Lake Shore Python Driver Documentation

Release 1.5.1

Lake Shore Cryotronics, Inc.

Jun 30, 2021

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 1

Supported Products

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

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

CHAPTER 2

Table of contents

2.1 Installation

2.1.1 Python version

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

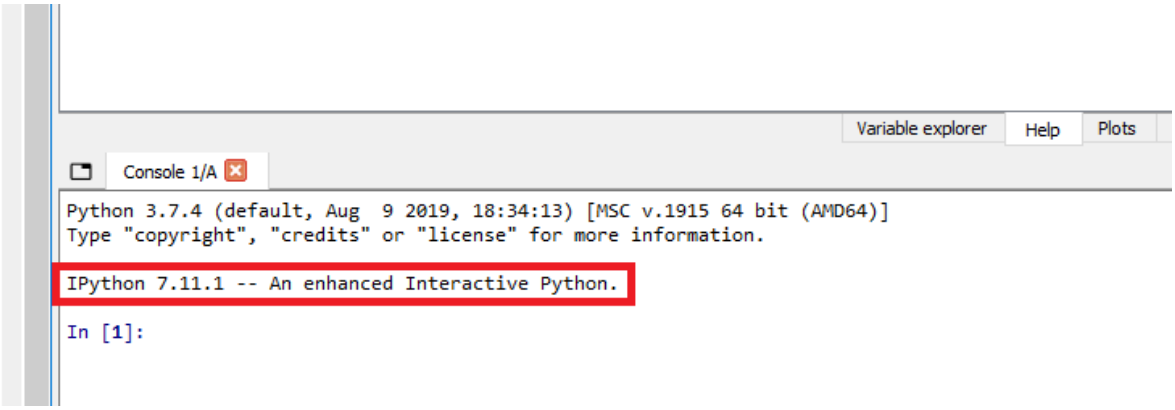
2.1.2 Install the Lake Shore Python driver

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

```
pip install lakeshore
```

2.1.3 Installing the driver through Spyder

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



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

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

Back in the Spyder console, type:

```
pip install lakeshore
```

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

2.2 Getting Started

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

2.2.1 A simple example

```
from lakeshore import Model155

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

2.2.2 Making Connections

Connecting to a specific instrument

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

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

```
from lakeshore import Teslameter

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

or the COM port it is connected to:

```
from lakeshore import FastHall

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

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

```
from lakeshore import Model372

my_instrument = Model372(9600)
```

Connecting over TCP

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

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

```
from lakeshore import Model155

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

2.2.3 Commands and queries

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

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

```
from lakeshore import Model155

my_instrument = Model155()

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

2.2.4 SCPI commands and queries

Grouping multiple commands & queries

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

```
from lakeshore import Teslameter

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

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

Checking for SCPI errors

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

```
from lakeshore import Teslameter

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

2.3 Advanced

2.3.1 Thread Safety

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

2.3.2 Logging

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

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

```
import logging
import sys

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

2.3.3 Status Registers

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

Reading a register

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

```
from lakeshore import Teslameter

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

will return the following:

```
{'no_probe': False, 'overload': False, 'ranging': False, 'ramp_done': False, 'no_data_
↪on_breakout_adapter': False}
```

Modifying a register mask

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

```
from lakeshore import PrecisionSource

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

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

```
from lakeshore import PrecisionSource, PrecisionSourceQuestionableRegister

my_instrument = PrecisionSource()
register_mask = PrecisionSourceQuestionableRegister(voltage_source_in_current_
↪limit=True,
                                    current_source_in_voltage_compliance=True,
                                    calibration_error=False,
                                    inter_processor_communication_error=False)

my_instrument.set_questionable_event_enable_mask(register_mask)
```

2.3.4 Instrument initialization options

Keep communication errors on initialization

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

```
from lakeshore import Teslameter

my_instrument = Teslameter(clear_errors_on_init=False)
```

2.3.5 Python 2 compatibility

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

2.4 Supported Instruments

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

2.4.1 Magnetics Instruments

F41 & F71 Teslameters

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

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

Example Scripts

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

Streaming F41/F71 teslameter data to a CSV file

```
from lakeshore import Teslameter

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

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

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

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

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

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

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

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

Classes and methods

Instrument class methods

Register classes

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

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

2.4.2 Magnet System Power Supplies

Model 643 Electromagnet Power Supply

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

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

Instrument methods

Model 648 Electromagnet Power Supply

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

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

Instrument methods

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
```

(continues on next page)

(continued from previous page)

```
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)

    # 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()
```

Classes and methods

Instrument class methods

Instrument classes

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

M81 Synchronous Source Measure System

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

Example Scripts

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

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

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

# Connect to instrument via USB
my_M81 = SSMSystem()

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

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

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

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

# Enable the source output
balanced_current_source.enable()

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

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

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

Classes and methods

SSMS instrument methods

Source Module methods

Measure Module methods

Instrument registers

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

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

Below are a few example scripts for the Model 335 that use the Lake Shore Python driver.

Setting a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

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

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

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

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature␣
↪points. The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(Model224SoftCalSensorTypes.DT_400, 30,
↪"SN123", (276, 10),
                                               (300, 5), (310, 2))
```

(continues on next page)

(continued from previous page)

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

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

Recording data with the Model 335

```
from lakeshore.model_335 import *

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

# Create a new instance of the input sensor settings class
sensor_settings = Model335InputSensorSettings(Model335InputSensorType.DIODE, True,
↪False,
                                              Model335InputSensorUnits.KELVIN,
                                              Model335DiodeRange.TWO_POINT_FIVE_VOLTS)

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

# Set diode excitation current on channel A to 10uA
my_model_335.set_diode_excitation_current("A", Model335DiodeCurrent.TEN_MICROAMPS)

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

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

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

Setting up autotune on the Model 335

```
from lakeshore import Model335
from lakeshore.model_335 import Model335DisplaySetup, Model335HeaterResistance, \
    Model335HeaterOutputDisplay, Model335HeaterRange, Model335AutoTuneMode,␣
↪Model335HeaterError
from time import sleep

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

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

# Configure the display mode
my_model_335.set_display_setup(Model335DisplaySetup.TWO_INPUT_A)

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

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

# Turn on the heater by setting the range
my_model_335.set_heater_range(1, Model335HeaterRange.HIGH)

# Check to see if there are any heater related errors
heater_error = my_model_335.get_heater_status(1)
if heater_error is not Model335HeaterError.NO_ERROR:
    raise Exception(heater_error.name)

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

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

# Initiate autotune in PI mode, initial conditions will not be met if the system is␣
↪not
# maintaining a temperature within 5 K of the setpoint
my_model_335.set_autotune(1, Model335AutoTuneMode.P_I)

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

(continues on next page)

(continued from previous page)

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

Classes and methods

Instrument class methods

Settings classes

This section outlines the classes used to interact with methods which return or accept an argument of a class object, specific to the Lake Shore model 335.

Enumeration objects

This section describes the Enum type objects that have been created to represent various settings of the model 335 that are used as an argument or return 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.

Register classes

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

Model 336 Cryogenic Temperature Controller

The Model 336 measures and controls cryogenic temperature environments.

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

Example Scripts

Below are a few example scripts for the Model 336 that use the Lake Shore Python driver.

Using calibration curves with a temperature instrument

```
import matplotlib.pyplot as plt
from lakeshore import Model224
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

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

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
```

(continues on next page)

(continued from previous page)

```
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

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

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature␣
↪points. The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(Model224SoftCalSensorTypes.DT_400, 30,
↪"SN123", (276, 10),
                                               (300, 5), (310, 2))

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

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

Setting up heater outputs on the Model 336

```
from lakeshore import Model336, Model336HeaterResistance, Model336HeaterOutputUnits,␣
↪Model336InputChannel, \
                      Model336InputSensorUnits, Model336Polarity, Model336HeaterRange,
↪ Model336HeaterVoltageRange, \
                      Model336HeaterOutputMode

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

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

# Configure heater output 1 with a 50 ohm load, 0.75 amp max current, and screen␣
↪display mode to power
my_model_336.set_heater_setup(1, Model336HeaterResistance.HEATER_50_OHM, 0.75,␣
↪Model336HeaterOutputUnits.POWER)
```

(continues on next page)

(continued from previous page)

```
# Configure analog heater output 3 to monitor sensor channel A, a high and low value␣
↪of 3.65 and 1.02 kelvin
# respectively as a unipolar output
my_model_336.set_monitor_output_heater(3, Model336InputChannel.CHANNEL_A,␣
↪Model336InputSensorUnits.KELVIN, 3.65, 1.02,
                                       Model336Polarity.UNIPOLAR)

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

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

# Set a control setpoint for outputs 1 and 3 to 1.5 kelvin
my_model_336.set_control_setpoint(1, 1.5)
my_model_336.set_control_setpoint(3, 2.5)

# Turn the heaters on by setting the heater range
my_model_336.set_heater_range(1, Model336HeaterRange.MEDIUM)
my_model_336.set_heater_range(3, Model336HeaterVoltageRange.VOLTAGE_ON)

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

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

Classes and methods

Instrument class methods

Instrument 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.

Enumeration Objects

This page describes the Enum type objects that have been created to represent various settings of the model 336 that are used as an argument or return 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.

Register Classes

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

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

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

Below are a few example scripts for the Model 372 that use the Lake Shore Python driver.

Setting a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

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

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

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

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature␣
↪points. The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(Model224SoftCalSensorTypes.DT_400, 30,
↪"SN123", (276, 10),
                                               (300, 5), (310, 2))
```

(continues on next page)

(continued from previous page)

```
# 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
from lakeshore import Model372SensorExcitationMode,␣
↪Model372MeasurementInputCurrentRange, \
    Model372AutoRangeMode, Model372InputSensorUnits,␣
↪Model372MeasurementInputResistance, Model372InputSetupSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure a sensor
# Create Model372InputSetupSettings object with current excitation mode, 31.6 uA␣
↪excitation current, autoranging on
# (tracking current), current source not shunted, preferred units of Kelvin, and a␣
↪resistance range of 20.0 kOhms
sensor_settings = Model372InputSetupSettings(Model372SensorExcitationMode.CURRENT,
                                             Model372MeasurementInputCurrentRange.
↪RANGE_31_POINT_6_MICRO_AMPS,
                                             Model372AutoRangeMode.CURRENT, False,␣
↪Model372InputSensorUnits.KELVIN,
                                             Model372MeasurementInputResistance.RANGE_
↪20_KIL_OHMS)

# Pass settings into method along with desired input channel
my_model_372.configure_input(1, sensor_settings)

# Get all readings (temperature, resistance, excitation power, quadrature) from sensor
sensor_1_readings = my_model_372.get_all_input_readings(1)

# Record readings to a file
file = open("372_sensor_1_data.csv", "w")
file.write("Header information\n")
# Call readings using the keys of the returned dictionary
file.write("Temperature Reading," + str(sensor_1_readings['kelvin']) + "\n")
file.write("Resistance Reading," + str(sensor_1_readings['resistance']) + "\n")
file.write("Excitation Power," + str(sensor_1_readings['power']) + "\n")
file.write("Imaginary Part of Resistance," + str(sensor_1_readings['quadrature']) +
↪"\n")
file.close()
```

Setting up a control loop with the model 372

```
from lakeshore import Model372
from lakeshore import Model372HeaterOutputSettings, Model372OutputMode,␣
↪Model372InputChannel, Model372ControlLoopZoneSettings

# Include baud rate when initializing instrument
my_model_372 = Model372(57600)

# Configure output for zone mode, controlled by control input, with power up enabled␣
↪filter enabled and a reading
# delay of 10 seconds
# Note; it's assumed that the control input is enabled and configured
heater_settings = Model372HeaterOutputSettings(Model372OutputMode.ZONE,␣
↪Model372InputChannel.CONTROL, True, True, 10)
my_model_372.configure_heater(1, heater_settings)

# Configure a relay for Warmup Heater Zone
my_model_372.set_relay_for_warmup_heater_control_zone(1)

# Set up control loop with an upper bound of 15K, a gain of 50, an integral value of␣
↪5000, a derivative of 2000, and a
# manual output of 50%. Range is set to true, setpoint ramp rate is set to 10 seconds,
↪ and relay 1 is configured for
# the zone and relay 2 is not configured
control_loop_settings = Model372ControlLoopZoneSettings(15, 50.0, 5000, 2000, 50,␣
↪True, 10, True, False)
# Set control loop on output 1 (Warmup Heater) in zone 4
my_model_372.set_control_loop_parameters(1, 4, control_loop_settings)

# Create a setpoint for 5 K
my_model_372.set_setpoint_kelvin(1, 5.0)
# Enable ramping to setpoint for output 1 at a rate of 10 Kelvin/minute
my_model_372.set_setpoint_ramp_parameter(1, True, 10)
```

Classes and methods

Instrument class methods

Instrument settings classes and registers

This section describes the classes used throughout the 372 methods to interact with instrument settings and other methods that use objects and classes.

Enumeration objects

This section describes the Enum type objects that have been created to represent 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.

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

Below is an example script for the Model 224 that uses the Lake Shore Python driver.

Configuring the model 224 with a temperature curve

```
import matplotlib.pyplot as plt
from lakeshore import Model224
from lakeshore.model_224 import Model224CurveHeader, Model224CurveFormat,␣
↪Model224CurveTemperatureCoefficients, \
    Model224SoftCalSensorTypes

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

# Configure a curve by first setting its header parameters. First, set the name and␣
↪serial number of the curve.
# Then, select the units used to set map the sensor units to temperature units. Set a␣
↪temperature limit, and
# then specify whether the coefficients are positive or negative.
curve_header_25 = Model224CurveHeader("My_Curve", "ABC123", Model224CurveFormat.VOLTS_
↪PER_KELVIN, 300.0,
                                      Model224CurveTemperatureCoefficients.POSITIVE)
myinstrument.set_curve_header(25, curve_header_25)

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

# You can create a softcal curve by inputting 1-3 calibration sensor/temperature␣
↪points. The instrument generates
# a new curve using your entered data points and the selected standard curve
myinstrument.generate_and_apply_soft_cal_curve(Model224SoftCalSensorTypes.DT_400, 30,
↪"SN123", (276, 10),
                                               (300, 5), (310, 2))

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

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

(continues on next page)

(continued from previous page)

```
myinstrument.delete_curve(25)
```

Classes and methods

Instrument class methods

Settings classes

Status registers

Enumeration objects

This section describes the Enum type objects that have been created to represent 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.

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

Below are a few example scripts for the Model 335 that use the Lake Shore Python driver.

Model 240 Input Channel Setup Example

```
from lakeshore import Model240
from lakeshore.model_240 import Model240InputParameter, Model240SensorTypes,␣
↪Model240Units, Model240InputRange
from time import sleep

# Connect to the first available Model 240 over USB
my_model_240 = Model240()

# Define the channel configuration for a sensor with a negative temperature␣
↪coefficient, autorange disabled
# current reversal disabled, the channel enabled, and set to the 100 kOhm range
rtd_config = Model240InputParameter(Model240SensorTypes.NTC_RTD, False, False,␣
↪Model240Units.SENSOR, True,
                                    Model240InputRange.RANGE_NTCRTD_100_KIL_OHMS)

# Apply the configuration to all channels
for channel in range(1, 9):
    my_model_240.set_input_parameter(channel, rtd_config)
```

(continues on next page)

(continued from previous page)

```
sleep(1)
print("Reading from channel 5: {} ohms".format(my_model_240.get_sensor_reading(5)))
```

Model 240 Profibus Configuration Example

```
from lakeshore import Model240, Model240Units, Model240ProfiSlot

# Connect to the first available Model 240 over USB
my_model_240 = Model240()

# Print the instrument's current PROFIBUS connection status to the console
print("Profibus connection status: " + my_model_240.get_profibus_connection_status())

# Configure the number of PROFIBUS slots for the instrument to present to the bus as␣
↪a modular station
# Setting the number of PROFIBUS slots to 2
my_model_240.set_profibus_slot_count(2)

# Create the ProfiSlot class object by specifying which input to associate the
# slot with and what temperature units the data will be presented in
# Setting the input channel as 2 and temperature units to Celsius
my_profibus_slot = Model240ProfiSlot(2, Model240Units.CELSIUS)

# Configure what data to be presented on the given PROFIBUS slot
# Profibus slot 1 will be associated with channel 2
my_model_240.set_profibus_slot_configuration(1, my_profibus_slot)

# Print the PROFIBUS address
# An address of 126 indicates that it is not configured and it can then be set by a␣
↪PROFIBUS master
print("Profibus adress: " + my_model_240.get_profibus_address())

# Set the desired address as 123
my_model_240.set_profibus_address("123")

# Acquiring settings that were configured above
print(my_model_240.get_profibus_slot_count())
slot_1_config = my_model_240.get_profibus_slot_configuration(1)
print(slot_1_config.slot_channel)
print(slot_1_config.slot_units)
```

Classes and methods

Instrument class methods

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.

Enumeration objects

This section describes the Enum type objects that have been created to represent 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.

2.4.6 Sources

Model 121 Programmable DC Current Source

The Lake Shore Model 121 provides low-noise, stable current.

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

Instrument methods

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
```

(continues on next page)

(continued from previous page)

```
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)
```

Classes and methods

Instrument class methods

Python Module Index

l

Index

L