limit circuitry is presently engaged.

This limits the voltage at the output of the module.

`get_present_questionable_status()`

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

`get_questionable_events()`

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

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

`get_questionable_event_enable_mask()`

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

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

`set_questionable_event_enable_mask(`*register_mask*`)`

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

`get_present_operation_status()`

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

`get_operation_events()`

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

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

`get_operation_event_enable_mask()`

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

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

`set_operation_event_enable_mask(`*register_mask*`)`

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

`get_identify_state()`

Returns the identification state for the given pod.

`set_identify_state(`*state*`)`

Configures the identification state for the given pod.

Args:

state (bool):
The desired state for the LED, 1 for identify, 0 for normal state.

get_dark_mode_state()

Returns the dark mode state for the given pod.

set_dark_mode_state(*state*)

Configures the dark mode state for the given pod.

Args:

state (bool):
The desired operation for the LED, 1 for normal mode, 0 for dark mode.

get_voltage_output_limit_high()

Returns the present voltage high output limit.

set_voltage_output_limit_high(*limit*)

Configures the high voltage output limit.

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

Args:

limit (float):
The desired high output limit.

get_voltage_output_limit_low()

Returns the present voltage low output limit.

set_voltage_output_limit_low(*limit*)

Configures the low voltage output limit.

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

Args:

limit (float):
The desired low voltage output limit.

get_current_output_limit_high()

Returns the present current high output limit.

set_current_output_limit_high(*limit*)

Configures the high current output limit.

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

Args:

limit (float):
The desired high output limit.

get_current_output_limit_low()

Returns the present current low output limit.

`set_disable_on_compliance`(*disable_on_compliance*)

Configures the module for disable on compliance.

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

Args:

disable_on_compliance (bool):
1 for the module to disable when in compliance; 0 for the module to remain enabled, even in compliance.

`get_disable_on_compliance`()

Returns the present state of disable on compliance.

`set_current_output_limit_low`(*limit*)

Configures the low current output limit.

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

Args:

limit (float):
The desired low current output limit.

`reset_settings`()

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

`unload`()

Unloads the specified module.

`get_load_state`()

Returns the loaded state for the specified module.

`get_self_cal_datetime`()

Returns the self calibration date and time for the specified module.

`get_self_cal_temperature`()

Returns the self calibration temperature for the specified module.

`get_source_sweep_step_size`(*sweep_type*)

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

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

Args:

sweep_type (SourceSweepType):
The type of sweep for which to return the step size.

`get_source_sweep_time`()

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

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

`get_source_sweep_state`()

Returns the state of the source sweep on the specified module.

`set_sweep_configuration`(*sweep_settings*)

Configures a source sweep for the specified module.

Args:

sweep_settings (SourceSweepSettings):
The configuration for a specific sweep on the specified module.

`get_sweep_configuration`(*sweep_type*)

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

Args:

sweep_type (SourceSweepType):
The sweep type for which to return the sweep settings.

`disable_all_sweeping`()

Disables all source signals that support sweeping on the specified module.

`disable_sweeping`(*sweep_type*)

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

Args:

sweep_type (SourceSweepType):
The type of sweep to disable.

`set_voltage_ramp_configuration`(*stop_amplitude*, *start_amplitude=None*, *slew_rate=1.0*)

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

Uses the smallest possible step size.

Args:

stop_amplitude (float):
The voltage amplitude of the output when the ramp completes.

start_amplitude (float):
The voltage amplitude of the output when the ramp starts. Default is the present amplitude setting.

slew_rate (float):
The rate in volts per second to ramp the output. Default is 1 volt per second.

`set_current_ramp_configuration`(*stop_amplitude*, *start_amplitude=None*, *slew_rate=0.001*)

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

Uses the smallest possible step size.

Args:

stop_amplitude (float):
The current amplitude of the output when the ramp completes.

start_amplitude (float):
The current amplitude of the output when the ramp starts. Default is the present amplitude setting.

slew_rate (float):
The rate in amps per second to ramp the output. Default is 1 mA per second.

Measure Module methods

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

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

get_name()

Returns the user-settable name of the module.

set_name(*new_name*)

Set the name of the module.

get_notes()

Returns the user-settable notes of the module.

set_notes(*new_note*)

Set the notes of the module.

get_model()

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

get_serial()

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

get_hw_version()

Returns the hardware version of the module.

get_self_cal_status()

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

run_self_cal()

Run a self calibration for the module.

reset_self_cal()

Restore factory self calibration for the module.

get_averaging_time()

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

set_averaging_time(*nplc*)

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

Args:

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

get_mode()

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

set_mode(*mode*)

Sets the measurement mode of the module.

Args:

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

get_coupling()

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

set_coupling(*coupling*)

Sets the input coupling of the module.

Args:

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

use_ac_coupling()

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

use_dc_coupling()

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

get_input_configuration()

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

set_input_configuration(*input_configuration*)

Sets the input configuration of the module.

Args:

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

enable_bias_voltage()

Enables the bias voltage applied to the amplifier.

disable_bias_voltage()

Disables the bias voltage applied to the amplifier.

get_bias_voltage_enabled()

Return whether the bias voltage is enabled.

get_bias_voltage()

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

set_bias_voltage(*bias_voltage*)

Sets the bias voltage applied on the amplifier input.

Args:

bias_voltage (float):
The new bias voltage in Volts.

get_filter_state()

Returns whether the hardware filter is engaged.

get_lowpass_corner_frequency()

Returns the low pass filter cutoff 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 cutoff frequency.

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

`get_highpass_rolloff()`

Returns the high pass filter roll-off.

'R6' or 'R12'.

`get_gain_allocation_strategy()`

Returns the gain allocation strategy used for the hardware filter.

'NOISE', or 'RESERVE'.

`set_gain_allocation_strategy(optimization_type)`

Sets the gain allocation strategy used for the hardware filter.

Args:

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

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

Configure the 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_current_range()`

Returns the current range in Amps.

`get_i_range()`

Returns the current range in Amps

Deprecated since version 1.5.4: Use get_current_range instead

`get_current_autorange_status()`

Returns whether auto-ranging is enabled for the module.

`get_i_autorange_status()`

Returns whether autoranging is enabled for the module

Deprecated since version 1.5.4: Use get_current_autorange_status instead

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

Configure current ranging for the module.

Args:

autorange (bool):
True to enable real time range decisions by the module. False for manual ranging.

max_level (float):
The largest current that needs to be measured by the module in Amps.

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

Configure current ranging for the module

Deprecated since version 1.5.4: Use configure_current_range instead

Args:

autorange (bool):
True to enable real time range decisions by the module. False for manual ranging.

max_level (float):
The largest current that needs to be measured by the module in Amps.

`get_voltage_range`()

Returns the voltage range in Volts.

`get_voltage_autorange_status`()

Returns whether auto-ranging is enabled for the module.

`configure_voltage_range`(*autorange*, *max_level=None*)

Configure voltage ranging for the module.

Args:

autorange (bool):
True to enable real time range decisions by the module. False for manual ranging.

max_level (float):
The largest voltage that needs to be measured by the module in Volts.

`get_reference_source`()

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

`set_reference_source`(*reference_source*)

Sets the lock-in reference source

Args:

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

`get_reference_harmonic`()

Returns the lock-in reference harmonic.

`set_reference_harmonic`(*harmonic*)

Sets the lock-in reference harmonic.

Args:

harmonic (int):
The new reference harmonic. 1 is the fundamental frequency, 2 is twice the fundamental frequency, etc.

`get_reference_phase_shift()`

Returns the lock-in reference phase shift in degrees.

`set_reference_phase_shift(`*phase_shift*`)`

Sets the lock-in reference phase shift.

Args:

phase_shift (float):
The new reference phase shift in degrees.

`auto_phase()`

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

`get_lock_in_time_constant()`

Returns the lock-in time constant in seconds.

`set_lock_in_time_constant(`*time_constant*`)`

Sets the lock-in time constant.

Args:

time_constant (float):
The new time constant in seconds.

`get_lock_in_settle_time(`*settle_percent=0.01*`)`

Returns the lock-in settle time in seconds.

Args:

settle_percent (float):
The desired percent signal has settled to in percent. A value of *0.1* is interpreted as 0.1 %.

`get_lock_in_equivalent_noise_bandwidth()`

Returns the equivalent noise bandwidth (ENBW) in Hz.

`get_lock_in_rolloff()`

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

`set_lock_in_rolloff(`*rolloff*`)`

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

Args:

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

`get_lock_in_iir_state()`

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

`set_lock_in_iir_state(`*state*`)`

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

Args:

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

`enable_lock_in_iir()`

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

`disable_lock_in_iir()`

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

`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.

`get_lock_in_fir_cycles()`

Returns the number of FIR cycles.

`set_lock_in_fir_cycles(cycles)`

Sets the number of FIR cycles.

Args:

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

`setup_dc_measurement(nplc=1)`

Set up the module for DC measurement.

Args:

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

`setup_ac_measurement(nplc=1)`

Set up the module for 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)`

Set up the module for lock-in measurement.

Args:

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

time_constant (float):
Time constant in seconds.

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

reference_phase_shift (float):
Lock-in reference phase shift in degrees.

reference_harmonic (int):
Lock-in reference harmonic. 1 is the fundamental frequency, 2 is twice the fundamental frequency, etc.

use_fir (bool):
Enable or disable the PSD output FIR filter.

`zero_relative_baseline()`

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

`set_relative_baseline(`*baseline*`)`

Sets the relative baseline.

`get_relative_baseline()`

Returns the relative baseline.

`get_multiple(`**data_sources*`)`

This function is deprecated. Use fetch_multiple() instead.

`get_dc()`

Returns the DC measurement in module units.

`get_dc_relative()`

Returns the relative DC measurement in module units.

`get_dc_minimum()`

Returns the minimum DC indication in module units.

`get_dc_maximum()`

Returns the maximum DC indication in module units.

`get_rms()`

Returns the RMS measurement in module units.

`get_rms_relative()`

Returns the relative RMS measurement in module units.

`get_rms_minimum()`

Returns the minimum RMS indication in module units.

`get_rms_maximum()`

Returns the maximum RMS indication in module units.

`get_peak_to_peak()`

Returns the peak to peak measurement in module units.

`get_peak_to_peak_minimum()`

Returns the minimum peak to peak indication in module units.

`get_peak_to_peak_maximum()`

Returns the maximum peak to peak indication in module units.

`get_positive_peak()`

Returns the positive peak measurement in module units.

get_positive_peak_minimum()

Returns the minimum positive peak indication in module units.

get_positive_peak_maximum()

Returns the maximum positive peak indication in module units.

get_negative_peak()

Returns the negative peak measurement in module units.

get_negative_peak_minimum()

Returns the minimum negative peak indication in module units.

get_negative_peak_maximum()

Returns the maximum negative peak indication in module units.

get_lock_in_x()

Returns the present X measurement from the lock-in.

get_lock_in_x_minimum()

Returns the minimum X indication from the lock in.

get_lock_in_x_maximum()

Returns the maximum X indication from the lock in.

get_lock_in_y()

Returns the present Y measurement from the lock-in.

get_lock_in_y_minimum()

Returns the minimum Y indication from the lock in.

get_lock_in_y_maximum()

Returns the maximum Y indication from the lock in.

get_lock_in_r()

Returns the present magnitude measurement from the lock-in.

get_lock_in_r_minimum()

Returns the minimum magnitude indication from the lock in.

get_lock_in_r_maximum()

Returns the maximum magnitude indication from the lock in.

get_lock_in_theta()

Returns the present angle measurement from the lock-in.

get_lock_in_theta_minimum()

Returns the minimum angle indication from the lock in.

get_lock_in_theta_maximum()

Returns the maximum angle indication from the lock in.

get_lock_in_frequency()

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

get_pll_lock_status()

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

`get_present_questionable_status()`

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

`get_questionable_events()`

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

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

`get_questionable_event_enable_mask()`

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

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

`set_questionable_event_enable_mask(`*register_mask*`)`

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

`get_present_operation_status()`

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

`get_overload_status()`

Returns whether the module is presently in overload or not

`get_settling_status()`

Returns whether the module is presently settling or not

`get_unlocked_status()`

Returns whether the module is presently unlocked or not

`get_operation_events()`

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

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

`get_operation_event_enable_mask()`

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

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

`set_operation_event_enable_mask(`*register_mask*`)`

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

`get_identify_state()`

Returns the identification state for the given pod.

`set_identify_state(`*state*`)`

Sets the identification state for the given pod.

Args:

state (bool):
The desired state for the LED, 1 for identify, 0 for normal state.

`get_dark_mode_state()`

Returns the dark mode state for the given pod.

`set_dark_mode_state(state)`

Configures the dark mode state for the given pod.

Args:

state (bool):
The desired operation for the LED, 1 for normal mode, 0 for dark mode.

`get_frequency_range_threshold()`

Returns the frequency range threshold for the module.

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

`set_frequency_range_threshold(threshold)`

Sets the frequency range threshold for the specified module.

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

Args:

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

`get_digital_high_pass_filter_state()`

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

`set_digital_high_pass_filter_state(state)`

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

Args:

state (bool):
The desired state fot he digital lock-in high pass filter. 1 for enabled, 0 for disabled.

`get_resistance()`

Returns the present resistance measurement in Ohms. A valid source must be configured.

Returns:
float: Present resistance measurement in Ohms.

`set_resistance_source(source_module)`

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

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

`get_resistance_source()`

Returns the present source module being used to calculate resistance.

Returns:
SourceModule: The channel used for calculating resistance.

`set_resistance_excitation_type`(*excitation_type*)

Sets the present resistance excitation type of the specified module.

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

Args:
excitation_type (ResistanceExcitationType): The desired resistance excitation type.

`get_resistance_excitation_type`()

Returns the present resistance excitation type of the specified module.

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

Returns:
ResistanceExcitationType: The resistance excitation type.

`set_resistance_mode`(*resistance_mode*)

Sets the resistance optimization mode of the specified module.

Args:
resistance_mode (ResistanceMode): The desired resistance optimization mode.

`get_resistance_mode`()

Returns the preset resistance optimization mode of the specified module.

Returns:
ResistanceMode: The present resistance optimization mode. "NOISe" or "POWer".

`set_resistance_range`(*resistance_range*)

Sets the resistance range of the specified module.

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

`get_resistance_range`()

Returns the present resistance range of the specified module.

Returns:
float: The resistance range in Ohms.

`set_resistance_optimization_state`(*optimization_state*)

Sets the state of resistance optimization on the specified module

Args:

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

`get_resistance_optimization_state`()

Returns the present state of optimization on the specified module.

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

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

`set_resistance_observation_time_state`(*state*)

Sets the state of the observation time on the specified module.

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

`get_resistance_observation_time_state`()

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

Returns:
bool: The state of resistance observation time.

`set_resistance_observation_time_requested`(*requested_time*)

Sets the requested observation time on the specified module.

Args:
requested_time (float): The requested observation time.

`get_resistance_observation_time_requested`()

Gets the present requested observation time on the specified module.

Returns:
float: The requested observation time.

`get_resistance_observation_time_actual`()

Gets the present actual observation time for the resistance calculation.

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

Returns:
float: The actual observation time.

`get_resistance_observation_time_enbw`()

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

Returns:
float: The calculated equivalent noise bandwidth.

`reset_settings`()

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

`unload`()

Unloads the specified module.

`get_load_state`()

Returns the loaded state for the specified module.

`fetch_multiple`(**data_sources*)

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

Args:

data_sources (SSMSystemDataSourceMnemonic or str):
Variable length list of data sources.

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

read_multiple(**data_sources*)

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

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

Args:

data_sources (SSMSystemReadDataSourceMnemonic or str):
Variable length list of data sources.

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

get_self_cal_datetime()

Returns the self calibration date and time for the specified module.

get_self_cal_temperature()

Returns the self calibration temperature for the specified module.

Settings Profiles methods

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

Class for interaction with settings profiles.

get_summary(*name*)

Returns a list containing a profile's description and module models.

Args:

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

create(*name*, *description=''*)

Create a new profile using the present instrument configuration.

Args:

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

description (str):
Optional description of the profile.

get_list()

Returns a list of the saved profile names.

get_description(*name*)

Returns a profile's description.

Args:

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

set_description(*name*, *description*)

Sets a profile's description. Any existing description will be overwritten.

Args:

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

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

`get_json`(*name*)

Returns a JSON object of a given profile.

Args:

name (str):
Name of the profile.

`rename`(*name*, *new_name*)

Rename a profile. New name must be unique.

Args:

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

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

`update`(*name*)

Update a profile with the present instrument configuration.

Args:

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

`get_valid_for_restore`(*name*)

Returns if a profile is valid to restore.

Args:

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

`restore`(*name*)

Restore a profile.

Args:

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

`delete`(*name*)

Delete a profile.

Args:

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

`delete_all`()

Delete all profiles.

Instrument registers

This page outlines the registers used to interact with various settings and methods of the M81.

class lakeshore.ssm_system.**SSMSystemOperationRegister**(*s1_summary*, *s2_summary*, *s3_summary*, *m1_summary*, *m2_summary*, *m3_summary*, *data_stream_in_progress*)

Class object representing the operation status register.

class lakeshore.ssm_system.**SSMSystemQuestionableRegister**(*s1_summary*, *s2_summary*, *s3_summary*, *m1_summary*, *m2_summary*, *m3_summary*, *critical_startup_error*, *critical_runtime_error*, *heartbeat*, *calibration*, *data_stream_overflow*)

Class object representing the questionable status register.

class lakeshore.ssm_base_module.**SSMSystemModuleQuestionableRegister**(*read_error=False*, *unrecognized_pod_error=False*, *port_direction_error=False*, *factory_calibration_failure=False*, *self_calibration_failure=False*)

Class object representing the questionable status register of a module.

class lakeshore.ssm_source_module.**SSMSystemSourceModuleOperationRegister**(*v_limit*, *i_limit*, *sweeping*)

Class object representing the operation status register of a source module.

class lakeshore.ssm_measure_module.**SSMSystemMeasureModuleOperationRegister**(*overload*, *settling*, *unlocked*)

Class object representing the operation status register of a measure module.

Enumeration Objects

This section describes the Enum type objects that have been created for the M81 SSM.

class lakeshore.ssm_system_enums.SSMSystemEnums.**DataSourceMnemonic**(*value*)

Enumeration of M81 data source mnemonics.

RELATIVE_TIME = 'RTIMe'

SOURCE_AMPLITUDE = 'SAMPlitude'

SOURCE_OFFSET = 'SOFFset'

SOURCE_FREQUENCY = 'SFRequency'

SOURCE_RANGE = 'SRANge'

SOURCE_VOLTAGE_LIMIT = 'SVLimit'

SOURCE_CURRENT_LIMIT = 'SILimit'

SOURCE_IS_SWEEPING = 'SSWeeping'

```
    MEASURE_DC = 'MDC'

    MEASURE_RMS = 'MRMS'

    MEASURE_POSITIVE_PEAK = 'MPPeak'

    MEASURE_NEGATIVE_PEAK = 'MNPeak'

    MEASURE_PEAK_TO_PEAK = 'MPTPeak'

    MEASURE_X = 'MX'

    MEASURE_Y = 'MY'

    MEASURE_R = 'MR'

    MEASURE_THETA = 'MTHeta'

    MEASURE_RANGE = 'MRANge'

    MEASURE_OVERLOAD = 'MOVerload'

    MEASURE_SETTLING = 'MSETtling'

    MEASURE_UNLOCK = 'MUNLock'

    MEASURE_REFERENCE_FREQUENCY = 'MRFRequency'

    GENERAL_PURPOSE_INPUT_STATES = 'GPIStates'

    GENERAL_PURPOSE_OUTPUT_STATES = 'GPOStates'
```

2.4.4 Temperature Controllers

Model 335 Cryogenic Temperature Controller

The Model 335 measures and controls cryogenic temperature environments.

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

Example Scripts

Setting a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
```

(continues on next page)

(continued from previous page)

```
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↪VOLTS_PER_KELVIN, 300.0,
                                      myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set␣
↪to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.␣
↪The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,␣
↪30, "SN123", (276, 10),
                                               (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can␣
↪then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that␣
↪curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Recording data with the Model 335

```
from lakeshore import Model335, Model335InputSensorSettings

# Connect to the first available Model 335 temperature controller over USB using a baud␣
↪rate of 57600
my_model_335 = Model335(57600)

# Create a new instance of the input sensor settings class
sensor_settings = Model335InputSensorSettings(my_model_335.InputSensorType.DIODE, True,␣
↪False,
                                              my_model_335.InputSensorUnits.KELVIN,
                                              my_model_335.DiodeRange.TWO_POINT_FIVE_
↪VOLTS)

# Apply these settings to input A of the instrument
my_model_335.set_input_sensor("A", sensor_settings)
```

(continues on next page)

(continued from previous page)

```
# Set diode excitation current on channel A to 10uA
my_model_335.set_diode_excitation_current("A", my_model_335.DiodeCurrent.TEN_MICROAMPS)

# Collect instrument data
heater_output_1 = my_model_335.get_heater_output(1)
heater_output_2 = my_model_335.get_heater_output(2)
temperature_reading = my_model_335.get_all_kelvin_reading()

# Open a csv file to write
file = open("335_record_data.csv", "w")

# Write the data to the file
file.write("Data retrieved from the Lake Shore Model 335\n")
file.write("Temperature Reading A: " + str(temperature_reading[0]) + "\n")
file.write("Temperature Reading B: " + str(temperature_reading[1]) + "\n")
file.write("Heater Output 1: " + str(heater_output_1) + "\n")
file.write("Heater Output 2: " + str(heater_output_2) + "\n")
file.close()
```

Setting up autotune on the Model 335

```
from lakeshore import Model335
from time import sleep

# Connect to the first available Model 335 temperature controller over USB using a baud␣
↪rate of 57600
my_model_335 = Model335(57600)

# It is assumed that the instrument is configured properly with a control input sensor␣
↪curve
# and heater output, capable of closed loop control

# Configure the display mode
my_model_335.set_display_setup(my_model_335.DisplaySetup.TWO_INPUT_A)

# Configure heater output 1 using the HeaterSetup class and set_heater_setup method
my_model_335.set_heater_setup_one(my_model_335.HeaterResistance.HEATER_50_OHM, 1.0, my_
↪model_335.HeaterOutputDisplay.POWER)

# Configure heater output 1 to a setpoint of 310 kelvin (units correspond to the␣
↪configured output units)
set_point = 325
my_model_335.set_control_setpoint(1, set_point)

# Turn on the heater by setting the range
my_model_335.set_heater_range(1, my_model_335.HeaterRange.HIGH)

# Check to see if there are any heater related errors
heater_error = my_model_335.get_heater_status(1)
if heater_error is not my_model_335.HeaterError.NO_ERROR:
```

(continues on next page)

(continued from previous page)

```
    raise Exception(heater_error.name)

# Allow the heater some time to turn on and start maintaining a setpoint
sleep(10)

# Ensure that the temperature is within 5 degrees kelvin of the setpoint
kelvin_reading = my_model_335.get_kelvin_reading(1)
if (kelvin_reading < (set_point - 5)) or (kelvin_reading > (set_point + 5)):
    raise Exception("Temperature reading is not within 5k of the setpoint")

# Initiate autotune in PI mode, initial conditions will not be met if the system is not
# maintaining a temperature within 5 K of the setpoint
my_model_335.set_autotune(1, my_model_335.AutotuneMode.P_I)

# Poll the instrument until the autotune process completes
autotune_status = my_model_335.get_tuning_control_status()
while autotune_status["active_tuning_enable"] and not autotune_status["tuning_error"]:
    autotune_status = my_model_335.get_tuning_control_status()
    # Print the status to the console every 5 seconds
    print("Active tuning: " + str(autotune_status["active_tuning_enable"]))
    print("Stage status: " + str(autotune_status["stage_status"]) + "/10")
    sleep(5)

if autotune_status["tuning_error"]:
    raise Exception("An error occurred while running autotune")
```

Instrument class methods

class lakeshore.model_335.**Model335**(*baud_rate*, *serial_number=None*, *com_port=None*, *timeout=2.0*, *ip_address=None*, *tcp_port=None*, ***kwargs*)

A class object representing the Lake Shore Model 335 cryogenic temperature controller.

get_analog_output_percentage(*output*)

Returns the output percentage of the analog voltage output.

Args:

output (int):
Specifies which analog voltage output to query.

Returns:

(float):
Analog voltage heater output percentage.

set_autotune(*output*, *mode*)

Initiates auto-tuning of the heater control loop.

Args:

output (int):
Specifies the output associated with the loop to be Auto-tuned.

mode (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.

Returns:

(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()

Returns the temperature of the ceramic thermocouple block from the room temperature compensation calculation.

Returns:

(float):
Temperature of the ceramic thermocouple block (kelvin).

set_soft_cal_curve_dt_470(*curve_number*, *serial_number*, *calibration_point_1=(4.2, 1.62622)*, *calibration_point_2=(77.35, 1.02032)*, *calibration_point_3=(305, 0.50691)*)

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured DT-470 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):
The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):
Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple):
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*)

Sets the excitation current of a specific channel.

The 10 uA excitation current is the only calibrated excitation current, and is used in almost all applications. The Model 336 will default the 10 uA current setting any time the input sensor type is changed.

Args:

channel (str):
Specifies which sensor input to configure: “A” - “D”.

excitation_current (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".

Returns:

(IntEnum):
A member of the instrument's DiodeCurrent IntEnum class. Diode excitation current.

`get_tuning_control_status`()

Returns dictionary of tuning control status values.

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

Returns:

(dict):

{"active_tuning_enable": bool, "output": int, "tuning_error": bool, "stage_status": int}
active_tuning_enable: False = no active tuning, True = active tuning. output: Heater output of the control loop being tuned. tuning_error: False = no tuning error, True = tuning error. stage_status: Specifies the current stage in the Autotune process. If tuning error occurred, stage status represents stage that failed.

`set_filter`(*input_channel*, *filter_enable*, *data_points*, *reset_threshold*)

Configures the input_channel filter parameter.

Args:

input_channel (int or str):
Specifies which input channel to configure.

filter_enable (bool):
Specified whether the filtering function is enabled or not.

data_points (int):
Specifies how many points the filter function uses: 2 - 64.

reset_threshold (int):
Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets. Options are: 1% - 10%.

`get_filter`(*input_channel*)

Returns the input_channel filter configuration.

Args:

input_channel (int or str):
Specifies which input channel to configure.

Returns:

(dict)

{"filter_enable": bool, "data_points": int, "reset_threshold": int}
filter_enable: Specified whether the filtering function is enabled or not. data_points: Specifies how many points the filter function uses. reset_threshold: Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets (1% - 10%).

`set_monitor_output_heater`(*channel*, *high_value*, *low_value*, *units=MonitorOutUnits.KELVIN*, *polarity=Polarity.UNIPOLAR*)

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

Args:

channel (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% output if 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.

Returns:

(dict):
{"channel": Model335InputSensor, "units": Model335MonitorOutUnits, "high_value": float, "low_value": float, "polarity": Model335Polarity}

See set_monitor_output_heater method arguments

`get_celsius_reading`(*channel*)

Returns the temperature value in Celsius of either channel.

Args:

channel (str):
Selects the sensor input to query ("A" or "B"),

`set_display_setup`(*mode*)

Sets the display mode.

Args:

mode (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 whether 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. {"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): {"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):
{"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.

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, True = on).

compensation (bool):
Specifies input compensation. (False = off, True = on).

units (Model335InputSensorUnits):
Specifies the preferred units parameter for sensor readings and for the control set-point.

input_range (IntEnum)
Specifies input range if autorange_enable is false. See IntEnum classes: Model335DiodeRange, Model335RTDRange, and 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 set-point boundary of this zone in kelvin.

proportional (float):
Specifies the proportional gain for this zone (0.1 to 1000).

integral (float):
Specifies the integral gain for this zone (0.1 to 1000).

derivative (float):
Specifies the derivative gain for this zone (0 to 200 %).

manual_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.

class lakeshore.model_335.**Model335Enums**

Class containing the enums relevant to the Model 335.

class **InputSensor**(*value*)

Enumeration when "NONE" is an option for sensor input.

NONE = 0

CHANNEL_A = 1

CHANNEL_B = 2

class **MonitorOutUnits**(*value*)

Units associated with a sensor channel.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

class InputSensorType(*value*)

Sensor type enumeration.

DISABLED = 0

DIODE = 1

PLATINUM_RTD = 2

NTC_RTD = 3

THERMOCOUPLE = 4

class DiodeRange(*value*)

Diode voltage range enumeration.

TWO_POINT_FIVE_VOLTS = 0

TEN_VOLTS = 1

class RTDRange(*value*)

RTD resistance range enumeration.

TEN_OHM = 0

THIRTY_OHM = 1

HUNDRED_OHM = 2

THREE_HUNDRED_OHM = 3

ONE_THOUSAND_OHM = 4

THREE_THOUSAND_OHM = 5

TEN_THOUSAND_OHM = 6

THIRTY_THOUSAND_OHM = 7

ONE_HUNDRED_THOUSAND_OHM = 8

class ThermocoupleRange(*value*)

Thermocouple range enumeration.

FIFTY_MILLIVOLT = 0

class HeaterOutType(*value*)

Heater output 2 enumeration.

CURRENT = 0

VOLTAGE = 1

class HeaterOutputDisplay(*value*)

Heater output display units enumeration.

CURRENT = 1

POWER = 2

class HeaterOutputMode(*value*)

Control loop enumeration.

OFF = 0

CLOSED_LOOP = 1

ZONE = 2

OPEN_LOOP = 3

MONITOR_OUT = 4

WARMUP_SUPPLY = 5

class WarmupControl(*value*)

Heater output 2 voltage mode warmup enumerations.

AUTO_OFF = 0

CONTINUOUS = 1

class HeaterRange(*value*)

Control loop heater range enumeration.

OFF = 0

LOW = 1

MEDIUM = 2

HIGH = 3

class DisplaySetup(*value*)

Panel display setup enumeration.

INPUT_A = 0

INPUT_A_MAX_MIN = 1

TWO_INPUT_A = 2

INPUT_B = 3

INPUT_B_MAX_MIN = 4

TWO_INPUT_B = 5

CUSTOM = 6

TWO_LOOP = 7

class HeaterVoltageRange(*value*)

Voltage mode heater enumerations.

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class InputChannel(*value*)

Panel display information enumeration.

NONE = 0

INPUT_A = 1

INPUT_B = 2

SETPOINT_1 = 3

SETPOINT_2 = 4

OUTPUT_1 = 5

OUTPUT_2 = 6

class DisplayFieldUnits(*value*)

Panel display units enumeration.

KELVIN = 1

CELSIUS = 2

SENSOR_UNITS = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

SENSOR_NAME = 6

Status register classes

This section describes the register objects. Each bit in the register is represented as a member of the register's class

class lakeshore.model_335.**Model335StatusByteRegister**(*message_available_summary_bit*, *event_status_summary_bit*, *service_request*, *operation_summary_bit*)

Class object representing the status byte register LSB to MSB.

class lakeshore.model_335.**Model335ServiceRequestEnable**(*message_available_summary_bit*, *event_status_summary_bit*, *operation_summary_bit*)

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

lakeshore.model_335.**Model335StandardEventRegister**

alias of *StandardEventRegister*

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

Class object representing the standard event register.

lakeshore.model_335.**Model335OperationEvent**

alias of *OperationEvent*

class lakeshore.temperature_controllers.**OperationEvent**(*alarm*, *sensor_overload*, *loop_2_ramp_done*, *loop_1_ramp_done*, *new_sensor_reading*, *autotune_process_completed*, *calibration_error*, *processor_communication_error*)

Class object representing the status byte register LSB to MSB.

class lakeshore.model_335.**Model335InputReadingStatus**(*invalid_reading*, *temp_underrange*, *temp_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, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↪VOLTS_PER_KELVIN, 300.0,
                                      myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set␣
↪to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.␣
↪The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,␣
↪30, "SN123", (276, 10),
```

(continues on next page)

(continued from previous page)

```
                                                (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can␣
↪then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that␣
↪curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Setting up heater outputs on the Model 336

```
from lakeshore import Model336

# Connect to the first available Model 336 temperature controller over USB with a baud␣
↪rate of 57600
my_model_336 = Model336()

# Set the control loop values for outputs 1 and 3
my_model_336.set_heater_pid(1, 40, 27, 0)
my_model_336.set_heater_pid(3, 35, 20, 0)

# Configure heater output 1 with a 50 ohm load, 0.75 amp max current, and screen display␣
↪mode to power
my_model_336.set_heater_setup(1, my_model_336.HeaterResistance.HEATER_50_OHM, 0.75, my_
↪model_336.HeaterOutputUnits.POWER)

# Configure analog heater output 3 to monitor sensor channel A, a high and low value of␣
↪3.65 and 1.02 kelvin
# respectively as a unipolar output
my_model_336.set_monitor_output_heater(3, my_model_336.InputChannel.CHANNEL_A, my_model_
↪336.InputSensorUnits.KELVIN, 3.65, 1.02,
                                       my_model_336.Polarity.UNIPOLAR)

# Set closed loop output mode for heater 1
my_model_336.set_heater_output_mode(1, my_model_336.HeaterOutputMode.CLOSED_LOOP, my_
↪model_336.InputChannel.CHANNEL_A)

# Set closed loop output mode for heater 3
my_model_336.set_heater_output_mode(3, my_model_336.HeaterOutputMode.CLOSED_LOOP, my_
↪model_336.InputChannel.CHANNEL_B)

# Set a control setpoint for outputs 1 and 3 to 1.5 kelvin
my_model_336.set_control_setpoint(1, 1.5)
```

(continues on next page)

(continued from previous page)

```
my_model_336.set_control_setpoint(3, 2.5)

# Turn the heaters on by setting the heater range
my_model_336.set_heater_range(1, my_model_336.HeaterRange.MEDIUM)
my_model_336.set_heater_range(3, my_model_336.HeaterVoltageRange.VOLTAGE_ON)

# Obtain the output percentage of output 1 and print it to the console
heater_one_output = my_model_336.get_heater_output(1)
print("Output 1: " + str(heater_one_output))

# Obtain the output percentage of output 3 and print it to the console
heater_three_output = my_model_336.get_analog_output_percentage(3)
print("Output 3: " + str(heater_three_output))
```

Instrument class methods

class lakeshore.model_336.**Model336**(*serial_number=None*, *com_port=None*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing the Lake Shore Model 336 cryogenic temperature controller.

status_byte_register

alias of *Model336StatusByteRegister*

service_request_enable

alias of *Model336ServiceRequestEnable*

get_analog_output_percentage(*output*)

Returns the output percentage of the analog voltage output.

Args:

output (int):
Specifies which analog voltage output to query.

Returns:

(float):
Analog voltage heater output percentage.

set_autotune(*output*, *mode*)

Initiates auto-tuning of the heater control loop.

Args:

output (int):
Specifies the output associated with the loop to be Auto-tuned.

mode (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()`

Returns the temperature of the ceramic thermocouple block from the room temperature compensation calculation.

Returns:

(float):
Temperature of the ceramic thermocouple block (kelvin).

`set_soft_cal_curve_dt_470(curve_number, serial_number, calibration_point_1=(4.2, 1.62622), calibration_point_2=(77.35, 1.02032), calibration_point_3=(305, 0.50691))`

Creates a SoftCal curve from any 1-3 temperature/sensor points using the preconfigured DT-470 curve.

When a calibration point other than one or more the default value(s) is entered a SoftCal curve is generated.

Args:

curve_number (int):
The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):
Specifies the curve serial number. Limited to 10 characters.

calibration_point_1 (tuple):
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.

Returns:

(dict)

{"filter_enable": bool, "data_points": int, "reset_threshold": int}
filter_enable: Specified whether the filtering function is enabled or not. data_points: Specifies how many points the filter function uses. reset_threshold: Specifies what percent of full scale reading limits the filtering function. When a raw reading differs from a filtered value by more than this threshold, the filter averaging resets (1% - 10%).

`set_network_settings`(*dhcp_enable*, *auto_ip_enable*, *ip_address*, *sub_mask*, *gateway*, *primary_dns*, *secondary_dns*, *pref_host*, *pref_domain*, *description*)

Network class constructor.

Args:

dhcp_enable (bool):
Enable or disable DHCP.

auto_ip_enable (bool):
Enable or disable dynamically configured link-local addressing (Auto IP).

ip_address (str):
IP address for static configuration.

sub_mask (str):
Subnet mask for static configuration.

gateway (str):
Gateway address for static configuration.

primary_dns (str):
Primary DNS address for static configuration.

secondary_dns (str):
Secondary DNS address for static configuration.

pref_host (str):
Preferred Hostname (15 character maximum).

pref_domain (str):
Preferred Domain name (64 character maximum).

description (str):
Instrument description (32 character maximum).

`get_network_settings`()

Method to retrieve the IP settings.

Returns:

(dict):
See set_network_settings arguments.

`get_network_configuration`()

Method to return the configured ethernet parameters.

Return:

(dict):

{"lan_status": LanStatus, "ip_address": str, "sub_mask": str, "gateway": str, "primary_dns": str,
"secondary_dns": str, "hostname" str, "domain": str, "mac_address": str} lan_status: Current status of the ethernet connection. Member of the instrument's LanStatus IntEnum class. ip_address: Configured IP address. sub_mask: Configured subnet mask. gateway: Configured gateway address. primary_dns: Configured primary DNS address. secondary_dns: Configured secondary DNS address. hostname: Assigned hostname. domain: Assigned domain. mac_address: Module MAC address.

`set_website_login`(*username*, *password*)

Sets the username and password for the web interface.

Args:

username (str):
15 character string representing the website username.

password (str):
15 character string representing the website password.

`get_website_login`()

Method to return the username and password for the web interface.

Returns:

website_login (dict): A dictionary containing 15 character string type items.
{"username": str, "password": str}

`get_celsius_reading`(*channel*)

Returns the temperature in Celsius of any channel.

Args:

channel (str):
"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.

Returns:

(IntEnum):
Member of instrument's Interface IntEnum class.

`get_tuning_control_status`()

Returns dictionary of tuning control status values.

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

Returns:

(dict):

{"active_tuning_enable": bool, "output": int, "tuning_error": bool, "stage_status": int}
active_tuning_enable: False = no active tuning, True = active tuning. output: Heater output of the control loop being tuned. tuning_error: False = no tuning error, True = tuning error. stage_status: Specifies the current stage in the Autotune process. If tuning error occurred, stage status represents stage that failed.

`set_diode_excitation_current`(*channel*, *excitation_current*)

Sets the excitation current of a specific channel.

The 10 uA excitation current is the only calibrated excitation current, and is used in almost all applications. The Model 336 will default the 10 uA current setting any time the input sensor type is changed.

Args:

channel (str):
Specifies which sensor input to configure: "A" - "D".

excitation_current (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".

Returns:

(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 (InputChannel):
Specifies which sensor input to monitor. A member of the InputChannel IntEnum class.

units (self.InputSensorUnits):
Specifies the units on which to base the output voltage. A member of the self.InputSensorUnits IntEnum class.

high_value (float):
Represents the data at which the Monitor Out reaches +100% output. Entered in the units designated by the <units> argument.

low_value (float):
Represents the data at which the analog output reaches -100% output if bipolar, or 0% output if unipolar. Entered in the units designated by the <units> argument.

polarity (self.Polarity):
Specifies whether the output voltage is unipolar or bipolar. Member of the self.Polarity IntEnum class.

`get_monitor_output_heater`(*output*)

Used to obtain all monitor out parameters for a specific output.

Args:

output (int):
Voltage-controlled output to configure (3 or 4).

Returns:

(dict):
See set_monitor_output_heater arguments

`set_display_setup`(*mode*, *num_fields=''*, *displayed_output=''*)

Sets the display mode.

Args:

mode (self.DisplaySetupMode):
Member of self.DisplaySetupMode IntEnum class Specifies display mode for default and 3062 options

num_fields (IntEnum)
When mode is set to custom, specifies the number of fields (Member of self.DisplayFields). When mode is set to all inputs, specifies size of readings (Member of self.DisplayFieldsSize).

displayed_output (int):
Configures the bottom half of the custom display screen. Only required if mode is set to CUSTOM. Output: (1 - 4)

`get_display_setup`()

Returns the display mode.

Returns:

(dict):
See set_display_setup method arguments. Keys: “mode”, “num_fields”, “displayed_output”

`set_heater_setup`(*output*, *heater_resistance*, *max_current*, *heater_output*)

Method to configure the heaters.

Args:

output (int):
Specifies which heater output to configure (1 or 2).

heater_resistance (self.HeaterResistance):
Member of self.HeaterResistance IntEnum class.

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

heater_output (self.HeaterOutputUnits):
Specifies whether the heater output displays in current or power. Member of self.HeaterOutputUnits IntEnum class.

`get_heater_setup`(*heater_output*)

Returns the heater configuration status.

Args:

heater_output (int):
Specifies which heater output to configure (1 or 2)

Returns:

(dict):
See set_heater_setup arguments Keys: heater_resistance, max_current, output_display_mode.

`set_input_sensor`(*channel*, *sensor_parameters*)

Sets the sensor type and associated parameters.

Args:

channel (str):
Specifies input to configure ("A" - "D"): 3062 option ("D1" - "D5")

sensor_parameters (Model336InputSensorSettings):
See Model336InputSensorSettings class.

`get_input_sensor`(*channel*)

Returns the sensor type and associated parameters.

Args:

channel (str):
Specifies sensor input to configure ("A" or "B")

Returns:

(Model336InputSensorSettings):
See Model336InputSensorSettings class.

`get_all_kelvin_reading`()

Returns the temperature value in kelvin of all channels.

Returns:

(list: float):
[channel_A, channel_B, channel_C, channel_D]

`set_heater_output_mode`(*output*, *mode*, *channel*, *powerup_enable=False*)

Configures the heater output mode.

Args:

output (int):
Specifies which output to configure (1 - 4)

mode (self.HeaterOutputMode):
Member of self.HeaterOutputMode IntEnum class. Specifies the control mode.

channel (InputChannel):
InputChannel IntEnum class. Specifies which input to use for control.

powerup_enable (bool):
Specifies whether the output remains on (True) or shuts off after power cycle (False).

`get_heater_output_mode`(*output*)

Returns the heater output mode for a given output and whether powerup is enabled.

Args:

output (int):
Specifies which output to retrieve (1 - 4).

Returns:

(dict):
See set_heater_output_mode method arguments. Keys: mode, channel, powerup_enable.

`set_heater_range`(*output*, *heater_range*)

Sets the heater range for a particular output.

The range setting has no effect if an output is in the Off mode, and does not apply to an output in Monitor Out mode. An output in Monitor Out mode is always on.

Args:

output (int):
Specifies which output to configure (1 - 4).

heater_range (IntEnum):
For Outputs 1 and 2: Member of self.HeaterRange IntEnum class. For Outputs 3 and 4: self.HeaterVoltageRange IntEnum class.

`get_heater_range`(*output*)

Returns the heater range for a particular output.

Args:

output (int):
Specifies which output to query (1 or 2).

Returns:

(IntEnum):
For Outputs 1 and 2: Member of self.HeaterRange IntEnum class. For Outputs 3 and 4: Member of self.HeaterVoltageRange IntEnum class.

`all_heaters_off`()

Recreates the front panel safety feature of shutting off all heaters.

`get_input_reading_status`(*channel*)

Reruns the state of the input status flag bits.

Args:

channel (str):
Specifies which channel to query ("A" - "D"). Use "D1" - "D5" for 3062 option.

Returns:

(Model336InputReadingStatus):
Boolean representation of each bit in the input status flag register.

`get_all_sensor_reading`()

Returns the sensor unit reading of all channels.

Returns:

(list: float):
[channel_A, channel_B, channel_C, channel_D]

`set_warmup_supply_parameter`(*output*, *control*, *percentage*)

Warmup mode applies only to voltage heater outputs 3 and 4.

The Output mode and Control Input parameters must be configured using the set_monitor_out_parameters() method.

Args:

output (int):
Specifies which output to configure (3 or 4).

control (self.ControlTypes):
Member of the self.ControlTypes IntEnum class.

percentage (float):
Specifies the percentage of full scale (10 V) Monitor Out voltage to apply to turn on the external power supply. (A value of 50.5 translates to a 50.5 percent output voltage).

`get_warmup_supply_parameter`(*output*)

Returns the warmup supply configuration for a particular output.

Args:

output (int):
Specifies which analog voltage heater output to retrieve (3 or 4).

Returns:

(dict):
See set_warmup_supply_parameter method arguments

`set_control_loop_zone_table`(*output*, *zone*, *control_loop_zone*)

Configures the output zone parameters.

Args:

output (int):
Specifies which analog voltage heater output to configure (1 or 2).

zone (int):
Specifies which zone in the table to configure (1 to 10).

control_loop_zone (Model336ControlLoopZoneSettings):
See Model336ControlLoopZoneSettings class.

`get_control_loop_zone_table`(*output*, *zone*)

Returns a list of zone control parameters for a selected output and zone.

Args:

output (int):
Specifies which heater output to query (1 or 2).

zone (int):
Specifies which zone in the table to query (1 to 10).

Returns:

(Model336ControlLoopZoneSettings):
See Model336ControlLoopZoneSettings class.

Settings classes

This section outlines the classes used to interact with methods which return or accept an argument of a class object, specific to the Lake Shore model 336.

class lakeshore.model_336.**Model336InputSensorSettings**(*sensor_type*, *autorange_enable*, *compensation*, *units*, *input_range=None*)

Class object used in the get/set_input_sensor methods.

__init__(*sensor_type*, *autorange_enable*, *compensation*, *units*, *input_range=None*)

Constructor for the InputSensorSettings class.

Args:

sensor_type (self.InputSensorType):
Specifies input sensor type

autorange_enable (bool):
Specifies auto-ranging (False = off, True = on)

compensation (bool):
Specifies input compensation (False = off, True = on)

units (self.InputSensorUnits):
Specifies the preferred units parameter for sensor readings and for the control set-point.

input_range (IntEnum)
Specifies input range if autorange_enable is false. See IntEnum classes: self.DiodeRange, self.RTDRange, andself.ThermocoupleRange.

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

Control loop configuration for a particular heater output and zone.

__init__(*upper_bound*, *proportional*, *integral*, *derivative*, *manual_out_value*, *heater_range*, *channel*, *rate*)

Constructor.

Args:

upper_bound (float):
Specifies the upper set-point boundary of this zone in kelvin.

proportional (float):
Specifies the proportional gain for this zone (0.1 to 1000).

integral (float):
Specifies the integral gain for this zone (0.1 to 1000).

derivative (float):
Specifies the derivative gain for this zone (0 to 200 %).

manual_out_value (float):
Specifies the manual output for this zone (0 to 100 %).

heater_range (self.HeaterRange):
Specifies the heater range for this zone. See self.HeaterRange IntEnum class.

channel (InputChannel):
See InputChannel IntEnum class. Passing the NONE member will use the previously assigned sensor.

rate (float):
Specifies the ramp rate for this zone (0 - 100 K/min).

lakeshore.model_336.**Model336AlarmSettings**

alias of *AlarmSettings*

class lakeshore.temperature_controllers.**AlarmSettings**(*high_value*, *low_value*, *deadband*, *latch_enable*, *audible=None*, *visible=None*, *alarm_enable=None*)

Class used to disable or configure an alarm in conjunction with the set/get_alarm_parameters() method.

__init__(*high_value*, *low_value*, *deadband*, *latch_enable*, *audible=None*, *visible=None*, *alarm_enable=None*)

Constructor for AlarmSettings class.

Args:

high_value (float):
Sets the value the source is checked against to activate the high alarm.

low_value (float):
Sets the value the source is checked against to activate low alarm.

deadband (float):
Sets the value that the source must change outside an alarm. condition to deactivate an unlatched alarm.

latch_enable (bool):
Specifies a latched alarm. False = off, True = on

audible (bool):
Specifies if the internal speaker will beep when an alarm condition occurs. False = off, True = on

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

lakeshore.model_336.**Model336CurveHeader**

alias of *CurveHeader*

class lakeshore.temperature_controllers.**CurveHeader**(*curve_name*, *serial_number*, *curve_data_format*, *temperature_limit*, *coefficient*)

A class to configure the temperature sensor curve header parameters.

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

Constructor for CurveHeader class.

Args:

curve_name (str):
Specifies curve name (limit of 15 characters).

serial_number (str):
Specifies curve serial number (limit of 10 characters).

curve_data_format (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.**Model336Enums**

Class containing the enums relevant to the Model 336.

class InputChannel(*value*)

Enumeration where "NONE" is an option for sensor input.

NONE = 0

CHANNEL_A = 1

CHANNEL_B = 2

CHANNEL_C = 3

CHANNEL_D = 4

CHANNEL_D2 = 5

CHANNEL_D3 = 6

CHANNEL_D4 = 7

CHANNEL_D5 = 8

class DisplaySetupMode(*value*)

Front panel display setup enum.

INPUT_A = 0

INPUT_B = 1

INPUT_C = 2

INPUT_D = 3

CUSTOM = 4

FOUR_LOOP = 5

ALL_INPUTS = 6

INPUT_D2 = 7

INPUT_D3 = 8

INPUT_D4 = 9

INPUT_D5 = 10

class InputSensorType(*value*)

Sensor type enumeration.

THERMOCOUPLE is only valid with the 3060 option, CAPACITANCE is only valid with the 3061 option.

DISABLED = 0

DIODE = 1

PLATINUM_RTD = 2

NTC_RTD = 3

THERMOCOUPLE = 4

CAPACITANCE = 5

class DiodeRange(*value*)

Diode voltage range enumeration

TWO_POINT_FIVE_VOLTS = 0

TEN_VOLTS = 1

class RTDRange(*value*)

RTD resistance range enumeration.

THIRTY_THOUSAND_OHM and ONE_HUNDRED_THOUSAND_OHM are only valid for NTC RTDs.

TEN_OHM = 0

THIRTY_OHM = 1

HUNDRED_OHM = 2

THREE_HUNDRED_OHM = 3

ONE_THOUSAND_OHM = 4

THREE_THOUSAND_OHM = 5

TEN_THOUSAND_OHM = 6

THIRTY_THOUSAND_OHM = 7

ONE_HUNDRED_THOUSAND_OHM = 8

class ThermocoupleRange(*value*)

Thermocouple range enumeration.

FIFTY_MILLIVOLT = 0

class HeaterOutputMode(*value*)

Control loop enumeration.

OFF = 0

CLOSED_LOOP = 1

ZONE = 2

OPEN_LOOP = 3

MONITOR_OUT = 4

WARMUP_SUPPLY = 5

class HeaterRange(*value*)

Current mode heater enumerations.

OFF = 0

LOW = 1

MEDIUM = 2

HIGH = 3

class HeaterVoltageRange(*value*)

Voltage mode heater enumerations.

VOLTAGE_OFF = 0

VOLTAGE_ON = 1

class DisplayFieldUnits(*value*)

Panel display units enumeration.

KELVIN = 1

CELSIUS = 2

SENSOR_UNITS = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

SENSOR_NAME = 6

Register Classes

This page describes the register objects. Each bit in the register is represented as a member of the register's class

lakeshore.model_336.**Model336StandardEventRegister**

alias of *StandardEventRegister*

class lakeshore.model_336.**Model336StatusByteRegister**(*message_available_summary_bit*, *event_status_summary_bit*, *service_request*, *operation_summary_bit*)

Class object representing the status byte register LSB to MSB.

bit_names = ['', '', '', '', 'message_available_summary_bit', 'event_status_summary_bit', 'service_request', 'operation_summary_bit']

class lakeshore.model_336.**Model336ServiceRequestEnable**(*message_available_summary_bit*, *event_status_summary_bit*, *operation_summary_bit*)

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

```
bit_names = ['', '', '', '', 'message_available_summary_bit',
'event_status_summary_bit', '', 'operation_summary_bit']
```

lakeshore.model_336.**Model336OperationEvent**

alias of *OperationEvent*

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

Class object representing the input status flag bits.

```
bit_names = ['invalid_reading', '', '', '', 'temp_underrange', 'temp_overrange',
'sensor_units_zero', 'sensor_units_overrange']
```

Model 350 Cryogenic Temperature Controller

The Model 350 measures and controls cryogenic temperature environments.

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

Instrument methods

class lakeshore.model_350.**Model350**(*serial_number=None*, *com_port=None*, *baud_rate=57600*, *data_bits=7*, *stop_bits=1*, *parity='O'*, *flow_control=False*, *handshaking=False*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing the Lake Shore Model 350 cryogenic temperature controller.

command(*command_string*)

Send a command to the instrument.

Args:

command_string (str):
A serial command.

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

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

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

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

query(*query_string*)

Send a query to the instrument and return the response.

Args:

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

Returns:
The instrument query response as a string.

write(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

Model 372 AC Resistance Bridge

The Model 372 is both an AC resistance bridge and temperature controller designed for measurements below 100 milliKelvin.

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

Example scripts

Setting a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↪VOLTS_PER_KELVIN, 300.0,
                                      myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set␣
↪to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.␣
↪The instrument generates
```

(continues on next page)

(continued from previous page)

```
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,␣
↪30, "SN123", (276, 10),
                                               (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can␣
↪then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that␣
↪curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Using enums to configure an input sensor

```
from lakeshore import Model372, Model372InputSetupSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure a sensor
# Create Model372InputSetupSettings object with current excitation mode, 31.6 uA␣
↪excitation current, autoranging on
# (tracking current), current source not shunted, preferred units of Kelvin, and a␣
↪resistance range of 20.0 kOhms
sensor_settings = Model372InputSetupSettings(my_model_372.SensorExcitationMode.CURRENT,
                                             my_model_372.MeasurementInputCurrentRange.
↪RANGE_31_POINT_6_MICRO_AMPS,
                                             my_model_372.AutoRangeMode.CURRENT, False,
                                             my_model_372.InputSensorUnits.KELVIN,
                                             my_model_372.MeasurementInputResistance.
↪RANGE_20_KIL_OHMS)

# Pass settings into method along with desired input channel
my_model_372.configure_input(1, sensor_settings)

# Get all readings (temperature, resistance, excitation power, quadrature) from sensor
sensor_1_readings = my_model_372.get_all_input_readings(1)

# Record readings to a file
file = open("372_sensor_1_data.csv", "w")
file.write("Header information\n")
# Call readings using the keys of the returned dictionary
file.write("Temperature Reading," + str(sensor_1_readings['kelvin']) + "\n")
```

(continues on next page)

(continued from previous page)

```
file.write("Resistance Reading," + str(sensor_1_readings['resistance']) + "\n")
file.write("Excitation Power," + str(sensor_1_readings['power']) + "\n")
file.write("Imaginary Part of Resistance," + str(sensor_1_readings['quadrature']) + "\n")
file.close()
```

Setting up a control loop with the model 372

```
from lakeshore import Model372, Model372HeaterOutputSettings,␣
↪Model372ControlLoopZoneSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure output for zone mode, controlled by control input, with power up enabled␣
↪filter enabled and a reading
# delay of 10 seconds
# Note; it's assumed that the control input is enabled and configured
heater_settings = Model372HeaterOutputSettings(my_model_372.OutputMode.ZONE, my_model_
↪372.InputChannel.CONTROL, True, True, 10)
my_model_372.configure_heater(1, heater_settings)

# Configure a relay for Warmup Heater Zone
my_model_372.set_relay_for_warmup_heater_control_zone(1)

# Set up control loop with an upper bound of 15K, a gain of 50, an integral value of␣
↪5000, a derivative of 2000, and a
# manual output of 50%. Range is set to true, setpoint ramp rate is set to 10 seconds,␣
↪and relay 1 is configured for
# the zone and relay 2 is not configured
control_loop_settings = Model372ControlLoopZoneSettings(15, 50.0, 5000, 2000, 50, True,␣
↪10, True, False)
# Set control loop on output 1 (Warmup Heater) in zone 4
my_model_372.set_control_loop_parameters(1, 4, control_loop_settings)

# Create a setpoint for 5 K
my_model_372.set_setpoint_kelvin(1, 5.0)
# Enable ramping to setpoint for output 1 at a rate of 10 Kelvin/minute
my_model_372.set_setpoint_ramp_parameter(1, True, 10)
```

Instrument class methods

class lakeshore.model_372.**Model372**(*baud_rate*, *serial_number=None*, *com_port=None*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing the Lake Shore Model 372 AC bridge and temperature controller.

clear_interface()

Clears the interface.

Clears all bits in the status byte register and the standard event status register. Does not clear the instrument.

`reset_instrument()`

Resets the instrument to power-up settings and parameters.

`set_display_settings(mode, number_of_fields='', displayed_info='')`

Sets which parameters to display and how to display them.

Args:

mode (self.DisplayMode):
Sets the input to monitor on the display, or configures display for custom.

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

displayed_info (self.DisplayInfo):
Determines whether to display information about the loop of the active scan channel or a specific heater in the bottom left of the display in custom mode.

`get_display_mode()`

Returns the current mode of the display.

Returns:

(self.DisplayMode):
Enumerated object representing the current mode of the display.

`get_custom_display_settings()`

Returns the settings of the display in custom mode.

Returns:

(dict):
mode: self.DisplayMode, number_of_fields: self.DisplayFields, displayed_info: self.DisplayInfo

`get_resistance_reading(input_channel)`

Returns the input reading in Ohms.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (for control input).

Returns:

(float):
Sensor reading in Ohms.

`get_quadrature_reading(input_channel)`

Returns the imaginary part of the reading in Ohms. Only valid for measurement inputs.

Args:

input_channel (int):
Specifies which input channel to read from. Options are: 1-16.

Returns:

(float):
The imaginary part of the sensor reading, in Ohms.

`get_all_input_readings(input_channel)`

Returns the kelvin reading, resistance reading, and, if a measurement input, the quadrature reading.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (for control input).

Returns:

(dict):

- **If measurement input:**
  - {kelvin: float, resistance: float, power: float, quadrature: float)
- **If control input:**
  - {kelvin: float, resistance: float, power: float}

`get_input_setup_parameters`(*input_channel*)

Returns the settings on the specified input.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

input_sensor_settings (Model372InputSetupSettings):
object of Model372InputSetupSettings representing the parameters of the excitation of the sensor on the specified channel

`configure_input`(*input_channel*, *settings*)

Sets the desired setup settings on the specified input.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

settings (Model372InputSetupSettings):
Object of Model372InputSetupSettings representing the parameters of the excitation of the sensor on the specified channel.

`disable_input`(*input_channel*)

Disables the desired input channel.

Args:

input_channel (str or int):
Specifies which input channel to disable. Options are: 1-16, or "A" (control input).

`get_input_channel_parameters`(*input_channel*)

Returns the settings on the specified input channel.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

input_channel_settings (Model372InputChannelSettings):
Contains variables representing the different channel settings parameters.

`set_input_channel_parameters`(*input_channel*, *settings*)

Sets the desired channel settings on the specified input channel.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

settings (Model372InputChannelSettings):
Defines how to set the various parameters.

`get_analog_heater_output`(*output_channel*)

Returns the output of the warm-up or analog/still heater.

Args:

output_channel (int):

Specifies which heater to read from. Options:
1 output 1 (warm up heater), or 2 output 2 (analog heater).

Returns:

reading (float):
Output of the analog heater being queried.

`all_off`()

Recreates the front panel safety feature of shutting off all heaters.

`set_heater_output_range`(*output_channel*, *heater_range*)

Sets the output range.

Args:

output_channel (int):
Specifies which heater to set. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

heater_range (Enum or bool):
Specifies the range of the output. Options: Sample Heater (Enum) - Object of type self.SampleHeaterOutputRange. Warmup Heater/Still Heater (bool) - False: output off, True: output on.

`get_heater_output_range`(*output_channel*)

Return's the range of the output on a given channel.

Args:

output_channel (int):
Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

Returns:

heater_range (bool or Enum):
If channel 1 or 2, returns bool for if output is on or off. If channel 0, an object of enum type SampleHeaterOutputRange.

`set_filter`(*input_channel*, *state*, *settle_time*, *window*)

Sets a filter for the specified input channel.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 0 (all channels/measurement inputs), 1-16, or "A" (control input).

state (bool):
Specifies whether to turn filter on or off. Options are: False for off, or True for on.

settle_time (float):
Specifies filter settle time. Options are: 1 - 200 s.

window (float):
Specifies what percent of full scale reading limits the filtering function. Options are: 1 - 80.

`get_filter`(*input_channel*)

Returns information about the filter set on the specified channel.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

state (bool):
Specifies whether to turn filter on or off.

settle_time (int):
Specifies filter settle time.

window (int):
Specifies what percent of full scale reading limits the filtering function.

`set_ieee_interface_parameter`(*address*)

Sets the IEEE address of the instrument.

Args:

address (int):
Specifies the IEEE address. Options are: 1 - 30.

`get_ieee_interface_parameter`()

Returns the IEEE address of the instrument.

Returns:

address (int):
The IEEE address.

`get_excitation_power`(*input_channel*)

Returns the most recent power calculation for the selected input channel.

Args:

input_channel (str or int):
Specifies which input channel to read from. Options are: 1-16, or "A" (control input).

Returns:

power (float):
Most recent power calculation for the input being queried.

`get_heater_output_settings`(*output_channel*)

Returns the mode and settings of the given output channel.

Args:

output_channel (int):
Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

Returns:

outputmode_settings (Model372HeaterOutputSettings):
Object of class Model372HeaterOutputSettings whose variables are set to reflect the current output settings of the queried heater.

`configure_heater`(*output_channel*, *settings*)

Sets up a heater output.

Analog heaters (outputs 1 and 2) might need to configure further settings in configure_analog_heater.

Args:

output_channel (int):
Specifies which heater to read from. Options: 0: sample heater, 1: output 1 (warm up heater), or 2: output 2 (analog heater).

settings (Model372HeaterOutputSettings):
Defines how to set the output mode settings.

`set_common_mode_reduction`(*state*)

Sets common mode reduction to given state for all measurement channels.

Args:

state (bool):
Sets CMR to enabled or disable. Options are: False (for disable), or True (for enable).

`get_common_mode_reduction`()

Returns whether CMR is set for measurement channels.

Returns:
False (boolean) if CMR is disabled, or True (boolean) if CMR is enabled.

`set_scanner_status`(*input_channel*, *status*)

Sets the scanner to the specified channel, and enables or disables auto scan.

Args:

input_channel (int):
Specifies which measurement input to set the scanner to. Options are: 1 - 16.

status (bool):
Specifies whether to turn auto scan feature on. Options are: False (disable), True (enable).

`get_scanner_status`()

Returns which channel the scanner is on and whether the auto scan feature is enabled.

Returns:

input_channel (int):
The measurement channel the scanner is currently on.

status (bool):
True if auto-scan in on, or False if auto-scan is off.

`set_alarm_beep`(*status*)

Enables or disables a beep for alarms.

Args:

status (bool):
False (for disable), or True (for enable).

`get_alarm_beep_status`()

Returns whether beep for alarms is enabled or disabled.

Returns

status (bool):
True (beep is enabled), or False (beep is disabled).

`set_still_output`(*power*)

Sets the still output of the still/analog heater to power% of full power.

Heater gets configured for still mode if not currently configured.

Args:

power (float):
Specifies the percent of full power for still output. Options are: 0 - 100.

`get_still_output`()

Returns the percent of full power being outputted by still heater in still mode.

Returns:

power (float):
Percent of full power being outputted by heater.

`set_warmup_output`(*auto_control*, *current*)

Sets up the warmup output to continuous control at the percent current specified.

Configures the warmup heater for continuous control mode from the control input.

Args:

auto_control (bool):
Specifies whether to turn on auto control. Options are: False for auto off, or True for continuous.

current (float):
Specifies percent of full current to apply to external output. Options are: 0 - 100

`get_warmup_output`()

Returns the control setting and percent current outputted in the warmup heater in warmup mode.

Returns:

auto_control (bool):
Specifies whether to turn on auto control. Returns: False for auto off, or True for continuous

current (float):
Specifies percent of full current to apply to external output.

`set_setpoint_kelvin`(*output_channel*, *setpoint*)

Sets the control set-point in Kelvin. Changes input parameters so preferred units are Kelvin.

Args:

output_channel (int):
Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

setpoint (float):
Specifies the set-point the heater ramps to, in Kelvin.

`set_setpoint_ohms`(*output_channel*, *setpoint*)

Sets the control set-point in Ohms. Changes input parameters so preferred units are Ohms.

Args:

output_channel (int):
Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

setpoint (float):
Specifies the set-point the heater ramps to, in Kelvin.

`get_setpoint_kelvin`(*output_channel*)

Returns the set-point for the given output channel in kelvin.

Changes the control input's preferred units to Kelvin as a result.

Args:

output_channel (int):
Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

Returns:

setpoint (float):
Set-point of the output in Kelvin.

`get_setpoint_ohms`(*output_channel*)

Returns the set-point for the given output channel in kelvin.

Changes the control input's preferred units to Kelvin as a result.

Args:

output_channel (int):
Specifies which heater to set a set-point. Options are: 0: sample heater, or 1: output 1 (warm up heater).

Returns:

setpoint (float):
Set-point of the output in Ohms.

`get_excitation_frequency`(*input_channel*)

Returns the excitation frequency in Hz for either the measurement or control inputs.

Args:

input_channel (int or str):
Specifies which input to get frequency from. Options are: 0 : measurement inputs, or "A" : control input.

Returns:

frequency (Enum):
The excitation frequency in Hz, returned as an object of self.InputFrequency Enum type.

`set_excitation_frequency`(*input_channel*, *frequency*)

Sets the excitation frequency (in Hz) for either the measurement or control inputs.

Args:

input_channel (int or str):
Specifies which input to get frequency from. Options are: 0 : measurement inputs, or "A" : control input.

frequency (Enum):
The excitation frequency in Hz (if float), represented as an object of type self.InputFrequency.

`set_digital_output`(*bit_weight*)

Sets the status of the 5 digital output lines to high or low.

Args:

bit_weight (DigitalOutputRegister):
Determines which bits to set or reset.

`get_digital_output`()

Returns which digital output bits are set or reset by representing them in a binary number.

Returns:

bit_weight (DigitalOutputRegister):
Determines which bits to set or reset.

`set_interface`(*interface*)

Sets the interface for the instrument to communicate over.

Args:

interface (self.Interface):
Selects the interface based on the values as defined in the self.Interface enum class.

`get_interface`()

Returns the interface connected to the instrument.

Returns:

interface (self.Interface):
Returns the interface as an object of the self.Interface enum class.

`set_alarm_parameters`(*input_channel*, *alarm_enable*, *alarm_settings=None*)

Sets an alarm on the specified channel as defined by parameters.

Args:

input_channel (int or str):
Defines which channel to configure an alarm on. Options are: 0 for all measurement inputs, 1 - 16, or "A" for control input.

alarm_enable (bool)
Defines whether to turn alarm on or off.

alarm_settings (Model372AlarmParameters)
Model372AlarmParameters object containing desired alarm settings. Optional if alarm is disabled.

`get_alarm_parameters`(*input_channel*)

Returns the parameters for the alarm set for the input at the specified channel.

Args:

input_channel (int or str):
Defines which channel to configure an alarm on. Options are: 1 - 16, or "A" for control input.

Returns:

(dict):
{"alarm_enable": bool, "alarm_settings": Model372AlarmParameters}

`set_relay_for_sample_heater_control_zone`(*relay_number*)

Configures a relay to follow the sample heater output as part of a control zone.

Settings can be further configured in set_control_loop_zone_parameters method.

Args:

relay_number (int):
The relay to configure. Options are: 1 or 2.

`set_relay_for_warmup_heater_control_zone`(*relay_number*)

Configures a relay to follow the warm-up heater output as part of a control zone.

Settings can be further configured in set_control_loop_zone_parameters method.

Args:

relay_number (int):
The relay to configure. Options are: 1 or 2.

`get_ieee_interface_mode`()

Returns the IEEE interface mode of the instrument.

Returns:

mode (self.InterfaceMode):
Returns the mode as an enum type of class self.InterfaceMode.

`set_ieee_interface_mode`(*mode*)

Sets the IEEE interface mode of the instrument.

Args:

mode (self.InterfaceMode):
Defines the mode of the instrument as an object of the enum type Model372IEEEInterfaceMode.

`set_monitor_output_source`(*source*)

Sets the source of the monitor output. Also affects the reference output.

Args:

source (self.MonitorOutputSource):
Defines the source to run the monitor output off of.

`get_monitor_output_source`()

Returns the source for the monitor output.

Returns:

source (MonitorOutputSource):
Returns the source as an object of the MonitorOutputSource class.

`get_warmup_heater_setup()`

Returns the settings regarding the resistance, current and units of the warmup heater (output channel 1).

Returns:

(dict):
{"resistance": float, "max_current": float, "units": self.HeaterOutputUnits}

`get_sample_heater_setup()`

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

Returns:

(dict):
{"resistance": float, "units": self.HeaterOutputUnits}

`setup_warmup_heater(`*resistance*, *max_current*, *units*`)`

Configures the current and power of the warmup heater (output channel 1).

The max current must not cause the heater to exceed it's max power (calculated by I = sqrt(P/R)) or it's max voltage (calculated by I = V/R). Check your heater's specifications before setting the max current, and use the lower current produced from the two calculations.

Args:

resistance (self.HeaterResistance):
Heater load in ohms, as an object of the enum type self.HeaterResistance.

max_current (float):
User specified max current in A.

units (self.HeaterOutputUnits):
Defines which units the output is displayed in (Current (A) or Power (W)).

`setup_sample_heater(`*resistance*, *units*`)`

Configures the current and power of the sample heater (output channel 0.)

Args:

resistance (float):
Heater load in ohms. Options are: 1 - 2000.

units (self.HeaterOutputUnits):
Defines which units the output is displayed in (Current (A) or Power (W)).

`configure_analog_monitor_output_heater(`*source*, *high_value*, *low_value*, *settings=None*`)`

Configures the still heater's analog settings for Monitor Out mode.

Can fully configure the heater by including the settings parameter, but it is recommended to configure non-analog properties of the heater through the configure_heater method.

Args:

source (self.InputSensorUnits):
The units to use for channel data.

high_value (float):
The data at which the output reaches +100% output.

low_value (float):
The data at which the outputs reach 0% output for unipolar output, or -100% for bipolar. output.

settings (Model372HeaterOutputSettings):
Optional if heater is already configured using configure_heater. Gives non-analog configurations for heater.

`get_analog_monitor_output_settings()`

Retrieves the analog monitor settings of output 2 configured in monitor output mode.

Returns:

(dict):
{"source": self.InputSensorUnits, "high_value": float, "low_value": float}

`configure_analog_heater(output_channel, manual_value, settings=None)`

Configures the analog settings of a heater for modes other than Monitor Out.

(Use configure_analog_monitor_out_heater for Monitor Out mode). Can fully configure the heater by including the settings parameter, but it is recommended to first configure the heater using the configure_heater method before using this method.

Args:

output_channel (Model372HeaterOutput):
The output to configure.

manual_value (float):
The value of the analog output as it applies to the set analog mode.

settings (Model372HeaterOutputSettings):
Optional if heater is already configured using configure_heater. Gives non-analog configurations for heater.

`get_analog_manual_value(output_channel)`

Returns the manual value of an analog heater.

The manual value is the analog value used for Open Loop, Closed Loop, Warm Up, or Still mode.

Args:

output_channel (int):
The analog output to query. Options are: 1 (Warm up heater), or 2 (Still heater).

Returns:

(float):
The manual analog value for the heater.

`set_website_login(username, password)`

Sets the username and password to connect instrument to website.

Args:

username (str):
Username to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

password (str):
Password to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

`get_website_login()`

Returns the set username and password for web login for the instrument.

Returns:

username (str):
The current set username for the web login

password (str):
The current set password for the web login

`get_control_loop_zone_parameters(output_channel, zone)`

Returns the settings parameters of the control loop on the specified output channel and zone.

Args:

output_channel (int):
Channel of the heater being queried. Options are: 0 for sample heater, or 1 for warm-up heater.

zone (int):
Control loop zone to configure. Options are: 1 - 10.

Returns:

settings (Model372ControlLoopZoneSettings):
An object of the Model372ControlLoopZoneSettings class containing information of the settings in the values of its variables.

`set_control_loop_parameters(output_channel, zone, settings)`

Returns the parameters of the control loop set in the specified zone for the specified heater output.

Args:

output_channel (int):
Channel of the heater being queried. Options are: 0 for sample heater, or 1 for warm-up heater.

zone (int):
Control loop zone to configure. Options are: 1 - 10.

settings (Model372ControlLoopZoneSettings):
An object of the Model372ControlLoopZoneSettings with the variable set to configure the desired settings.

`get_reading_status(input_channel)`

Returns any flags raised during a measurement reading.

Args:

input_channel (str or int):
The input whose reading status is being queried. Options are: 1 - 16, or "A" (control input).

Returns:

bit_states (dict):
Dictionary containing the names of the flag and a boolean value corresponding to if the flag is raised or not.

Instrument settings classes and registers

class lakeshore.model_372.**Model372InputChannelSettings**(*enable*, *dwell_time*, *pause_time*, *curve_number*, *temperature_coefficient=None*)

Class object representing parameters for the channel settings of an self.InputChannel.

__init__(*enable*, *dwell_time*, *pause_time*, *curve_number*, *temperature_coefficient=None*)

The constructor for Model372InputChannelSettings class.

Args:

enable (bool):
Whether to enable or disable input.

dwell_time (int):
Specifies a value for the auto-scanning dwell time in seconds. Not applicable to control input. Options are: 1 to 200 s.

pause_time (int):
Specifies a value for the change pause time in seconds. Options are: 3 to 200 s.

curve_number (int):
Specifies which calibration curve to use on input sensor. Options are: 0 (none), or 1 - 59.

temperature_coefficient (self.CurveTemperatureCoefficient):
Sets coefficient for temperature control if no curve is selected.

class lakeshore.model_372.**Model372InputSetupSettings**(*mode*, *excitation_range*, *auto_range*, *current_source_shunted*, *units*, *resistance_range=None*)

Class object representing parameters for the sensor and measurement settings of an self.InputChannel.

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

The constructor for Model372InputSetupSettings class.

Args:

mode (self.SensorExcitationMode):
Determines whether to use current or voltage for sensor excitation.

excitation_range (IntEnum):
The voltage or current (depending on mode) excitation range.

auto_range (Model372AutoRangeMode):
Specifies whether auto range is Off, Auto-ranging Current, or in ROX 102B mode.

current_source_shunted (bool):
Specifies whether the current source is shunted. If current source is shunted, excitation is off. If current source is not shunted, excitation is on.

units (self.InputSensorUnits):
Specifies the preferred units, Kelvin or Ohms, for the sensor.

resistance_range (Model372MeasurementInputResistance):
For measurement inputs only, specifies the measurement input resistance range.

class lakeshore.model_372.**Model372HeaterOutputSettings**(*output_mode*, *input_channel*, *powerup_enable*, *reading_filter*, *delay*, *polarity=None*)

Class object representing parameters to configure Heater Output Settings.

__init__(*output_mode*, *input_channel*, *powerup_enable*, *reading_filter*, *delay*, *polarity=None*)

The constructor for Model372HeaterOutputSettings class.

Args:

output_mode (self.OutputMode):
The control or output mode to configure the heater for. Defines how the output is controlled.

input_channel (self.InputChannel):
Which input to control output from in a control loop.

powerup_enable (bool):
Specifies whether output stays on after powerup cycle. True if enabled, False if disabled.

reading_filter (bool):
Specifies whether readings are filtered on unfiltered. True if filtered, False if unfiltered.

delay (int):
Specifies delay in seconds for set-point during AutoScanning. Options are: 1 - 255.

polarity (self.Polarity):
Specifies output polarity. Not applicable to warmup heater.

class lakeshore.model_372.**Model372ControlLoopZoneSettings**(*upper_bound*, *p_value*, *i_value*, *d_value*, *manual_output*, *heater_range*, *ramp_rate*, *relay_1*, *relay_2*)

Defines the parameters to set up a Control Loop.

__init__(*upper_bound*, *p_value*, *i_value*, *d_value*, *manual_output*, *heater_range*, *ramp_rate*, *relay_1*, *relay_2*)

The constructor for Model372ControlLoopZoneSettings class.

Args:

upper_bound (float):
Upper bound setpoint in Kelvin.

p_value (float):
The gain for a PID system. Options are: 0.0 - 1000.

i_value (float):
The integral value for a PID system. Options are: 0 - 10000.

d_value (float):
The rate for a PID system. Options are: 0 - 2500.

manual_output (float):
Percentage full scale manual output.

heater_range (float or bool):
Heater range for the control zone. Entered as a float for the sample heater. Entered as a bool for the warm-up heater.

ramp_rate (float):
Specifies ramp rate for this zone.

relay_1 (bool):
Specifies if relay 1 is on or off. Only applicable if relay is configured in zone mode and relay's control output matches configured output.

relay_2 (bool):
Specifies if relay 2 is on or off. Only applicable if relay is configured in zone mode and relay's control output matches configured output.

class lakeshore.model_372.**Model372AlarmParameters**(*high_value*, *low_value*, *deadband*, *latch_enable*, *audible=None*, *visible=None*)

Sets up an alarm for an input channel.

__init__(*high_value*, *low_value*, *deadband*, *latch_enable*, *audible=None*, *visible=None*)

The constructor for Model372AlarmParameters class.

Args:

high_value (int):
Sets value for source to be checked against to set high alarm.

low_value (int):
Sets value for source to be checked against to set low alarm.

deadband (int):
Sets value that source must change outside an alarm condition to deactivate an unlatched alarm.

latch_enable (bool):
Specifies if alarm is latched or not.

audible (bool):
Specifies if an alarm is audible or not.

visible (bool):
Specifies if an alarm is visible via LED on front panel or not.

lakeshore.model_372.**Model372CurveHeader**

alias of *CurveHeader*

class lakeshore.model_372.**CurveHeader**(*curve_name*, *serial_number*, *curve_data_format*, *temperature_limit*, *coefficient*)

A class to configure the temperature sensor curve header parameters.

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

Constructor for CurveHeader class.

Args:

curve_name (str):
Specifies curve name (limit of 15 characters).

serial_number (str):
Specifies curve serial number (limit of 10 characters).

curve_data_format (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 *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.**Model372Enums**

Class containing the enums relevant to the Model 372.

class **OutputMode**(*value*)

Enumeration of the different modes for heater output setup.

OFF = 0

MONITOR_OUT = 1

OPEN_LOOP = 2

ZONE = 3

STILL = 4

CLOSED_LOOP = 5

WARMUP = 6

class **InputChannel**(*value*)

Enumeration of the input channels of the Model 372.

NONE = 0

ONE = 1

TWO = 2

THREE = 3

FOUR = 4

FIVE = 5

SIX = 6

SEVEN = 7

EIGHT = 8

NINE = 9

TEN = 10

ELEVEN = 11

TWELVE = 12

THIRTEEN = 13

FOURTEEN = 14

FIFTEEN = 15

SIXTEEN = 16

CONTROL = 'A'

class SensorExcitationMode(*value*)

Enumeration of the possible excitation modes for an input sensor.

VOLTAGE = 0

CURRENT = 1

class AutoRangeMode(*value*)

Enumeration for the possible modes of the auto ranging feature.

ROX102B mode is a special auto-ranging mode that applies only to Lake Shore ROX-102B sensor.

OFF = 0

CURRENT = 1

ROX102B = 2

class InputSensorUnits(*value*)

Enumeration of the units to handle input readings and display in.

KELVIN = 1

OHMS = 2

class MonitorOutputSource(*value*)

Enumeration of the source for an output to monitor.

OFF = 0

CS_NEG = 1

CS_POS = 2

VCM_NEG = 3

VCM_POS = 4

VDIF = 5

VAD_MEASUREMENT = 6

VAD_CONTROL = 7

class RelayControlMode(*value*)

Enumeration of the control modes of the configurable relays of the 372.

RELAY_OFF = 0

RELAY_ON = 1

ALARMS = 2

SAMPLE_HEATER_ZONE = 3

WARMUP_HEATER_ZONE = 4

class DisplayMode(*value*)

Enumeration of the possible information to display.

MEASUREMENT_INPUT = 0

CONTROL_INPUT = 1

CUSTOM = 2

class DisplayInfo(*value*)

Enumeration of the information to a display in the bottom left of the custom display mode.

NONE = 0

SAMPLE_HEATER = 1

WARMUP_HEATER = 2

ACTIVE_SCAN_CHANNEL = 3

class CurveFormat(*value*)

Enumeration of the units to use in a calibration curve.

OHM_PER_KELVIN = 3

LOGOHM_PER_KELVIN = 4

OHM_PER_KELVIN_CUBIC_SPLINE = 7

class DisplayFieldUnits(*value*)

Enumeration for the possible units to display in a single display field.

KELVIN = 1

OHMS = 2

QUADRATURE = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

SENSOR_NAME = 6

class SampleHeaterOutputRange(*value*)

Enumeration of the output range of the sample heater (output 0).

OFF = 0

RANGE_31_POINT_6_MICRO_AMPS = 1

RANGE_100_MICRO_AMPS = 2

RANGE_316_MICRO_AMPS = 3

RANGE_1_MILLI_AMP = 4

RANGE_3_POINT_16_MILLI_AMPS = 5

RANGE_10_MILLI_AMPS = 6

RANGE_31_POINT_6_MILLI_AMPS = 7

RANGE_100_MILLI_AMPS = 8

class InputFrequency(*value*)

Defines the enumeration of the excitation frequency of an input.

FREQUENCY_9_POINT_8_HZ = 1

FREQUENCY_13_POINT_7_HZ = 2

FREQUENCY_16_POINT_2_HZ = 3

FREQUENCY_11_POINT_6_HZ = 4

FREQUENCY_18_POINT_2_HZ = 5

class MeasurementInputVoltageRange(*value*)

Enumerates the possible voltage ranges for a measurement input.

RANGE_2_MICRO_VOLTS = 1

RANGE_6_POINT_32_MICRO_VOLTS = 2

RANGE_20_MICRO_VOLTS = 3

RANGE_63_POINT_2_MICRO_VOLTS = 4

RANGE_200_MICRO_VOLTS = 5

RANGE_632_MICRO_VOLTS = 6

RANGE_2_MILLI_VOLTS = 7

RANGE_6_POINT_32_MILLI_VOLTS = 8

RANGE_20_MILLI_VOLTS = 9

RANGE_63_POINT_2_MILLI_VOLTS = 10

RANGE_200_MILLI_VOLTS = 11

RANGE_632_MILLI_VOLTS = 12

class MeasurementInputCurrentRange(*value*)

Enumeration of the current range of a measurement input.

RANGE_1_PICO_AMP = 1

RANGE_3_POINT_16_PICO_AMPS = 2

RANGE_10_PICO_AMPS = 3

RANGE_31_POINT_6_PICO_AMPS = 4

RANGE_100_PICO_AMPS = 5

RANGE_316_PICO_AMPS = 6

RANGE_1_NANO_AMP = 7

RANGE_3_POINT_16_NANO_AMPS = 8

RANGE_10_NANO_AMPS = 9

RANGE_31_POINT_6_NANO_AMPS = 10

RANGE_100_NANO_AMPS = 11

RANGE_316_NANO_AMPS = 12

RANGE_1_MICRO_AMP = 13

RANGE_3_POINT_16_MICRO_AMPS = 14

RANGE_10_MICRO_AMPS = 15

RANGE_31_POINT_6_MICRO_AMPS = 16

RANGE_100_MICRO_AMPS = 17

RANGE_316_MICRO_AMPS = 18

RANGE_1_MILLI_AMP = 19

RANGE_3_POINT_16_MILLI_AMPS = 20

RANGE_10_MILLI_AMPS = 21

RANGE_31_POINT_6_MILLI_AMPS = 22

class ControlInputCurrentRange(*value*)

Enumeration of the current range of the control input.

RANGE_316_PICO_AMPS = 1

RANGE_1_NANO_AMP = 2

RANGE_3_POINT_16_NANO_AMPS = 3

RANGE_10_NANO_AMPS = 4

RANGE_31_POINT_6_NANO_AMPS = 5

RANGE_100_NANO_AMPS = 6

class MeasurementInputResistance(*value*)

Enumeration of the resistance range of a measurement input.

RANGE_2_MILLI_OHMS = 1

RANGE_6_POINT_32_MILLI_OHMS = 2

RANGE_20_MILLI_OHMS = 3

RANGE_63_POINT_2_MILLI_OHMS = 4

RANGE_200_MILLI_OHMS = 5

RANGE_632_MILLI_OHMS = 6

RANGE_2_OHMS = 7

RANGE_6_POINT_32_OHMS = 8

RANGE_20_OHMS = 9

RANGE_63_POINT_2_OHMS = 10

RANGE_200_OHMS = 11

RANGE_632_OHMS = 12

RANGE_2_KIL_OHMS = 13

RANGE_6_POINT_32_KIL_OHMS = 14

RANGE_20_KIL_OHMS = 15

RANGE_63_POINT_2_KIL_OHMS = 16

RANGE_200_KIL_OHMS = 17

RANGE_632_KIL_OHMS = 18

RANGE_2_MEGA_OHMS = 19

RANGE_6_POINT_32_MEGA_OHMS = 20

RANGE_20_MEGA_OHMS = 21

RANGE_63_POINT_2_MEGA_OHMS = 22

class HeaterOutput(*value*)

Enumeration of heater output.

```
WARM_UP_HEATER = 1

STILL_HEATER = 2
```

2.4.5 Temperature Monitors

Model 224 Temperature Monitor

The Lake Shore Model 224 measures up to 12 temperature sensor channels.

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

Example Scripts

Configuring the model 224 with a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224, Model224CurveHeader

# Connect to a temperature instrument (the Model 224 in this case) over USB
myinstrument = Model224()

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", myinstrument.CurveFormat.
↪VOLTS_PER_KELVIN, 300.0,
                                      myinstrument.CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

# Edit individual data points of the curve. In this case, a sensor value of 1.23 is set␣
↪to equal a Kelvin value of
# 276.0
myinstrument.set_curve_data_point(25, 1, 1.23, 276.0)

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature points.␣
↪The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(myinstrument.SoftCalSensorTypes.DT_400,␣
↪30, "SN123", (276, 10),
                                               (300, 5), (310, 2))

# Use the get_curve method to get all the data points for a curve as a list. This can␣
↪then be used to create a plot
# of the calibration curve.
data_points_list = myinstrument.get_curve(30)
x_list = [item[0] for item in data_points_list]
y_list = [item[1] for item in data_points_list]
```

(continues on next page)

(continued from previous page)

```
plt.plot(x_list, y_list)

# Finally, a curve can be deleted if you would like to reset the data points for that␣
↪curve. Only user curves
# can be deleted.
myinstrument.delete_curve(25)
```

Instrument class methods

class lakeshore.model_224.**Model224**(*serial_number=None*, *com_port=None*, *baud_rate=57600*, *data_bits=7*, *stop_bits=1*, *parity='O'*, *flow_control=False*, *handshaking=False*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing the Lake Shore Model 224 temperature monitor.

command(**commands*, *check_errors=True*)

Sends an SCPI command or multiple commands to the instrument.

Args:

commands (str):
A serial command.

check_errors (bool):
Chooses whether to check for and raise errors after sending a command. True by default. kwarg. Optional Parameter

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

Send a query to the instrument and return the response.

Args:

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

Returns:
The instrument query response as a string.

get_standard_event_enable_mask()

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

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

set_standard_event_enable_mask(*register_mask*)

Configures values of the standard event enable register bits.

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

Args:

register_mask (Model224StandardEventRegister):
An StandardEventRegister class object with all bits set to a value.

`clear_interface_command()`

Clears the bits of the interface and terminates all pending operations.

Clears the bits in the Status Byte Register, Standard Event Status Register, and Operation Event Register, and terminates all pending operations. Clears the interface, but not the controller.

`reset_instrument()`

Sets controller parameters to power-up settings.

`set_service_request(register_mask)`

Manually enable/disable the mask of the corresponding status-flag bit in the status byte register.

Args:

register_mask (Model224ServiceRequestRegister):
A Model224ServiceRequestRegister class object with all bits configured.

`get_service_request()`

Returns the status byte register bits and their values as a class instance.

`get_status_byte()`

Returns the status flag bits as a class instance without resetting the register.

`get_self_test()`

Instrument self test result completed at power up.

Returns:

test_errors (bool):
True means errors found, and False means no errors found.

`set_wait_to_continue()`

Causes the IEEE-488 interface to hold off until all pending operations have been completed.

This has the same function as the set_operation_complete() method, except that it does not set the Operation Complete event bit in the Event Status Register.

`set_to_factory_defaults()`

Sets all the settings and configurations to their factory default values.

`get_reading_status(input_channel)`

Returns the reading status of any input status flags that may be set.

Args:

input_channel (str):
The input to check for reading status flags. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(dict):
{"invalid_reading": bool, "temperature_under_range": bool, "temperature_over_range": bool, "sensor_units_zero": bool, "sensor_units_over_range": bool}

`get_kelvin_reading(input_channel)`

Returns the temperature value in kelvin of either channel.

Args:

input_channel:
Selects the channel to retrieve measurement. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(float):
The reading of the sensor in kelvin.

`get_sensor_reading`(*input_channel*)

Returns the sensor reading in the sensor's units.

Args:

input_channel:
Selects the channel to retrieve measurement. Options are: Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

reading (float):
The raw sensor reading in the units of the connected sensor.

`get_celsius_reading`(*input_channel*)

Returns the given input's temperature reading in degrees Celsius.

Args:

input_channel (str):
Selects input to retrieve measurement from. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(float):
Temperature readings in degrees Celsius.

`get_all_inputs_celsius_reading`()

Returns the temperature reading in degrees Celsius of all the inputs.

Returns:

(dict):
{"input_a_reading": float, "input_b_reading": float, "input_c1_reading": float, "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.

Returns:

(int):
Contrast level of the front panel LCD screen.

`set_ieee_488`(*address*)

Specifies the IEEE address.

Args:

address (int):
1-30 (0 and 31 reserved).

`get_ieee_488`()

Returns the IEEE address set.

Returns:

address (int):
1-30 (0 and 31 reserved).

`set_led_state`(*state*)

Sets the front panel LEDs to on or off.

Args:

state (bool):
Sets the LEDs to functional or nonfunctional. Options are: False for off, True for on.

`get_led_state()`

Returns whether front panel LEDs are enabled.

Returns:

state (bool):
Specifies whether front panel LEDs are functional. Returns: False if disabled, True enabled.

`set_keypad_lock(state, code)`

Locks or unlocks front panel keypad (except for alarms and disabling heaters).

Args:

state (bool):
Sets the keypad to locked or unlocked. Options are: False for unlocked or True for locked.

code (int):
Specifies 3 digit lock-out code. Options are: 000 - 999.

`get_keypad_lock()`

Returns the state of the keypad lock and the lock-out code.

Returns:

(dict):
{"state": bool, "code": int}

`get_min_max_data(input_channel)`

Returns the minimum and maximum data from an input.

Args:

input_channel (str):
Specifies which input to query.

Returns:

min_max_data (dict):
{"minimum": float, "maximum": float}

`reset_min_max_data()`

Resets the minimum and maximum input data.

`set_input_curve(input_channel, curve_number)`

Specifies the curve an input uses for temperature conversion.

Args:

input_channel (str):
Specifies which input to configure.

curve_number (int):
0 = none, 1-20 = standard curves, 21-59 = user curves.

`get_input_curve(input_channel)`

Returns the curve number being used for a given input.

Args:

input_channel (str):
Specifies which input to query.

Returns:

curve_number (int):
0-59.

`set_website_login`(*username*, *password*)

Sets the username and password to connect instrument to website.

Args:

username (str):
Username to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

password (str):
Password to set for login. Must be less than or equal to 15 characters. Method automatically puts quotation marks around string, so they are not needed in the string literal passed into the method.

`get_website_login`()

Returns the set username and password for web login for the instrument.

Returns:

(dict):
{"username": str, "password": str}

`set_alarm_parameters`(*input_channel*, *alarm_enable*, *alarm_settings=None*)

Configures the alarm parameters for an input.

Args:

input_channel (str):
Specifies which input to configure.

alarm_enable (bool):
Specifies whether to turn on the alarm for the input, or turn the alarm off.

alarm_settings (Model224AlarmParameters):
See Model224AlarmParameters class. Optional if alarm_enable is set to False.

`get_alarm_parameters`(*input_channel*)

Returns the present state of all alarm parameters.

Args:

input_channel (str):
Specifies which input to configure.

Returns:

(dict):
{"alarm_enable": bool, "alarm_settings": Model224AlarmParameters}.

`get_alarm_status`(*input_channel*)

Returns the high state and low state of the alarm for the specified channel.

Args:

input_channel (str):
Specifies which input channel to read from. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(dict):
{"high_state": bool, "low_state": bool}

`reset_alarm_status()`

Clears the high and low status of all alarms.

`set_curve_header(curve_number, curve_header)`

Configures the user curve header.

Args:

curve_number (int):
Specifies which curve to configure. Options are: 21 - 59.

curve_header (Model224CurveHeader):
A Model224CurveHeader class object containing the desired curve information.

`get_curve_header(curve)`

Returns parameters set on a particular user curve header.

Args:

curve (int):
Specifies a curve to retrieve. Options are: 21 - 59.

Returns:

header (Model224CurveHeader):
A Model224CurveHeader class object containing the desired curve information.

`set_curve_data_point(curve, index, sensor_units, temperature)`

Configures a user curve point.

Args:

curve (int or str):
Specifies which curve to configure.

index (int):
Specifies the points index in the curve.

sensor_units (float):
Specifies sensor units for this point to 6 digits.

temperature (float):
Specifies the corresponding temperature in Kelvin for this point to 6 digits.

`get_curve_data_point(curve, index)`

Returns a standard or user curve data point.

Args:

curve (int):
Specifies which curve to query.

index (int):
Specifies the points index in the curve.

Returns:

curve_point (tuple):
(sensor_units: float, temp_value: float)).

`delete_curve`(*curve*)

Deletes the user curve.

Args:

curve (int):
Specifies a user curve to delete.

`generate_and_apply_soft_cal_curve`(*source_curve*, *curve_number*, *serial_number*, *calibration_point_1*, *calibration_point_2=(0, 0)*, *calibration_point_3=(0, 0)*)

Creates a SoftCal curve from 1-3 temperature/sensor points and a standard curve.

Inputs generated curve into the given curve number.

Args:

source_curve (Model224SoftCalSensorTypes):
The standard curve to use to generate the SoftCal curve from along with calibration points.

curve_number (int):
The curve number to save the generated curve to. Options are: 21 - 59.

serial_number (str):
Serial number of the user curve. Maximum of 10 characters.

calibration_point_1 (tuple):
Tuple of two floats in the form (temperature_value, sensor_value).

calibration_point_2 (tuple):
Tuple of two floats in the form (temperature_value, sensor_value). Optional Parameter.

calibration_point_3 (tuple):
Tuple of two floats in the form (temperature_value, sensor_value). Optional parameter.

`get_curve`(*curve*)

Returns a list of all the data points in a particular curve.

Args:

curve (int):
Specifies which curve to set.

Return:

data_points (list):
A list containing every point in the curve represented as a tuple (sensor_units: float, temp_value: float).

`set_curve`(*curve*, *data_points*)

Method to define a user curve using a list of data points.

Args:

curve (int):
Specifies which curve to set.

data_points (list):

A list containing every point in the curve represented as a tuple
(sensor_units: float, temp_value: float).

`get_relay_status`(*relay_channel*)

Returns whether the specified relay is On or Off.

Args:

relay_channel (int):
The relay channel to query. Options are: 1 or 2.

Returns:

(bool):
True if relay is on, False if relay is off.

`set_filter`(*input_channel*, *filter_enabled*, *number_of_points=8*, *filter_reset_threshold=10*)

Enables or disables a filter for the readings of the specified input channel.

Filter is a running average that smooths input readings exponentially.

Args:

input_channel (str):
The input to set or disable a filter for. Options are: A, B, C(1 - 5), D(1 - 5).

filter_enabled (bool):
Enables or disables a filter for the input channel. True for enabled, False for disabled.

number_of_points (int):
Specifies the number of points used for the filter. Inputting a larger number of points will slow down the instrument's response to changes in temperature. Options are: 2 - 64 Optional if disabling the filter function.

filter_reset_threshold (int):
Specifies the limit for restarting the filter, represented by a percent of the full scale reading. If raw reading differs from filtered value by more than this threshold, filter averaging resets. Options are: 1% - 10%. Optional if disabling the filter function.

`get_filter`(*input_channel*)

Retrieves information about the filter set on the specified input channel.

Args:

input_channel (str):
The input to query for filter information. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(dict):
{"filter_enabled": bool, "number_of_points": int, "filter_reset_threshold": int}

`configure_input`(*input_channel*, *settings*)

Configures a sensor for measurement input readings.

Args:

input_channel (str):
The input to configure the input for. Options are: A, B, C(1 - 5), D(1 - 5).

settings (Model224InputSensorSettings):
Object of the Model224InputSensorSettings containing information for sensor setup.

`disable_input`(*input_channel*)

Disables the selected input channel.

Args:

input_channel (str):
The input to disable. Options are: A, B, C(1 - 5), D(1 - 5).

`get_input_configuration`(*input_channel*)

Returns the configuration settings of the sensor at the specified input channel.

Args:

input_channel (str):
The input to query. Options are: A, B, C(1 - 5), D(1 - 5).

Returns:

(Model224InputSensorSettings):
Object of type Model224InputSensorSettings containing information about the sensor at the given input_channel.

`select_remote_interface`(*remote_interface*)

Selects the remote interface to use for communications.

Args:

remote_interface (Model224RemoteInterface):
Object of enum type Model224RemoteInterface, representing the type of interface used for communications.

`get_remote_interface`()

Returns the remote interface being used for communications.

Returns:

(Model224RemoteInterface):
Object of enum type Model224RemoteInterface representing the interface being used for communications.

`select_interface_mode`(*interface_mode*)

Selects the mode for the remote interface being used.

Args:

interface_mode (Model224InterfaceMode):
Object of enum type Model224InterfaceMode representing the desired communication mode.

`get_interface_mode`()

Returns the mode of the remote interface.

Returns:

(Model224InterfaceMode):
Object of enum type Model224InterfaceMode representing the communication mode.

`set_display_field_settings`(*field*, *input_channel*, *display_units*)

Configures a display field in custom display mode.

Args:

field (int):
Specifies which display field to configure. Options are: 1 - 8.

input_channel (Model224InputChannel):
Defines which input to display.

display_units (Model224DisplayFieldUnits):
Defines which units to display reading in.

`get_display_field_settings`(*field*)

Returns the settings of a single display field in custom display mode.

Args:

field (int):
Specifies the display field to query. Options are: 1 - 8.

Returns:

(dict):
{"input_channel": Model224InputChannel, "display_units": Model224DisplayFieldUnits}

`configure_display`(*display_mode*, *number_of_fields=0*)

Configures the display of the instrument.

Args:

display_mode (Model224DisplayMode):
Defines what mode to set the display in. Mode either defines which input to display, or sets up a custom display using display fields.

number_of_fields (Model224NumberOfFields):
Defines the number of display locations to display. Only valid if mode is set to CUSTOM.

`get_display_configuration`()

Returns the mode of the display.

If display mode is Custom, this method also returns the number of display fields in the custom display.

Returns:

(dict):
{"display_mode": DisplayMode, "number_of_fields": NumberOfFields}.

`turn_relay_on`(*relay_number*)

Turns the specified relay on.

Args:

relay_number (int):
The relay to turn on. Options are: 1 or 2.

`turn_relay_off`(*relay_number*)

Turns the specified relay off.

Args:

relay_number (int):
The relay to turn off. Options are: 1 or 2.

`set_relay_alarms`(*relay_number*, *activating_input_channel*, *alarm_relay_trigger_type*)

Sets a relay to turn on and off automatically based on the state of the alarm of the specified input channel.

Args:

relay_number (int):
The relay to configure. Options are: 1 or 2.

activating_input_channel (str):
Specifies which input alarm activates the relay when the relay is in alarm mode. Only applies if ALARM mode is chosen. Options are: A, B, C(1 - 5), D(1 - 5).

alarm_relay_trigger_type (Model224RelayControlAlarm):
Specifies the type of alarm that triggers the relay. Only applies if ALARM mode is chosen.

`get_relay_alarm_control_parameters`(*relay_number*)

Returns the relay alarm configuration for either of the two configurable relays.

Relay must be configured for alarm mode to retrieve parameters.

Args:

relay_number (int):
Specifies which relay to query. Options are: 1 or 2.

Return:

(dict):
{"activating_input_channel": str, "alarm_relay_trigger_type": Model224RelayControlAlarm}.

`get_relay_control_mode`(*relay_number*)

Returns the configured mode of the specified relay.

Args:

relay_number (int):
Specifies which relay to query. Options are: 1 or 2.

Returns:

(Model224RelayControlMode):
The configured mode of the relay, represented as an object of the enum type Model224RelayControlMode.

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 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 set-points 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. Optional parameter.

compensation (bool):
Defines if thermal input compensation is on or off. Not applicable for diode sensors. Defaults to false. Optional parameter.

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.**Model224Enums**

Class containing the enums relevant to the Model 224.

class InputSensorType(*value*)

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

INPUT_DISABLED = 0

DIODE = 1

PLATINUM_RTD = 2

NTC_RTD = 3

class InputSensorUnits(*value*)

Enumeration for the preferred units of an input sensor.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

class DiodeSensorRange(*value*)

Enumeration for the voltage range of a diode sensor.

RANGE_2_POINT_5_VOLTS = 0

RANGE_10_VOLTS = 1

class PlatinumRTDSensorResistanceRange(*value*)

Enumeration of the resistance range of a platinum RTD input sensor.

TEN_OHMS = 0

THIRTY_OHMS = 1

ONE_HUNDRED_OHMS = 2

THREE_HUNDRED_OHMS = 3

ONE_KILOHM = 4

THREE_KILOHMS = 5

TEN_KILOHMS = 6

class NTCRTDSensorResistanceRange(*value*)

Enumeration of the resistance range of a NTC RTD input sensor.

TEN_OHMS = 0

THIRTY_OHMS = 1

ONE_HUNDRED_OHMS = 2

THREE_HUNDRED_OHMS = 3

ONE_KILOHM = 4

THREE_KILOHMS = 5

TEN_KILOHMS = 6

THIRTY_KILOHMS = 7

ONE_HUNDRED_KILOHMS = 8

class InterfaceMode(*value*)

Enumeration for the mode of the remote interface.

LOCAL = 0

REMOTE = 1

REMOTE_LOCAL_LOCK = 2

class RemoteInterface(*value*)

Enumeration for the remote interface being used to communicate with the instrument.

USB = 0

ETHERNET = 1

IEEE_488 = 2

class DisplayFieldUnits(*value*)

Enumerated type defining how units are enumerated for settings and using Display Fields.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

MINIMUM_DATA = 4

MAXIMUM_DATA = 5

class InputChannel(*value*)

Enumerated type defining which input channels correspond to ints for setting and using Display Fields.

NO_INPUT = 0

INPUT_A = 1

INPUT_B = 2

INPUT_C = 3

INPUT_D1 = 4

INPUT_D2 = 5

INPUT_D3 = 6

INPUT_D4 = 7

INPUT_D5 = 8

INPUT_C2 = 9

INPUT_C3 = 10

INPUT_C4 = 11

INPUT_C5 = 12

class DisplayMode(*value*)

Enumeration defining what input or information is shown on the front panel display.

INPUT_A = 0

INPUT_B = 1

INPUT_C = 2

INPUT_D1 = 3

CUSTOM = 4

ALL_INPUTS = 5

INPUT_D2 = 6

INPUT_D3 = 7

INPUT_D4 = 8

INPUT_D5 = 9

INPUT_C2 = 10

INPUT_C3 = 11

INPUT_C4 = 12

INPUT_C5 = 13

class NumberOfFields(*value*)

Enumerated type specifying the number of display fields to configure in the Custom display mode.

LARGE_4 = 0

LARGE_8 = 1

LARGE_4_SMALL_8 = 2

SMALL_16 = 3

class RelayControlAlarm(*value*)

Enumeration of the setting determining which alarm(s) cause a relay to activate in alarm mode.

LOW_ALARM = 0

HIGH_ALARM = 1

BOTH_ALARMS = 2

class RelayControlMode(*value*)

Enumeration of the configured mode of a relay.

RELAY_OFF = 0

RELAY_ON = 1

ALARMS = 2

class CurveFormat(*value*)

Enumerations specify formats for temperature sensor curves.

MILLIVOLT_PER_KELVIN = 1

VOLTS_PER_KELVIN = 2

OHMS_PER_KELVIN = 3

LOG_OHMS_PER_KELVIN = 4

class CurveTemperatureCoefficients(*value*)

Enumerations specify positive/negative temperature sensor curve coefficients.

NEGATIVE = 1

POSITIVE = 2

class DiodeExcitationCurrent(*value*)

Enum type representing the different excitation currents available for a diode sensor.

TEN_MICRO_AMPS = 0

ONE_MILLI_AMP = 1

class SoftCalSensorTypes(*value*)

Enum type representing the standard curves used to generate a SoftCal curve.

The 3 standard curves each represent a different type of sensor that can be calibrated with a SoftCal curve.

DT_400 = 1

PT_100 = 6

PT_1000 = 7

Status register classes

lakeshore.model_224.**Model224StandardEventRegister**

alias of *StandardEventRegister*

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

Class object representing the standard event register.

bit_names = ['operation_complete', '', 'query_error', '', 'execution_error', 'command_error', '', 'power_on']

class lakeshore.model_224.**Model224ServiceRequestRegister**(*message_available*, *event_summary*, *operation_summary*)

Class object representing the Service Request Enable register.

bit_names = ['', '', '', '', 'message_available', 'event_summary', 'operation_summary']

class lakeshore.model_224.**Model224StatusByteRegister**(*message_available*, *event_summary*, *master_summary_status*, *operation_summary*)

Class object representing the status byte register.

bit_names = ['', '', '', '', 'message_available', 'event_summary', 'master_summary_statusoperation_summary']

class lakeshore.model_224.**Model224ReadingStatusRegister**(*invalid_reading*, *temperature_under_range*, *temperature_over_range*, *sensor_units_zero*, *sensor_units_over_range*)

Class object representing the reading status of an input.

While not a literal register, the return of an int representation of multiple booleans makes it convenient to represent this functionality as a register.

bit_names = ['invalid_reading', '', '', '', 'temperature_under_range', 'temperature_over_range', 'sensor_units_zero', 'sensor_units_over_range']

Model 240 Input Modules

The 240 Series Input Modules employ distributed PLC control for large scale cryogenic temperature monitoring.

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

Example Scripts

Model 240 Input Channel Setup Example

```
from lakeshore import Model240, Model240InputParameter
from time import sleep

# Connect to the first available Model 240 over USB
my_model_240 = Model240()

# Define the channel configuration for a sensor with a negative temperature coefficient, autorange disabled
# current reversal disabled, the channel enabled, and set to the 100 kOhm range
rtd_config = Model240InputParameter(my_model_240.SensorTypes.NTC_RTD, False, False, my_model_240.Units.SENSOR, True,
                                    my_model_240.InputRange.RANGE_NTCRTD_100_KIL_OHMS)
```

(continues on next page)

(continued from previous page)

```
# Apply the configuration to all channels
for channel in range(1, 9):
    my_model_240.set_input_parameter(channel, rtd_config)

sleep(1)
print("Reading from channel 5: {} ohms".format(my_model_240.get_sensor_reading(5)))
```

Model 240 Profibus Configuration Example

```
from lakeshore import Model240, Model240ProfiSlot

# Connect to the first available Model 240 over USB
my_model_240 = Model240()

# Print the instrument's current PROFIBUS connection status to the console
print("Profibus connection status: " + my_model_240.get_profibus_connection_status())

# Configure the number of PROFIBUS slots for the instrument to present to the bus as a␣
↪modular station
# Setting the number of PROFIBUS slots to 2
my_model_240.set_profibus_slot_count(2)

# Create the ProfiSlot class object by specifying which input to associate the
# slot with and what temperature units the data will be presented in
# Setting the input channel as 2 and temperature units to Celsius
my_profibus_slot = Model240ProfiSlot(2, my_model_240.Units.CELSIUS)

# Configure what data to be presented on the given PROFIBUS slot
# Profibus slot 1 will be associated with channel 2
my_model_240.set_profibus_slot_configuration(1, my_profibus_slot)

# Print the PROFIBUS address
# An address of 126 indicates that it is not configured and it can then be set by a␣
↪PROFIBUS master
print("Profibus 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*)

Returns the current status indicator of the specified channel

The integer returned represents the sum of the bit weighting of the channel status flag bits. A "000" response indicates a valid reading is present.

Args:

channel (int):
Specifies which channel to query (1-8).

Returns:

bit_status (dict):
Dictionary containing the current status indicator.

`get_sensor_units_channel_reading`(*channel*)

Returns the sensor units value of channel being queried.

Args:

channel (int):
Specifies which channel to query (1-8).

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 auto-ranging 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 auto-range 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.**Model240Enums**

Class containing the enums relevant to the Model 240.

class Units(*value*)

Enumerations that specify temperature units.

KELVIN = 1

CELSIUS = 2

SENSOR = 3

FAHRENHEIT = 4

class CurveFormat(*value*)

Enumerations that specify temperature sensor curve format units.

VOLTS_PER_KELVIN = 2

OHMS_PER_KELVIN = 3

LOG_OHMS_PER_KELVIN = 4

class Coefficients(*value*)

Enumerations that specify a positive or negative coefficient.

NEGATIVE = 1

POSITIVE = 2

class SensorTypes(*value*)

Enumerations specify types of temperature sensors.

DIODE = 1

PLATINUM_RTD = 2

NTC_RTD = 3

class BrightnessLevel(*value*)

Enumerations for the screen brightness levels.

OFF = 0

LOW = 25

MED_LOW = 50

MED_HIGH = 75

HIGH = 100

class TemperatureCoefficient(*value*)

Enumerations specify positive/negative temperature sensor curve coefficients.

NEGATIVE = 1

POSITIVE = 2

class InputRange(*value*)

Enumerations to specify the input range when auto-range is off.

RANGE_DIODE = 0

RANGE_PTRTD_1_KIL_OHMS = 0

RANGE_NTCRTD_10_OHMS = 0

RANGE_NTCRTD_30_OHMS = 1

RANGE_NTCRTD_100_OHMS = 2

RANGE_NTCRTD_1_KIL_OHMS = 4

```
RANGE_NTCRTD_3_KIL_OHMS = 5

RANGE_NTCRTD_10_KIL_OHMS = 6

RANGE_NTCRTD_30_KIL_OHMS = 7

RANGE_NTCRTD_100_KIL_OHMS = 8
```

2.4.6 Sources

Model 121 Programmable DC Current Source

The Lake Shore Model 121 provides low-noise, stable current.

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

Example Scripts

Setting a current

```
from lakeshore import Model121

# Connect to Model 121 over USB
current_source = Model121()

# Set current source to 150 nA and start
current_source.set_current(150e-9)

# Query and print the current source level
print(current_source.get_current())
```

Instrument methods

class lakeshore.model_121.**Model121**(*serial_number=None*, *com_port=None*, *baud_rate=57600*, *data_bits=7*, *stop_bits=1*, *parity='O'*, *flow_control=False*, *handshaking=False*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing the Lake Shore Model 121 programmable DC current source.

command(*command_string*)

Sends a command to the instrument.

Args:

command_string (str): A serial command.

set_current(*current*)

Set and start outputting a specific current from the instrument.

Switches to the user current range and applies the specified current. Current can be set between 100nA and 100mA. The sign of the current determines the polarity of the output.

Args:

current (float): New current value as a floating point number. The minimum value is 100e-9 A and the maximum
is 100e-3 A. Positive or negative. Up to three significant digits.

`get_current()`

Returns the present user current setting in Amps.

Returns:
float: User current value in Amps.

`enable_current()`

Enables current source output.

`disable_current()`

Disables current source output.

`reset_instrument()`

Sets instrument parameters to power-up settings.

`set_display_brightness(`*brightness_level*`)`

Sets the display contrast for the front panel seven-segment display.

A higher number makes the display brighter. The default setting is 8. The display can be turned off by setting the brightness to 0.

Args:
brightness_level (int): The display brightness (0-15).

`get_display_brightness()`

Gets the display contrast for the front panel seven-segment display.

A higher number makes the display brighter. The default setting is 8. Brightness level 0 means the display is turned off.

Returns:
int: The display brightness (0-15).

`get_compliance_limit_status()`

Returns the voltage compliance status of the current source output.

Returns:
bool: False = normal operation. True = in compliance limit.

`set_factory_defaults()`

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

`lock_front_panel()`

Locks the front panel keypad.

`unlock_front_panel()`

Unlocks the front panel keypad.

`get_front_panel_lock_status()`

Returns if the front panel keypad is locked or not.

Returns:
bool: True = Locked, False = Unlocked.

`set_power_up_enable`(*state*)

Specifies whether the output remains on or shuts off after power cycle.

Args:
state (bool): True = Enabled, False = Disabled.

`save_current_state`()

Saves the present range, polarity, and user current value.

This saved state will be loaded on future power ups.

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

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

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

Establish a serial USB connection.

`disconnect_tcp`()

Disconnect the TCP connection.

`disconnect_usb`()

Disconnect the USB connection.

`query`(*query_string*)

Send a query to the instrument and return the response.

Args:

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

Returns:
The instrument query response as a string.

`write`(*command_string*)

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

Model 155 Precision Current and Voltage Source

The Lake Shore 155 is a low noise, high precision current and voltage source.

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

Example Scripts

Below are a few example scripts for the Model 155 that use the Lake Shore Python driver.

Model 155 Sweep Example

```
from lakeshore import PrecisionSource

# The purpose of this script is to sweep frequency, amplitude, and offset of an output␣
↪signal using
# a Lake Shore AC/DC 155 Precision Source

# Create a new instance of the Lake Shore 155 Precision Source.
# It will connect to the first instrument it finds via serial USB
my_source = PrecisionSource()

# Define a custom list of frequencies to sweep through
frequency_sweep_list = ['1', '10', '100', '250', '500', '750', '1000', '2000', '5000',
↪'10000']

# Sweep frequency in voltage mode. Wait 1 second at each step
my_source.sweep_voltage(1, frequency_values=frequency_sweep_list)


# Creates a list of whole number offset values between -5V and 5V.
offset_sweep_list = range(-5, 6)
# Creates a list of amplitudes between 0 and 5V incrementing by 100mV
amplitude_sweep_list = [value/10 for value in range(0, 51)]
# Creates a list of frequencies starting with 0.1 Hz and increasing by powers of ten up␣
↪to 10 kHz
frequency_sweep_list = [10**exponent for exponent in range(-1, 5)]

# Use the lists defined above to sweep across all combinations of the lists.
# For each combination, wait 10ms before moving to the next one.
# Note that the dwell time will be limited by the response time of the serial␣
↪communication.
my_source.sweep_voltage(0.01,
                        offset_values=offset_sweep_list,
                        amplitude_values=amplitude_sweep_list,
                        frequency_values=frequency_sweep_list)
```

Instrument class methods

class lakeshore.model_155.**PrecisionSource**(*serial_number=None*, *com_port=None*, *baud_rate=115200*, *flow_control=False*, *timeout=2.0*, *ip_address=None*, *tcp_port=7777*, ***kwargs*)

A class object representing a Lake Shore 155 precision I/V source.

sweep_voltage(*dwell_time*, *offset_values=None*, *amplitude_values=None*, *frequency_values=None*)

Sweep source output voltage parameters based on list arguments.

Args:

dwell_time (float):
The length of time in seconds to wait at each parameter combination. Note that the update rate will be limited by the SCPI communication response time. The response time is usually on the order of 10-30 milliseconds.

offset_values (list):
DC offset values in volts to sweep over.

amplitude_values (list):
Peak to peak values in volts to sweep over.

frequency_values (list):
Frequency values in Hertz to sweep over.

`sweep_current`(*dwell_time*, *offset_values=None*, *amplitude_values=None*, *frequency_values=None*)

Sweep the source output current parameters based on list arguments

Args:

dwell_time (float):
The length of time in seconds to wait at each parameter combination. Note that the update rate will be limited by the SCPI communication response time. The response time is usually on the order of 10-30 milliseconds.

offset_values (list):
DC offset values in volts to sweep over.

amplitude_values (list):
Peak to peak values in volts to sweep over.

frequency_values (list):
Frequency values in Hertz to sweep over.

`enable_output`()

Turns on the source output.

`disable_output`()

Turns off the source output.

`set_output`(*output_on*)

Configure the source output on or off.

Args:

output_on (bool):
Turns the source output on when True, off when False.

`route_terminals`(*output_connections_location='REAR'*)

Configures whether the source output is routed through the front or rear connections.

Args:

output_connections_location (str):
Valid options are: "REAR" (Output is routed out the rear connections), and "FRONT" (Output is routed out the front connections).

`output_sine_current`(*amplitude*, *frequency*, *offset=0.0*, *phase=0.0*)

Configures and enables the source output to be a sine wave current source.

Args:

amplitude (float):
The peak current amplitude value in amps.

frequency (float):
The source frequency value in hertz.

offset (float):
The DC offset current in amps.

phase (float):
Shifts the phase of the output relative to the reference out. Must be between -180 and 180 degrees.

`output_sine_voltage`(*amplitude*, *frequency*, *offset=0.0*, *phase=0.0*)

Configures and enables the source output to be a sine wave voltage source.

Args:

amplitude (float):
The peak voltage amplitude value in volts.

frequency (float):
The source frequency value in hertz.

offset (float):
The DC offset voltage in volts.

phase (float):
Shifts the phase of the output relative to the reference out. Must be between -180 and 180 degrees.

`output_dc_current`(*current_level*)

Configures the source output to be a DC current source.

Args:

current_level (float):
The output current level in amps.

`output_dc_voltage`(*voltage_level*)

Configures the source output to be a DC current source.

Args:

voltage_level (float):
The output voltage level in volts.

`get_output_settings`()

Returns a dictionary of the output settings.

`enable_autorange`()

Enables the instrument to automatically select the best range for the given output parameters.

`disable_autorange`()

Enables the instrument to automatically select the best range for the given output parameters.

`set_current_range`(*current_range='100E-3'*)

Manually sets the current range when autorange is disabled.

Args:

current_range (str):
The range in amps. Valid ranges are: “100E-3”, “10E-3”, “1E-3”, “100E-6”, “10E-6”, and “1E-6”.

`set_voltage_range`(*voltage_range='10'*)

Manually sets the voltage range when autorange is disabled.

Args:

voltage_range (str):
The range in volts. Valid ranges are: “100”, “10”, “1”, “0.1”, and “0.01”.

`set_current_limit`(*current_limit*)

Sets the highest settable current output value when in current mode.

Args:

current_limit (float):
The maximum settable current in amps. Must be between 0 and 100 milli-amps.

`set_voltage_limit`(*voltage_limit*)

Sets the highest settable voltage output value when in voltage mode.

Args:

voltage_limit (float):
The maximum settable voltage in amps. Must be between 0 and 100 volts.

`set_current_mode_voltage_protection`(*max_voltage*)

Sets the maximum voltage level permitted by the instrument when sourcing current.

Args:

max_voltage (float):
The maximum permissible voltage. Must be between 1 and 100 volts.

`set_voltage_mode_current_protection`(*max_current*)

Sets the maximum current level permitted by the instrument when sourcing voltage.

Args:

max_current (float):
The maximum permissible voltage. Must be between 1 and 100 volts.

`enable_ac_high_voltage_compliance`()

Configures the current mode compliance voltage to be 100V in AC output modes.

`disable_ac_high_voltage_compliance`()

Configures the current mode compliance voltage to be 10V in AC output modes.

`command`(**commands*, *check_errors=True*)

Send an SCPI command or multiple commands to the instrument.

Args:

commands (str):
Any number of SCPI commands.

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

connect_tcp(*ip_address*, *tcp_port*, *timeout*)

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

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

Establish a serial USB connection.

disconnect_tcp()

Disconnect the TCP connection.

disconnect_usb()

Disconnect the USB connection.

factory_reset()

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

get_operation_event_enable_mask()

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

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

get_operation_events()

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

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

get_present_operation_status()

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

get_present_questionable_status()

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

get_questionable_event_enable_mask()

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

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

get_questionable_events()

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

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

get_service_request_enable_mask()

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

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

get_standard_event_enable_mask()

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

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

get_standard_events()

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

get_status_byte()

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

`modify_operation_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_questionable_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_service_request_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`modify_standard_event_register_mask`(*bit_name*, *value*)

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

Args:

bit_name (str):
The name of the bit to modify.

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

`query`(**queries*, *check_errors=True*)

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

Args:

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

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

Returns:
The instrument query response as a string.

`reset_measurement_settings`()

Resets measurement settings to their default values.

`reset_status_register_masks()`

Resets status register masks to preset values.

`set_operation_event_enable_mask(register_mask)`

Configures the values of the operation event enable register bits.

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

Args:

register_mask ([Instrument]OperationRegister):
An instrument specific OperationRegister class object with all bits configured true or false.

`set_questionable_event_enable_mask(register_mask)`

Configures the values of the questionable event enable register bits.

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

Args:

register_mask ([Instrument]QuestionableRegister):
An instrument specific QuestionableRegister class object with all bits configured true or false.

`set_service_request_enable_mask(register_mask)`

Configures values of the service request enable register bits.

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

Args:

register_mask (StatusByteRegister):
A StatusByteRegister class object with all bits configured true or false.

`set_standard_event_enable_mask(register_mask)`

Configures values of the standard event enable register bits.

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

Args:

register_mask (StandardEventRegister):
A StandardEventRegister class object with all bits configured true or false.

`write(command_string)`

Alias of command. Send a command to the instrument.

Args:

command_string (str):
A serial command.

PYTHON MODULE INDEX

l

INDEX

Symbols

A

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

T

U

V

W

Z