Slave Documentation

Release 0.4.0.dev

Marco Halder

November 26, 2014

This is the documentation of the Slave library, a micro framework designed to simplify instrument communication and control. It is divided into three parts, a quick *overview*, the *user guide* and of course the *api reference*.

CHAPTER 1

Overview

Slave provides an intuitive way of creating instrument api's, inspired by object relational mappers.

```
from slave.iec60488 import IEC60488, PowerOn
from slave.core import Command
from slave.types import Integer, Enum

class Device(IEC60488, PowerOn):
    """An iec60488 conforming device api with additional commands."""
    def __init__(self, transport):
        super(Device, self).__init__(transport)
        # A custom command
        self.my_command = Command(
            'QRY?', # query message header
            'WRT',  # command message header
            # response and command data type
            [Integer, Enum('first', 'second')]
        )
```

Commands mimic instance attributes. Read access queries the device, parses and converts the response and finally returns it. Write access parses and converts the arguments and sends them to the device. This leads to very intuitive interfaces.

Several device drivers are already implemented, and many more are under development. A short usage example is given below

```
#!/usr/bin/env python
import time

from slave.srs import SR830
from slave.transport import Visa


lockin = SR830(Visa('GPIB::08'))
lockin.frequency = 22.08
lockin.amplitude = 5.0
lockin.reserve = 'high'
for i in xrange(60):
    print lockin.x
    time.sleep(1)
```

For a complete list of built-in device drivers, see *Built-in Device Drivers*.

CHAPTER 2

User Guide

2.1 Installing Slave

Installation is quite easy. The latest stable version is available on the python package index or github. It can be installed with the package managers *pip* or *easy_install* via:

```
pip install slave
```

or:

```
easy_install slave
```

2.1.1 Installing from source

To install it from source, download and extract it and execute:

```
python setup.py install
```

2.1.2 Installing the latest development version

To work with the latest version of slave, clone the github repository and install it in development mode:

```
git clone git://github.com/p3trus/slave.git
cd slave
python setup.py develop
```

2.2 slave Changelog

Here you can see the full list of changes between each slave release.

2.2.1 Version 0.4.0

Note: This release breaks backwards compatibility.

- Changed the notation of a *connection* to *transport*. This name is more widely used, see e.g. twisted, or asyncio.

- Removed the direct usage of the transport object in device driver methods. Now almost all methods use the *InstrumentBase._query()* and *InstrumentBase._write()* methods.

  There are only a few exceptions:

  - *slave.srs.sr830.SR830.trace()*
  - *slave.srsr.sr830.SR830.snap()*
  - *slave.srs.sr850.MarkList.active()*

- Renamed *slave.core.InstrumentBase._cfg* attribute to *_protocol* and in all dependant cases.

Changes to the *slave.core* module:

- The *InstrumentBase.transport'attribute was renamed to '_transport* to be more consistent. This avoids shadowing of possible commands and show the intent that in general the transport should not be used directly.

Changes to the *slave.lakeshore.ls340* module:

- The *slave.lakeshore.ls340.Curve.delete()* method now raises a *RuntimeError* when called on a read-only curve.
- The *slave.lakeshore.ls340.LS340._factory_defaults()* method does not take the boolean *confirm* argument anymore. The trailing underscore should be warning enough that you know what you're doing.

Changes to the *slave.lakeshore.ls370* module:

- Implemented the missing relay commands *LS370.low_relay* and *LS370.high_relay*.

2.3 Quickstart

Using slave is easy. The following example shows how device drivers are used. We are going to implement a short measurement script, which initializes a Stanford Research SR830 lock-in amplifier and performs a measurement.

The first step is to initialize a transport to the lock-in amplifier. Here we are using pyvisa to establish a GPIB (General Purpose Interface Bus) transport with the device at primary address 8.

```
from slave.transport import Visa
transport = Visa('GPIB::08')
```

Slave does not communicate directly with the device. It uses an object referred to as *transport object* for the low level communication (see *The transport object* for a detailed explanation).

In the next step, we construct a `SR830` instance and inject the pyvisa transport.

```
from slave.srs import SR830

lockin = SR830(transport)
```

This creates a fully functional, high level interface to the lock-in amplifier. Before we start the actual measurement, we're going to configure the lock-in.

```
lockin.frequency = 22.08  # Set the internal frequency generator to 22.08 Hz
lockin.amplitude = 5.0    # Use an amplitude of 5 V
lockin.reserve = 'high'
```

And finally measure 60 times, waiting one second between each measurement, and print the result.

```
import time

for i in xrange(60):
    print lockin.x
    time.sleep(1)
```

Putting it all together, we get a small 13 line script, capable of performing a complete measurement.

```
#!/usr/bin/env python
import time

from slave.srs import SR830
from slave.transport import Visa


lockin = SR830(Visa('GPIB::08'))
lockin.frequency = 22.08
lockin.amplitude = 5.0
lockin.reserve = 'high'
for i in xrange(60):
    print lockin.x
    time.sleep(1)
```

2.4 The transport object

2.4.1 The interface

Slave does not communicate with a device directly. It uses an object referred to as *transport object*. This abstraction makes it possible to use the same device drivers with different types of transports (e.g. rs232, GPIB, usb, serial, ...). As long as an object conforms to the *transport interface* it can be used with slave.

The interface is quite simple. Just two methods are required. They are defined as follows:

`Transport.`**`ask`**`(`*command*`)`
    Takes a command string and returns a string response.

`Transport.`**`write`**`(`*command*`)`
    Takes a command string.

2.4.2 Adapters

To enhance the compatibility with different communication libraries, the `slave.transport` module implements several adapter classes. These are

Class	Description	Notes
`GpibDevice`	A transport object wrapping the Linux-gpib driver.	Linux only
`TCPIPDevice`	A tiny wrapper around a socket transport.	
`UsbTmcDevice`	A transport object, wrapping a usbtmc file descriptor.	Linux only

2.4.3 Simulating a transport

Slave has a rudimentary simulation mode. Just use the `SimulatedTransport` instead of an actual transport. An example is shown below:

```
from slave.transport import SimulatedTransport
from slave.srs import SR380

lockin = SR830(SimulatedTransport())
print lockin.x  # prints a random float
```

A simple algorithm is used to create the responses.

For read-only commands, the response is randomly created with each query, since these typically represent measured values. For read and writable commands, the response is created just once and cached afterwards. Repeated queries will return the same result unless a write was issued.

2.4.4 Implementing custom transports

2.5 Logging

Slave makes use of python's standard logging module. It is quite useful for development of new device drivers and diagnosing of communication errors.

You can use it in the following way:

```
import logging

logging.basicConfig(filename='log.txt', filemode='w', level=logging.DEBUG)

# Now use slave ...
```

2.6 Built-in Device Drivers

Slave ships with several completely implemented device drivers.

2.6.1 Lock-in Amplifiers

Manufacturer	Model	Class
Lakeshore	LS370 AC Resistance Bridge	`slave.lakeshore.ls370.LS370`
Signal Recovery	SR7225	`slave.signal_recovery.sr7225.SR7225`
Signal Recovery	SR7230	`slave.signal_recovery.sr7230.SR7230`
Stanford Research	SR830	`slave.srs.sr830.SR830`
Stanford Research	SR850	`slave.srs.sr850.SR850`

2.6.2 Temperature Controllers

Manufacturer	Model	Class
Lakeshore	LS340	`slave.lakeshore.LS340`

2.6.3 Magnet Power Supplies

Manufacturer	Model	Class
CryoMagnetics Inc.	Magnet Power Supply Model 4G	`slave.cryomagnetics.MPS4G`

2.6.4 Cryostats

Manufacturer	Model	Class
Quantum Design	PPMS Model 6000	`slave.quantum_design.PPMS`

2.7 Implementing Custom Device Drivers

Implementing custom device drivers is straight forward. The following sections will guide you through several use cases. We will implement a driver for an imaginary device, extending it's interface step-by-step, showing more and more functionality and tricks.

Note: When developing new device drivers, it is useful to enable logging. See *Logging* for more information.

2.7.1 First Steps

Let's assume we've got a simple device supporting the following commands:

- 'ENABLED <on/off>' – Enables/disables the control loop of the device. *<on/off>* is either 0 or 1.
- 'ENABLED?' – returns <on/off>

A possible implementation could look like this:

```
from slave.core import Command, InstrumentBase
from slave.types import Boolean

class Device(InstrumentBase):
    def __init__(self, transport):
        super(Device, self).__init__(transport)
        self.enabled = Command('ENABLED?', 'ENABLED', Boolean())
```

Now let's try it. We're using a `SimulatedTransport` here (see *Simulating a transport* for a detailed explanation):

```
>>> from slave.core import SimulatedTransport
>>> device = Device(SimulatedTransport())
>>> device.enabled = False
>>> device.enabled
False
```

It looks as if an instance variable with the name 'enabled' and a value of *False* was created. But this is not the case. We can check it with the following line:

```
>>> type(device.__dict__['enabled'])
<class 'slave.core.Command'>
```

The assignment did not overwrite the Command attribute. Instead, the `InstrumentBase` base class forwarded the *False* to the `write()` method of the `Command`. The `write()` method then created the command message **'ENABLED 0'**, using the `Boolean` type to convert the False and passed it to the transport's `write()` method. Likewise the read call was forwarded to the `Command`'s query method.

2.7.2 The IEC60488-2 standard

The IEC 60488-2 describes a standard digital interface for programmable instrumentation. It is used by devices connected via the IEEE 488.1 bus, commonly known as GPIB. It is an adoption of the *IEEE std. 488.2-1992* standard.

The IEC 60488-2 requires the existence of several commands which are logically grouped.

Reporting Commands

- **CLS* - Clears the data status structure [1] .
- **ESE* - Write the event status enable register [2] .
- **ESE?* - Query the event status enable register [3] .
- **ESR?* - Query the standard event status register [4] .
- **SRE* - Write the status enable register [5] .
- **SRE?* - Query the status enable register [6] .
- **STB* - Query the status register [7] .

Internal operation commands

- **IDN?* - Identification query [8] .
- **RST* - Perform a device reset [9] .
- **TST?* - Perform internal self-test [10] .

Synchronization commands

- **OPC* - Set operation complete flag high [11] .
- **OPC?* - Query operation complete flag [12] .
- **WAI* - Wait to continue [13] .

To ease development, these are implemented in the `IEC60488` base class. To implement a IEC 60488-2 compliant device driver, you only have to inherit from it and implement the device specific commands, e.g:

```
from slave.core import Command
from slave.iec60488 import IEC60488

class CustomDevice(IEC60488):
    pass
```

This is everything you need to do to implement the required IEC 60488-2 command interface.

Optional Commands

Despite the required commands, there are several optional command groups defined. The standard requires that if one command is used, it's complete group must be implemented. These are

Power on common commands

[1] IEC 60488-2:2004(E) section 10.3
[2] IEC 60488-2:2004(E) section 10.10
[3] IEC 60488-2:2004(E) section 10.11
[4] IEC 60488-2:2004(E) section 10.12
[5] IEC 60488-2:2004(E) section 10.34
[6] IEC 60488-2:2004(E) section 10.35
[7] IEC 60488-2:2004(E) section 10.36
[8] IEC 60488-2:2004(E) section 10.14
[9] IEC 60488-2:2004(E) section 10.32
[10] IEC 60488-2:2004(E) section 10.38
[11] IEC 60488-2:2004(E) section 10.18
[12] IEC 60488-2:2004(E) section 10.19
[13] IEC 60488-2:2004(E) section 10.39

- **PSC* - Set the power-on status clear bit [14] .
- **PSC?* - Query the power-on status clear bit [15] .

Parallel poll common commands

- **IST?* - Query the individual status message bit [16] .
- **PRE* - Set the parallel poll enable register [17] .
- **PRE?* - Query the parallel poll enable register [18] .

Resource description common commands

- **RDT* - Store the resource description in the device [19] .
- **RDT?* - Query the stored resource description [20] .

Protected user data commands

- **PUD* - Store protected user data in the device [21] .
- **PUD?* - Query the protected user data [22] .

Calibration command

- **CAL?* - Perform internal self calibration [23] .

Trigger command

- **TRG* - Execute trigger command [24] .

Trigger macro commands

- **DDT* - Define device trigger [25] .
- **DDT?* - Define device trigger query [26] .

Macro Commands

- **DMC* - Define device trigger [27] .
- **EMC* - Define device trigger query [28] .
- **EMC?* - Define device trigger [29] .
- **GMC?* - Define device trigger query [30] .
- **LMC?* - Define device trigger [31] .

[14] IEC 60488-2:2004(E) section 10.25
[15] IEC 60488-2:2004(E) section 10.26
[16] IEC 60488-2:2004(E) section 10.15
[17] IEC 60488-2:2004(E) section 10.23
[18] IEC 60488-2:2004(E) section 10.24
[19] IEC 60488-2:2004(E) section 10.30
[20] IEC 60488-2:2004(E) section 10.31
[21] IEC 60488-2:2004(E) section 10.27
[22] IEC 60488-2:2004(E) section 10.28
[23] IEC 60488-2:2004(E) section 10.2
[24] IEC 60488-2:2004(E) section 10.37
[25] IEC 60488-2:2004(E) section 10.4
[26] IEC 60488-2:2004(E) section 10.5
[27] IEC 60488-2:2004(E) section 10.7
[28] IEC 60488-2:2004(E) section 10.8
[29] IEC 60488-2:2004(E) section 10.9
[30] IEC 60488-2:2004(E) section 10.13
[31] IEC 60488-2:2004(E) section 10.16

- **PMC* - Define device trigger query [32] .

Option Identification command

- **OPT?* - Option identification query [33] .

Stored settings commands

- **RCL* - Restore device settings from local memory [34] .
- **SAV* - Store current settings of the device in local memory [35] .

Learn command

- **LRN?* - Learn device setup query [36] .

System configuration commands

- **AAD* - Accept address command [37] .
- **DLF* - Disable listener function command [38] .

Passing control command

- **PCB* - Pass control back [39] .

The optional command groups are implemented as Mix-in classes. A device supporting required IEC 60488-2 commands as well as the optional Power-on commands is implemented as follows:

```python
from slave.core import Command
from slave.iec60488 import IEC60488, PowerOn

class CustomDevice(IEC60488, PowerOn):
    pass
```

2.8 Asynchronous IO

Warning: This is currently not working!

Slave has a built-in compatibility layer for the tornado framework. It is currently in an early state and only socket transports are supported. Nevertheless, it is already usable. The following examples will show how to make use of it.

2.8.1 A simple asynchronous poller

In this example we will implement a simple, basically useless, asynchronous poller to explain the concept. It simply prints out the polled value. We will extend this example to implement a monitor with a web interface.

[32] IEC 60488-2:2004(E) section 10.22
[33] IEC 60488-2:2004(E) section 10.20
[34] IEC 60488-2:2004(E) section 10.29
[35] IEC 60488-2:2004(E) section 10.33
[36] IEC 60488-2:2004(E) section 10.17
[37] IEC 60488-2:2004(E) section 10.1
[38] IEC 60488-2:2004(E) section 10.6
[39] IEC 60488-2:2004(E) section 10.21

```python
from tornado.ioloop import IOLoop, PeriodicCallback
from tornado.gen import coroutine

import slave.async
# Monkey patch slave to use the asynchronous implementation
slave.async.patch()

# Due to the call to `patch()`, the driver and the transport automatically
# use the asynchronous implementation.
from slave.sr7230 import SR7230
from slave.transport import TCPIPDevice

lockin1 = SR7230(TCPIPDevice('192.168.178.11:80000'))

def show(fn):
    @coroutine
    def print_fn():
        value = yield fn()
        print value
    return print_fn

ioloop = tornado.ioloop.IOLoop.Instance()
poller = [
    # poll the x voltage every 2 seconds, the sensitivity every 5.
    PeriodicCallback(show(lambda: lockin1.x), 2000),
    PeriodicCallback(show(lambda: lockin1.sensitivity), 5000)
]
for p in poller:
    ioloop.add_callback(p.start)

ioloop.start()
```

2.9 Usage Examples

2.9.1 Simple Measurement

This examples shows a simple measurement script, using a Stanford Research Systems LockIn amplifier and is discussed in more detail in the *Quickstart* section.

```python
#!/usr/bin/env python
import time

from slave.srs import SR830
from slave.transport import Visa


lockin = SR830(Visa('GPIB::08'))
lockin.frequency = 22.08
lockin.amplitude = 5.0
lockin.reserve = 'high'
for i in xrange(60):
    print lockin.x
    time.sleep(1)
```

2.9.2 Magnetotransport Measurement

In this example, we assume a sample with standard four terminal wiring is placed inside a Quantum Desing PPMS. We're using our own Lock-In amlifier to measure the resistance as a function of temperature.

```python
"""This example shows a measurement routine for a custom magnetotransport setup
in the [P]hysical [P]roperties [M]easurement [S]ystem PPMS Model 6000.

"""
import datetime

import visa

from slave.quantum_design import PPMS
from slave.sr830 import SR830
from slave.transport import Visa # pyvisa wrapper
from slave.misc import Measurement

# Connect to the lockin amplifier and the ppms
lockin = SR830(Visa('GPIB::10'))
ppms = PPMS(Visa('GPIB::15'))

try:
    # Configure the lockin amplifier
    lockin.frequency = 22.08  # Use a detection frequency of 22.08 Hz
    lockin.amplitude = 5.0    # and an amplitude of 5 V.
    lockin.reserve = 'low'
    lockin.time_constant = 3

    # Set the ppms temperature to 10 K, cooling with a rate of 20 K per min.
    ppms.set_temperature(10, 20, wait_for_stability=True)
    # Now sweep slowly to avoid temperature instabilities.
    ppms.set_temperature(1.2, 0.5, wait_for_stability=True)
    # Set a magnetic field of 1 T at a rate of 150 mT per second and set the magnet
    # in persistent mode.
    #
    # Note: The PPMS uses Oersted instead of Tesla. 1 Oe = 0.1 mT.
    ppms.set_field(10000, 150, mode='persistent', wait_for_stability=True)

    # Set the appropriate gain. (We're assuming the measured voltage decreases
    # with increasing temperature.
    lockin.auto_gain()

    # Define the measurement parameters
    parameters = [
        lambda: datetime.datetime.now(), # Get timestamp
        lambda: lockin.x,
        lambda: lockin.y,
        lambda: ppms.temperature,
    ]
    # Finally start the measurement, using the Measurement helper class as a
    # context manager (This automatically closes the measurement file).
    with Measurement('1.2K-300K_1T.dat', measure=parameters) as measurement:
        ppms.scan_temperature(measurement, 300, 0.5)
except Exception, e:
    # Catch possible errors and print a message.
    print 'An error occured:', e
finally:
    # Finally put the ppms in standby mode.
```

```
ppms.shutdown()
```

CHAPTER 3

API Reference

This part of the documentation covers the complete api of the slave library.

3.1 API

This part covers the complete api documentation of the slave library.

3.1.1 `slave` Package

3.1.2 `transport` Module

Several implementations of the transport api.

The `slave.transport` module implements the lowest level abstraction layer in the slave library. The transport is responsible for sending and receiving raw bytes. It interfaces with the hardware, but has no knowledge of the meaning of the bytes transfered.

The `Transport` class defines a common api used in higher abstraction layers. Subclasses of `slave.Transport` must implement *__read__()* and *__write__()* methods.

The following transports are already available:

- `Serial` - A wrapper of the pyserial library
- `Socket` - A wrapper around the standard socket library.
- `LinuxGpib` - A wrapper of the linux-gpib library
- `visa()` - A wrapper of the pyvisa library. (Supports pyvisa 1.4 - 1.5).

class `slave.transport.`**`LinuxGpib`**(*primary=0*, *secondary=0*, *board=0*, *timeout=13*, *send_eoi=1*, *eos=0*)

Bases: `slave.transport.Transport`

A linuxgpib adapter.

`close`()

Closes the gpib transport.

`trigger`()

Triggers the device.

The trigger method sens a GET(group execute trigger) command byte to the device.

class `slave.transport.`**`Serial`**`(`**args*, ***kw*`)`
Bases: `slave.transport.Transport`

A pyserial adapter.

class `slave.transport.`**`SimulatedTransport`**
Bases: `future.types.newobject.newobject`

The SimulatedTransport.

The SimulatedTransport does not have any functionallity. It servers as a sentinel value for the Command class to enable the simulation mode.

class `slave.transport.`**`Socket`**`(`*address*, *alwaysopen=True*, **args*, ***kw*`)`
Bases: `slave.transport.Transport`

A slave compatible adapter for pythons socket.socket class.

Parameters

- **address** – The socket address a tuple of host string and port. E.g.

```
from slave.signal_recovery import SR7230
from slave.transport import Socket

lockin = SR7230(Socket(address=('192.168.178.1', 50000)))
```

- **alwaysopen** – A boolean flag deciding wether the socket should be opened and closed for each use as a contextmanager or should be opened just once and kept open until closed explicitely. E.g.:

```
from slave.transport import Socket

transport = Socket(address=('192.168.178.1', 50000), alwaysopen=False)
with transport:
    # connection is created
    transport.write(b'*IDN?')
    response = transport.read_until(b'\n')
    # connection is closed again.

transport = Socket(address=('192.168.178.1', 50000), alwaysopen=True)
# connection is already opened.
with transport:
    transport.write(b'*IDN?')
    response = transport.read_until(b'\n')
    # connection is kept open.
```

`close``()`

`open``()`

class `slave.transport.`**`Transport`**`(`*max_bytes=1024*, *lock=None*`)`
Bases: `future.types.newobject.newobject`

A utility class to write and read data.

The `Transport` base class defines a common interface used by the *slave* library. Transports are intended to be used as context managers.

Subclasses must implement __*read*__ and __*write*__.

`read_bytes``(`*num_bytes*`)`
Reads at most *num_bytes*.

read_exactly(*num_bytes*)
Reads exactly *num_bytes*

read_until(*delimiter*)
Reads until the delimiter is found.

write(*data*)

class slave.transport.**Visa_1_4**(*instrument*)
Bases: slave.transport.Transport

A pyvisa 1.4 adapter.

class slave.transport.**Visa_1_5**(*instrument*)
Bases: slave.transport.Transport

A pyvisa 1.5 adapter.

slave.transport.**visa**(**args*, ***kw*)
A pyvisa adapter factory function.

3.1.3 `protocol` Module

class slave.protocol.**IEC60488**(*msg_prefix=u’‘*, *msg_header_sep=u’ ‘*, *msg_data_sep=u’, ‘*, *msg_term=u’n’*, *resp_prefix=u’‘*, *resp_header_sep=u’‘*, *resp_data_sep=u’, ‘*, *resp_term=u’n’*, *encoding=u’ascii’*)
Bases: slave.protocol.Protocol

Implementation of IEC60488 protocol.

This class implements the IEC-60488 protocol, formerly known as IEEE 488.2.

Parameters

- **msg_prefix** – A string which will be prepended to the generated command string.
- **msg_header_sep** – A string separating the message header from the message data.
- **msg_data_sep** – A string separating consecutive data blocks.
- **msg_term** – A string terminating the message.
- **resp_prefix** – A string each response is expected to begin with.
- **resp_header_sep** – The expected separator of the response header and the response data.
- **resp_data_sep** – The expected data separator of the response message.
- **resp_term** – The response message terminator.
- **stb_callback** – For each read and write operation, a status byte is received. If a callback function is given, it will be called with the status byte.

create_message(*header*, **data*)

parse_response(*response*, *header=None*)
Parses the response message.

The following graph shows the structure of response messages.

```
                                         +----------+
                                      +--+ data sep +<-+
                                      |  +----------+  |
                                      |                |
+--------+         +------------+     |    +------+    |
  +-->| header +------->+ header sep +---+--->+ data +----+----+
  |   +--------+        +------------+        +------+         |
  |                                                            |
--+                                         +----------+       +-->
  |                                      +--+ data sep +<-+    |
  |                                      |  +----------+  |    |
  |                                      |                |    |
  |                                      |    +------+    |    |
  +--------------------------------------+--->+ data +----+----+
                                              +------+
```

query (*transport*, *header*, **data*)

write (*transport*, *header*, **data*)

class slave.protocol.**OxfordIsobus** (*address=None*, *echo=True*, *msg_term=u'r'*, *resp_term=u'r'*, *encoding=u'ascii'*)

Bases: slave.protocol.Protocol

Implements the oxford isobus protocol.

Parameters

- **address** – The isobus address.
- **echo** – Enables/Disables device command echoing.
- **msg_term** – The message terminator.
- **resp_term** – The response terminator.
- **encoding** – The message and response encoding.

Oxford Isobus messages messages are created in the following manner, where *HEADER* is a single char:

```
+--------+    +------+
+ HEADER +--->+ DATA +
+--------+    +------+
```

Isobus allows to connect multiple devices on a serial line. To address a specific device a control character '@' followed by an integer address is used:

```
+---+    +---------+    +--------+    +------+
+ @ +--->+ ADDRESS +--->+ HEADER +--->+ DATA +
+---+    +---------+    +--------+    +------+
```

On success, the device answeres with the header followed by data if requested. If no echo response is desired, the '$' control char must be prepended to the command message. This is useful if a single command must sent to all connected devices at once.

On error, the device answeres with a '?' char followed by the command message. E.g the error response to a message *@7R10* would be *?R10*.

create_message (*header*, **data*)

parse_response (*response*, *header*)

query (*transport*, *header*, **data*)

write (*transport*, *header*, **data*)

class slave.protocol.**Protocol**

Bases: future.types.newobject.newobject

Abstract protocol base class.

query(*transport*, **args*, ***kw*)

write(*transport*, **args*, ***kw*)

class slave.protocol.**SignalRecovery**(*msg_prefix=u'‘*, *msg_header_sep=u' ‘*, *msg_data_sep=u' ‘*, *msg_term=u'x00'*, *resp_prefix=u'‘*, *resp_header_sep=u'‘*, *resp_data_sep=u'*, *‘*, *resp_term=u'x00'*, *stb_callback=None*, *olb_callback=None*, *encoding=u'ascii'*)

Bases: slave.protocol.IEC60488

An implementation of the signal recovery network protocol.

Modern signal recovery devices are fitted with a ethernet port. This class implements the protocol used by these devices. Command messages are built with the following algorithm.

```
                                      +----------+
                                   +--+ data sep +<-+
                                   |  +----------+  |
                                   |                |
    +--------+    +------------+   |    +------+    |    +----------+
--->+ header +--->+ header sep +---+--->+ data +----+--->+ msg term +-->
    +--------+    +------------+        +------+         +----------+
```

Each command, query or write, generates a response. It is terminated with a null character '0' followed by the status byte and the overload byte.

Parameters

- **msg_prefix** – A string which will be prepended to the generated command string.
- **msg_header_sep** – A string separating the message header from the message data.
- **msg_data_sep** – A string separating consecutive data blocks.
- **msg_term** – A string terminating the message.
- **resp_prefix** – A string each response is expected to begin with.
- **resp_header_sep** – The expected separator of the response header and the response data.
- **resp_data_sep** – The expected data separator of the response message.
- **resp_term** – The response message terminator.
- **stb_callback** – For each read and write operation, a status byte is received. If a callback function is given, it will be called with the status byte.
- **olb_callback** – For each read and write operation, a overload status byte is received. If a callback function is given, it will be called with the overload byte.
- **encoding** – The encoding used to convert the message string to bytes and vice versa.

E.g.:

```
>>>from slave.protocol import SignalRecovery
>>>from slave.transport import Socket

>>>transport = Socket(('192.168.178.1', 5900))
>>>protocol = SignalRecovery()
>>>print protocol.query(transport, '*IDN?')
```

call_byte_handler(*status_byte*, *overload_byte*)

query(*transport*, *header*, **data*)

query_bytes (*transport*, *num_bytes*, *header*, **data*)

Queries for binary data

Parameters

- **transport** – A transport object.
- **num_bytes** – The exact number of data bytes expected.
- **header** – The message header.
- **data** – Optional data.

Returns The raw unparsed data bytearray.

write (*transport*, *header*, **data*)

3.1.4 `core` Module

The core module contains several helper classes to ease instrument control.

Implementing an instrument interface is pretty straight forward. A simple implementation might look like:

```
from slave.core import InstrumentBase, Command
from slave.types import Integer


class MyInstrument(InstrumentBase):
    def __init__(self, transport):
        super(MyInstrument, self).__init__(transport)
        # A simple query and writeable command, which takes and writes an
        # Integer.
        self.my_cmd = Command('QRY?', 'WRT', Integer)
        # A query and writeable command that converts a string parameter to
        # int and vice versa.
        self.my_cmd2 = Command('QRY2?', 'WRT2', Enum('first', 'second'))
```

class `slave.core.`**Command** (*query=None*, *write=None*, *type_=None*, *protocol=None*)

Bases: `object`

Represents an instrument command.

The Command class handles the communication with the instrument. It converts the user input into the appropriate command string and sends it to the instrument via the transport object. For example:

```
# a read and writeable command
cmd1 = Command('STRING?', 'STRING', String)

# a readonly command returning a tuple of two strings
cmd2 = Command(('STRING?', [String, String]))

# a writeonly command
cmd3 = Command(write=('STRING', String))
```

Parameters

- **query** – A string representing the *query program header*, e.g. *'*IDN?'*. To allow customisation of the queriing a 2-tuple or 3-tuple value with the following meaning is also possible.
  - (<query header>, <response data type>)
  - (<query header>, <response data type>, <query data type>)

The types have the same requirements as the type parameter. If they are

- **write** – A string representing the *command program header*, e.g. *'*CLS'*. To allow for customization of the writing a 2-tuple value with the following requirements is valid as well.

  – (<command header>, <response data type>)

  The types have the same requirements as the type parameter.

- **protocol** – When a protocol (an object implementing the `slave.protocol.Protocol` interface) is given, `query()` and `write()` methods ignore it's protocol argument and use it instead.

`query`(*transport*, *protocol*, **data*)
Generates and sends a query message unit.

Parameters

- **transport** – An object implementing the *.Transport* interface. It is used by the protocol to send the message and receive the response.
- **protocol** – An object implementing the *.Protocol* interface.
- **data** – The program data.

Raises AttributeError if the command is not queryable.

`simulate_query`(*data*)

`simulate_write`(*data*)

`write`(*transport*, *protocol*, **data*)
Generates and sends a command message unit.

Parameters

- **transport** – An object implementing the *.Transport* interface. It is used by the protocol to send the message.
- **protocol** – An object implementing the *.Protocol* interface.
- **data** – The program data.

Raises AttributeError if the command is not writable.

class `slave.core.`**`CommandSequence`**(*transport*, *protocol*, *iterable*)
Bases: `slave.misc.ForwardSequence`

A sequence forwarding item access to the query and write methods.

class `slave.core.`**`InstrumentBase`**(*transport*, *protocol=None*, **args*, ***kw*)
Bases: `object`

Base class of all instruments.

The InstrumentBase class applies some *magic* to simplify the Command interaction. Read access on `Command` attributes is redirected to the `Command.query`, write access to the `Command.write` member function.

Parameters

- **transport** – The transport object.
- **protocol** – The protocol object. If no protocol is given, a `IEC60488` protocol is used as default.

3.1.5 `cryomagnetics` Module

class `slave.cryomagnetics.mps4g.`**`MPS4G`**(*transport*, *shims=None*, *channel=None*)
Bases: `slave.iec60488.IEC60488`

Represents the Cryomagnetics, inc. 4G Magnet Power Supply.

Parameters

- **transport** – A transport object.
- **channel** – This parameter is used to set the MPS4G in single channel mode. Valid entries are *None*, *1* and *2*.

Variables

- **channel** – The selected channel.
- **error** – The error response mode of the usb interface.
- **current** – The magnet current.Queriing returns a value, unit tuple. While setting the current, the unit is omited. The value must be supplied in the configured units (ampere, kilo gauss).
- **output_current** – The power supply output current.
- **lower_limit** – The lower current limit. Queriing returns a value, unit tuple. While setting the lower current limit, the unit is omited. The value must be supplied in the configured units (ampere, kilo gauss).
- **mode** – The selected operation mode, either *'Shim'* or *'Manual'*.
- **name** – The name of the currently selected coil. The length of the name is in the range of 0 to 16 characters.
- **switch_heater** – The state of the persistent switch heater. If *True* the heater is switched on and off otherwise.
- **upper_limit** – The upper current limit. Queriing returns a value, unit tuple. While setting the upper current limit, the unit is omited. The value must be supplied in the configured units (ampere, kilo gauss).
- **unit** – The unit used for all input and display operations. Must be either *'A'* or *'G'* meaning Ampere or Gauss.
- **voltage_limit** – The output voltage limit. Must be in the range of 0.00 to 10.00.
- **magnet_voltage** – The magnet voltage in the range -10.00 to 10.00.
- **magnet_voltage** – The output voltage in the range -12.80 to 12.80.
- **standard_event_status** – The standard event status register.
- **standard_event_status_enable** – The standard event status enable register.
- **id** – The identification, represented by the following tuple *(<manufacturer>, <model>, <serial>, <firmware>, <build>)*
- **operation_completed** – The operation complete bit.
- **status** – The status register.
- **service_request_enable** – The service request enable register.
- **sweep_status** – A string representing the current sweep status.

Warning: Due to a bug in firmware version 1.25, a semicolon must be appended to. the end of the commands *'LLIM'* and *'ULIM'*. This is done automatically. Writing the name crashes the MPS4G software. A restart does not fix the problem. You need to load the factory defaults.

Note: If something bad happens and the MPS4G isn't reacting, you can load the factory defaults via the simulation mode. To enter it press *SHIFT* and *5* on the front panel at startup.

`disable_shims`()
Disables all shims.

`enable_shims`()
Enables all shims.

`local`()
Sets the front panel in local mode.

`locked`()
Sets the front panel in locked remote mode.

`quench_reset`()
Resets the quench condition.

`remote`()
Sets the front panel in remote mode.

`sweep`(*mode*, *speed=None*)
Starts the output current sweep.

Parameters

- **mode** – The sweep mode. Valid entries are *'UP'*, *'DOWN'*, *'PAUSE'`or `'ZERO'*. If in shim mode, *'LIMIT'* is valid as well.
- **speed** – The sweeping speed. Valid entries are *'FAST'*, *'SLOW'* or *None*.

class `slave.cryomagnetics.mps4g.`**`Range`**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

Represents a MPS4G current range.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The current range index. Valid values are 0, 1, 2, 3, 4.

Variables

- **limit** – The upper limit of the current range.
- **rate** – The sweep rate of this current range.

`slave.cryomagnetics.mps4g.`**`SHIMS`** **= [u'Z', u'Z2', u'Z3', u'Z4', u'X', u'Y', u'ZX', u'ZY', u'C2', u'S2', u'Z2X', u'Z2Y**
A list with all valid shim identifiers.

class `slave.cryomagnetics.mps4g.`**`Shim`**(*transport*, *protocol*, *shim*)
Bases: `slave.core.InstrumentBase`

Represents a Shim option of the 4GMPS.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **shim** – The identifier of the shim.

Variables

- **limit** – The current limit of the shim.
- **status** – Represents the shim status, *True* if it's enabled, *False* otherwise.
- **current** – The magnet current of the shim. Queriing returns a value, unit tuple. While setting the current, the unit is omited. The value must be supplied in the configured units (ampere, kilo gauss).

disable()
Disables the shim.

select()
Selects the shim as the current active shim.

class slave.cryomagnetics.mps4g.**UnitFloat**(*min=None*, *max=None*, **args*, ***kw*)
Bases: slave.types.Float

Represents a floating point type. If a unit is present in the string representation, it will get stripped.

3.1.6 iec60488 Module

The iec60488 module implements a IEC 60488-2:2004(E) compliant interface.

The minimal required interface is implemented by the IEC60488 class. Optional command groups a provided by mixin classes. They should not be used on their own.

Usage:

```
from slave.IEC60488 import IEC60488, PowerOn

class CustomInstrument(IEC60488, PowerOn):
    '''A custom instrument compliant with the IEC 60488-2:2004(E),
    supporting the optional PowerOn commands.
    '''
    def __init__(self, transport):
        super(CustomInstrument, self).__init__(transport)
        # Implement custom commands.
```

class slave.iec60488.**Calibration**(**args*, ***kw*)
Bases: object

A mixin class, implementing the optional calibration command.

Variables **protected_user_data** – The protected user data. This is information unique to the device, such as calibration date, usage time, environmental conditions and inventory control numbers.

Note: This is a mixin class designed to work with the IEC60488 class.

The IEC 60488-2:2004(E) defines the following optional calibration command:

- **CAL?* - See IEC 60488-2:2004(E) section 10.2

calibrate()
Performs a internal self-calibration.

Returns An integer in the range -32767 to + 32767 representing the result. A value of zero indicates that the calibration completed without errors.

class `slave.iec60488.`**`IEC60488`**(*transport*, *protocol=None*, *esb=None*, *stb=None*, **args*, ***kw*)

Bases: `slave.core.InstrumentBase`

The IEC60488 class implements a IEC 60488-2:2004(E) compliant base class.

Parameters

- **transport** – A transport object.
- **esb** – A dictionary mapping the 8 bit standard event status register. Integers in the range 0 to 7 are valid keys. If present they replace the default values.
- **stb** – A dictionary mapping the 8 bit status byte. Integers in the range 0 to 7 are valid keys. If present they replace the default values.

Variables

- **event_status** – A dictionary representing the 8 bit event status register.
- **event_status_enable** – A dictionary representing the 8 bit event status enable register.
- **status** – A dictonary representing the 8 bit status byte.
- **status_enable** – A dictionary representing the status enable register.
- **operation_complete** – The operation complete flag.
- **identification** – The device identification represented by the following tuple *(<manufacturer>, <model>, <serial number>, <firmware level>)*.

A IEC 60488-2:2004(E) compliant interface must implement the following status reporting commands:

- **CLS* - See IEC 60488-2:2004(E) section 10.3
- **ESE* - See IEC 60488-2:2004(E) section 10.10
- **ESE?* - See IEC 60488-2:2004(E) section 10.11
- **ESR* - See IEC 60488-2:2004(E) section 10.12
- **SRE* - See IEC 60488-2:2004(E) section 10.34
- **SRE?* - See IEC 60488-2:2004(E) section 10.35
- **STB?* - See IEC 60488-2:2004(E) section 10.36

In addition, the following internal operation common commands are required:

- **IDN?* - See IEC 60488-2:2004(E) section 10.14
- **RST* - See IEC 60488-2:2004(E) section 10.32
- **TST?* - See IEC 60488-2:2004(E) section 10.38

Furthermore the following synchronisation commands are required:

- **OPC* - See IEC 60488-2:2004(E) section 10.18
- **OPC?* - See IEC 60488-2:2004(E) section 10.19
- **WAI* - See IEC 60488-2:2004(E) section 10.39

`clear`()

Clears the status data structure.

complete_operation()
Sets the operation complete bit high of the event status byte.

reset()
Performs a device reset.

test()
Performs a internal self-test and returns an integer in the range -32767 to + 32767.

wait_to_continue()
Prevents the device from executing any further commands or queries until the no operation flag is *True*.

Note: In devices implementing only sequential commands, the no-operation flag is always True.

class slave.iec60488.**Learn**(**args*, ***kw*)
Bases: object

A mixin class, implementing the optional learn command.

The IEC 60488-2:2004(E) defines the following optional learn command:

- **LRN?* - See IEC 60488-2:2004(E) section 10.17

learn()
Executes the learn command.

Returns A string containing a sequence of *response message units*. These can be used as *program message units* to recover the state of the device at the time this command was executed.

class slave.iec60488.**Macro**(**args*, ***kw*)
Bases: object

A mixin class, implementing the optional macro commands.

Variables **macro_commands_enabled** – Enables or disables the expansion of macros.

The IEC 60488-2:2004(E) defines the following optional macro commands:

- **DMC* - See IEC 60488-2:2004(E) section 10.7
- **EMC* - See IEC 60488-2:2004(E) section 10.8
- **EMC?* - See IEC 60488-2:2004(E) section 10.9
- **GMC?* - See IEC 60488-2:2004(E) section 10.13
- **LMC?* - See IEC 60488-2:2004(E) section 10.16
- **PMC* - See IEC 60488-2:2004(E) section 10.22

define_macro(*macro*)
Executes the define macro command.

Parameters **macro** – A macro string, e.g. *“‘SETUP1”,#221VOLT 14.5;CURLIM 2E-3’*

Note: The macro string is not validated.

disable_macro_commands()
Disables all macro commands.

enable_macro_commands()
Enables all macro commands.

get_macro(*label*)
Returns the macro.

Parameters label – The label of the requested macro.

macro_labels()
Returns the currently defined macro labels.

purge_macros()
Deletes all previously defined macros.

class slave.iec60488.**ObjectIdentification**(**args*, ***kw*)
Bases: object

A mixin class, implementing the optional object identification command.

Variables object_identification – Identifies reportable device options.

The IEC 60488-2:2004(E) defines the following optional object identification command:

•**OPT?* - See IEC 60488-2:2004(E) section 10.20

class slave.iec60488.**ParallelPoll**(*ppr=None*, **args*, ***kw*)
Bases: object

A mixin class, implementing the optional parallel poll common commands.

Parameters ppr – A dictionary mapping the 8-16 bit wide parallel poll register. Integers in the range 8 to 15 are valid keys. If present they replace the default values.

Variables

- **individual_status** – Represents the state of the IEEE 488.1 "ist" local message in the device.
- **parallel_poll_enable** – A dictionary representing the 16 bit parallel poll enable register.

Note: This is a mixin class designed to work with the IEC60488 class.

The IEC 60488-2:2004(E) defines the following optional parallel poll common commands:

•**IST?* - See IEC 60488-2:2004(E) section 10.15

•**PRE* - See IEC 60488-2:2004(E) section 10.23

•**PRE?* - See IEC 60488-2:2004(E) section 10.24

These are mandatory for devices implementing the PP1 subset.

class slave.iec60488.**PassingControl**(**args*, ***kw*)
Bases: object

A mixin class, implementing the optional passing control command.

The IEC 60488-2:2004(E) defines the following optional passing control command:

•**PCB* - See IEC 60488-2:2004(E) section 10.21

pass_control_back(*primary*, *secondary*)
The address to which the controll is to be passed back.

Tells a potential controller device the address to which the control is to be passed back.

Parameters

- **primary** – An integer in the range 0 to 30 representing the primary address of the controller sending the command.
- **secondary** – An integer in the range of 0 to 30 representing the secondary address of the controller sending the command. If it is missing, it indicates that the controller sending this command does not have extended addressing.

class `slave.iec60488.`**`PowerOn`**(**args*, ***kw*)

Bases: `object`

A mixin class, implementing the optional power-on common commands.

Variables **poweron_status_clear** – Represents the power-on status clear flag. If it is *False* the event status enable, service request enable and serial poll enable registers will retain their status when power is restored to the device and will be cleared if it is set to *True*.

Note: This is a mixin class designed to work with the IEC60488 class

The IEC 60488-2:2004(E) defines the following optional power-on common commands:

- **PSC* - See IEC 60488-2:2004(E) section 10.25
- **PSC?* - See IEC 60488-2:2004(E) section 10.26

class `slave.iec60488.`**`ProtectedUserData`**(**args*, ***kw*)

Bases: `object`

A mixin class, implementing the protected user data commands.

Variables **protected_user_data** – The protected user data. This is information unique to the device, such as calibration date, usage time, environmental conditions and inventory control numbers.

Note: This is a mixin class designed to work with the IEC60488 class.

The IEC 60488-2:2004(E) defines the following optional protected user data commands:

- **RDT* - See IEC 60488-2:2004(E) section 10.27
- **RDT?* - See IEC 60488-2:2004(E) section 10.28

class `slave.iec60488.`**`ResourceDescription`**(**args*, ***kw*)

Bases: `object`

A mixin class, implementing the optional resource description common commands.

Variables **resource_description** – Represents the content of the resource description memory.

Note: Writing does not perform any validation.

Note: This is a mixin class designed to work with the IEC60488 class.

The IEC 60488-2:2004(E) defines the following optional resource description common commands:

- **RDT* - See IEC 60488-2:2004(E) section 10.30
- **RDT?* - See IEC 60488-2:2004(E) section 10.31

class `slave.iec60488.`**`StoredSetting`**(**args*, ***kw*)

Bases: `object`

A mixin class, implementing the optional stored setting commands.

The IEC 60488-2:2004(E) defines the following optional stored setting commands:

- **RCL* - See IEC 60488-2:2004(E) section 10.29
- **SAV* - See IEC 60488-2:2004(E) section 10.33

`recall`(*idx*)

Restores the current settings from a copy stored in local memory.

Parameters **idx** – Specifies the memory slot.

save(*idx*)
Stores the current settings of a device in local memory.

Parameters **idx** – Specifies the memory slot.

class `slave.iec60488.`**SystemConfiguration**(**args*, ***kw*)
Bases: `object`

A mixin class, implementing the optional system configuration commands.

The IEC 60488-2:2004(E) defines the following optional system configuration commands:

- **AAD* - See IEC 60488-2:2004(E) section 10.1
- **DLF* - See IEC 60488-2:2004(E) section 10.6

accept_address()
Executes the accept address command.

disable_listener()
Executes the disable listener command.

class `slave.iec60488.`**Trigger**(**args*, ***kw*)
Bases: `object`

A mixin class, implementing the optional trigger command.

Variables **protected_user_data** – The protected user data. This is information unique to the device, such as calibration date, usage time, environmental conditions and inventory control numbers.

Note: This is a mixin class designed to work with the IEC60488 class.

The IEC 60488-2:2004(E) defines the following optional trigger command:

- **TRG* - See IEC 60488-2:2004(E) section 10.37

It is mandatory for devices implementing the DT1 subset.

trigger()
Creates a trigger event.

Note: It first tries to execute *transport._trigger()*. If this fails, the **TRG* is sent.

class `slave.iec60488.`**TriggerMacro**(**args*, ***kw*)
Bases: `object`

A mixin class, implementing the optional trigger macro commands.

Variables **trigger_macro** – The trigger macro, e.g. *'#217TRIG WFM;MEASWFM?'*.

Note: This is a mixin class designed to work with the IEC60488 class and the Trigger mixin.

The IEC 60488-2:2004(E) defines the following optional trigger macro commands:

- **DDT* - See IEC 60488-2:2004(E) section 10.4
- **DDT?* - See IEC 60488-2:2004(E) section 10.5

3.1.7 `lakeshore` Module

The ls340 module implements an interface for the Lakeshore model LS340 temperature controller.

The `LS340` class models the excellent Lakeshore model LS340 temperature controller. Using it is simple:

```
# We use pyvisa to connect to the controller.
import visa
from slave.lakeshore import LS340

# We assume the LS340 is listening on GPIB channel.
ls340 = LS340(visa.instrument('GPIB::08'))
# Show kelvin reading of channel A.
print ls340.input['A'].kelvin

# Filter channel 'B' data through 10 readings with 2% of full scale window.
ls340.input['B'].filter = True, 10, 2
```

Since the `LS340` supports different scanner options, these are supported as well. They extend the available input channels. To use them one simply passes the model name at construction, e.g.:

```
import visa
from slave.lakeshore import LS340

# We assume the LS340 is equipped with the 3468 eight channel input option
# card.
ls340 = LS340(visa.instrument('GPIB::08'), scanner='3468')

# Show sensor reading of channel D2.
print ls340.input['D2'].sensor_units
```

class `slave.lakeshore.ls340.`**`Column`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

Represents a column of records.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The column index.

The LS340 stores data in table form. Each row is a record consisting of points. Each column has an associated type. The type can be read or written with `type()`. The records can be accessed via the indexing syntax, e.g.

```
# Assuming an LS340 instance named ls340, the following should print
# point1 of record 7.
print ls340.column1[7]
```

Note: Currently there is no parsing done on the type and the record. These should be written or read as strings according to the manual. Also slicing is not supported yet.

`type`

class `slave.lakeshore.ls340.`**`Curve`**(*transport*, *protocol*, *idx*, *writeable*, *length=None*)

Bases: `slave.core.InstrumentBase`

Represents a LS340 curve.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The curve index.
- **writeable** – Specifies if the represented curve is read-only or writeable as well. User curves are writeable in general.
- **length** – The maximum number of points. Default: 200.

Variables header – The curve header configuration. *(<name><serial><format><limit><coefficient>)*, where

- *<name>* The name of the curve, a string limited to 15 characters.
- *<serial>* The serial number, a string limited to 10 characters.
- *<format>* Specifies the curve data format. Valid entries are *'mV/K'*, *'V/K'*, *'Ohm/K'*, *'logOhm/K'*, *'logOhm/logK'*.
- *<limit>* The curve temperature limit in Kelvin.
- *<coefficient>* The curves temperature coefficient. Valid entries are *'negative'* and *'positive'*.

The Curve is implementing the *collections.sequence* protocoll. It models a sequence of points. These are tuples with the following structure *(<units value>, <temp value>)*, where

- *<units value>* specifies the sensor units for this point.
- *<temp value>* specifies the corresponding temperature in kelvin.

To access the points of this curve, use slicing operations, e.g.:

```
# assuming an LS340 instance named ls340, the following will print the
# sixth point of the first user curve.
curve = ls340.user_curve[0]
print curve[5]

# You can use negative indices. This will print the last point.
print curve[-1]

# You can use the builtin function len() to get the length of the curve
# buffer. This is **not** the length of the stored points, but the
# maximum number of points that can be stored in this curve.
print len(curve)

#Extended slicing is available too. This will print every second point.
print curve[::2]

# Set this curves data point to 0.10191 sensor units and 470.000 K.
curve[5] = 0.10191, 470.000

# You can use slicing as well
points = [
    (0.1, 470.),
    (0.2, 480.),
    (0.4, 490.),
]
curve[2:6:2] = points
# To copy a complete sequence of points in one go, do
curve[:] = sequence_of_points
# This will copy all points in the sequence, but points exceeding the
# buffer length are stripped.
```

Warning: In contrast to the LS340 device, point indices start at 0 **not** 1.

delete()
Deletes the current curve.

Raises RuntimeError Raises when‘ when one tries to delete a read-only curve.

class slave.lakeshore.ls340.**Heater**(*transport*, *protocol*)
Bases: slave.core.InstrumentBase

Represents the LS340 heater.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

Variables

- **output** – The heater output in percent.
- **range** – The heater range. An integer between 0 and 5, where 0 deactivates the heater.
- **status** – The heater error status.

ERROR_STATUS = [u’no error’, u’power supply over voltage’, u’power supply under voltat’, u’output digital-to-analog co

class slave.lakeshore.ls340.**Input**(*transport*, *protocol*, *channels*)
Bases: slave.core.InstrumentBase, _abcoll.Mapping

class slave.lakeshore.ls340.**InputChannel**(*transport*, *protocol*, *name*)
Bases: slave.core.InstrumentBase

Represents a LS340 input channel.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **name** – A string value indicating the input in use.

Variables

- **alarm** – The alarm configuration, represented by the following tuple *(<enabled>, <source>, <high value>, <low value>, <latch>, <relay>)*, where:
  - *<enabled>* Enables or disables the alarm.
  - *<source>* Specifies the input data to check.
  - *<high value>* Sets the upper limit, where the high alarm sets off.
  - *<low value>* Sets the lower limit, where the low alarm sets off.
  - *<latch>* Enables or disables a latched alarm.
  - *<relay>* Specifies if the alarm can affect the relays.
- **alarm_status** – The high and low alarm status, represented by the following list: *(<high status>, <low status>)*.
- **celsius** – The input value in celsius.

- **curve** – The input curve number. An Integer in the range [0-60].
- **filter** – The input filter parameters, represented by the following tuple: *(<enable>, <points>, <window>)*.
- **input_type** – The input type configuration, represented by the tuple: *(<type>, <units>, <coefficient>, <excitation>, <range>)*, where
  - <*type*> Is the input sensor type.
  - <*units*> Specifies the input sensor units.
  - <*coefficient*> The input coefficient.
  - <*excitation*> The input excitation.
  - <*range*> The input range.
- **kelvin** – The kelvin reading.
- **linear** – The linear equation data.
- **linear_equation** – The input linear equation parameters. *(<equation>, <m>, <x source>, <b source>, <b>)*, where
  - <*equation*> is either *'slope-intercept'* or *'point-slope'*, meaning 'y = mx + b' or 'y = m(x + b)'.
  - <*m*> The slope.
  - <*x source*> The input data to use, either 'kelvin', 'celsius' or 'sensor units'.
  - <*b source*> Either 'value', '+sp1', '-sp1', '+sp2' or '-sp2'.
  - <*b*> The b value if <*b source*> is set to 'value'.
- **linear_status** – The linear status register.
- **minmax** – The min max data, *(<min>, <max>)*, where
  - <*min*> Is the minimum input data.
  - <*max*> Is the maximum input data.
- **minmax_parameter** – The minimum maximum input function parameters. *(<on/pause>, <source>)*, where
  - <*on/pause*> Starts/pauses the min/max function. Valid entries are *'on'*, *'pause'*.
  - <*source*> Specifies the input data to process. Valid entries are *'kelvin'*, *'celsius'*, *'sensor units'* and *'linear'*.
- **minmax_status** – The min/max reading status. *(<min status>, <max status>)*, where
  - <*min status*> is the reading status register of the min value.
  - <*max status*> is the reading status register of the max value.
- **reading_status** – The reading status register.
- **sensor_units** – The sensor units reading of the input.
- **set** – The input setup parameters, represented by the following tuple: *(<enable>, <compensation>)*

`READING_STATUS` **= {0: u'invalid reading', 1: u'old reading', 4: u'temp underrange', 5: u'temp overrange', 6: u'units z**

class `slave.lakeshore.ls340.`**`LS340`**(*transport*, *scanner=None*)

Bases: `slave.iec60488.IEC60488`

Represents a Lakeshore model LS340 temperature controller.

The LS340 class implements an interface to the Lakeshore model LS340 temperature controller.

Parameters

- **transport** – An object, modeling the transport interface, used to communicate with the real instrument.
- **scanner** – A string representing the scanner in use. Valid entries are
  - *None*, No scanner is used.
  - *"3462"*, The dual standard input option card.
  - *"3464"*, The dual thermocouple input option card.
  - *"3465"*, The single capacitance input option card.
  - *"3468"*, The eight channel input option card.

Variables

- **input** – An instance of `Input`.
- **beeper** – A boolean value representing the beeper mode. *True* means enabled, *False* means disabled.
- **beeping** – A Integer value representing the current beeper status.
- **busy** – A Boolean representing the instrument busy status.
- **columnx** – A Column instance, x is a placeholder for an integer between 1 and 4.
- **com** – The serial interface configuration, represented by the following tuple: *(<terminator>, <baud rate>, <parity>)*.
  - *<terminator>* valid entries are *"CRLF"*, *"LFCR"*, *"CR"*, *"LF"*
  - *<baud rate>* valid entries are 300, 1200, 2400, 4800, 9600, 19200
  - *<parity>* valid entries are 1, 2, 3. See LS340 manual for meaning.
- **datetime** – The configured date and time. *(<MM>, <DD>, <YYYY>, <HH>, <mm>, <SS>, <sss>)*, where
  - *<MM>* represents the month, an Integer in the range 1-12.
  - *<DD>* represents the day, an Integer in the range 1-31.
  - *<YYYY>* represents the year.
  - *<mm>* represents the minutes, an Integer in the range 0-59.
  - *<SS>* represents the seconds, an Integer in the range 0-59.
  - *<sss>* represents the miliseconds, an Integer in the range 0-999.
- **digital_io_status** – The digital input/output status. *(<input status>, <output status>)*, where
  - *<input status>* is a Register representing the state of the 5 input lines DI1-DI5.
  - *<output status>* is a Register representing the state of the 5 output lines DO1-DO5.

- **digital_output_param** – The digital output parameters. *(<mode>, <digital output>)*, where:
  - *<mode>* Specifies the mode of the digital output, valid entries are *'off'*, *'alarms'*, *'scanner'*, *'manual'*,
  - *<digital output>* A register to enable/disable the five digital outputs DO1-DO5, if *<mode>* is *'manual'*.
- **display_fieldx** – The display field configuration values. x is just a placeholder and varies between 1 and 8, e.g. *.display_field2*. *(<input>, '<source>')*, where
  - *<input>* Is the string name of the input to display.
  - *<source>* Specifies the data to display. Valid entries are *'kelvin'*, *'celsius'*, , *'sensor units'*, *'linear'*, *'min'* and *'max'*.
- **heater** – An instance of the `Heater` class.
- **high_relay** – The configuration of the high relay, represented by the following tuple *(<mode>, <off/on>)*, where
  - *<mode>* specifies the relay mode, either *'off'* , *'alarms'* or *'manual'*.
  - *<off/on>* A boolean enabling disabling the relay in manual mode.
- **high_relay_status** – The status of the high relay, either *'off'* or *'on'*.
- **ieee** – The IEEE-488 interface parameters, represented by the following tuple *(<terminator>, <EOI enable>, <address>)*, where
  - *<terminator>* is *None*, *\r\n*, *\n\r*, *\r* or *\n*.
  - *<EOI enable>* A boolean.
  - *<address>* The IEEE-488.1 address of the device, an integer between 0 and 30.
- **key_status** – A string representing the keypad status, either *'no key pressed'* or *'key pressed'*.
- **lock** – A tuple representing the keypad lock-out and the lock-out code. *(<off/on>, <code>)*.
- **logging** – A Boolean value, enabling or disabling data logging.
- **logging_params** – The data logging parameters. *(<type>, <interval>, <overwrite>, <start mode>)*, where
  - *<type>* Valid entries are *'readings'* and *'seconds'*.
  - ***<interval>* The number of readings between each record if *<type>*** is readings and number of seconds between each record otherwise. Valid entries are 1-3600.
  - *<overwrite>* *True* if overwrite is enabled, *False* otherwise.
  - *<start mode>* The start mode, either *clear* or *continue*.

  Note: If no valid SRAM data card is installed, queriing returns *('invalid', 0, False, 'clear')*.

- **loop1** – An instance of the Loop class, representing the first control loop.
- **loop2** – Am instance of the Loop class, representing the second control loop.
- **low_relay** – The configuration of the low relay, represented by the following tuple *(<mode>, <off/on>)*, where
  - *<mode>* specifies the relay mode, either *'off'* , *'alarms'* or *'manual'*.

  - *<off/on>* A boolean enabling disabling the relay in manual mode.
- **low_relay_status** – The status of the low relay, either *'off'* or *'on'*.
- **mode** – Represents the interface mode. Valid entries are *"local"*, *"remote"*, *"lockout"*.
- **output1** – First output channel.
- **output2** – Second output channel.
- **programs** – A tuple of 10 program instances.
- **program_status** – The status of the currently running program represented by the following tuple: *(<program>, <status>)*. If program is zero, it means that no program is running.
- **revision** – A tuple representing the revision information. *(<master rev date>, <master rev number>, <master serial number>, <switch setting SW1>, <input rev date>, <input rev number>, <option ID>, <option rev date>, <option rev number>)*.
- **scanner_parameters** – The scanner parameters. *(<mode>, <channel>, <intervall>)*, where
  - *<mode>* represents the scan mode. Valid entries are *'off'*, *'manual'*, *'autoscan'*, *'slave'*.
  - *<channel>* the input channel to use, an integer in the range 1-16.
  - *<interval>* the autoscan intervall in seconds, an integer in the range 0-999.
- **std_curve** – A tuple of 20 standard curves. These `Curve` instances are read-only.
- **user_curve** – A tuple of 40 user definable `Curve` instances. These are read and writeable.

PROGRAM_STATUS = [u'No errors', u'Too many call commands', u'Too many repeat commands', u'Too many end repeat

clear_alarm()
Clears the alarm status for all inputs.

lines()
The number of program lines remaining.

reset_minmax()
Resets Min/Max functions for all inputs.

save_curves()
Updates the curve flash with the current user curves.

scanner
A string representing the different scanner models supported by the ls340 temperature controller. Valid entries are:

- *"3462"*, The dual standard input option card.
- *"3464"*, The dual thermocouple input option card.
- *"3465"*, The single capacitance input option card.
- *"3468"*, The eight channel input option card.

The different scanner options support a different number of input channels.

softcal(*std*, *dest*, *serial*, *T1*, *U1*, *T2*, *U2*, *T3=None*, *U3=None*)
Generates a softcal curve.

Parameters

- **std** – The standard curve index used to calculate the softcal curve. Valid entries are 1-20
- **dest** – The user curve index where the softcal curve is stored. Valid entries are 21-60.

- **serial** – The serial number of the new curve. A maximum of 10 characters is allowed.
- **T1** – The first temperature point.
- **U1** – The first sensor units point.
- **T2** – The second temperature point.
- **U2** – The second sensor units point.
- **T3** – The third temperature point. Default: *None*.
- **U3** – The third sensor units point. Default: *None*.

stop_program()
Terminates the current program, if one is running.

class slave.lakeshore.ls340.**Loop**(*transport*, *protocol*, *idx*)
Bases: slave.core.InstrumentBase

Represents a LS340 control loop.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The loop index.

Variables

- **display_parameters** – The display parameter of the loop. *(<loop>, <resistance>, <current/power>, <large output enable>)*, where
  - *<loop>* specifies how many loops should be displayed. Valid entries are *'none'*, *'loop1'*, *'loop2'*, *'both'*.
  - *<resistance>* The heater load resistance, an integer between 0 and 1000.
  - *<current/power>* Specifies if the heater output should be displayed as current or power. Valid entries are *'current'* and *'power'*.
  - *<large output enable>* Disables/Enables the large output display.
- **filter** – The loop filter state.
- **limit** – The limit configuration, represented by the following tuple *(<limit>, <pos slope>, <neg slope>, <max current>, <max range>)*
- **manual_output** – The manual output value in percent of full scale. Valid entries are floats in the range -100.00 to 100.00 with a resolution of 0.01.
- **mode** – The control-loop mode. Valid entries are *'manual', 'zone', 'open', 'pid', 'pi', 'p'*
- **parameters** – The control loop parameters, a tuple containing *(<input>, <units>, <enabled>, <powerup>)*, where
  - *<input>* specifies the input channel. Valid entries are *'A'* and *'B'*.
  - *<units>* The setpoint units. Either *'kelvin'*, *'celsius'* or *'sensor'*.
  - *<enabled>* A boolean enabling/disabling the control loop.
  - *<powerup>* Specifies if the control loop is enabled/disabled after powerup.
- **pid** – The PID values.

- **ramp** – The control-loop ramp parameters, represented by the following tuple *(<enabled>, <rate>)*, where
  - *<enabled>* Enables, disables the ramping.
  - *<rate>* Specifies the ramping rate in kelvin/minute.
- **ramping** – The ramping status. *True* if ramping and *False* otherwise.
- **setpoint** – The control-loop setpoint in its configured units.
- **settle** – The settle parameters. *(<threshold>, <time>)*, where
  - ***<threshold>* Specifies the allowable band around the setpoint. Must** be between 0.00 and 100.00.
  - *<time>* The time in seconds, the reading must stay within the band. Valid entries are 0-86400.

  Note: This command is only available for loop1.
- **tuning_status** – A boolean representing the tuning status, *True* if tuning *False* otherwise. .. note:: This attribute is only available for loop1.
- **zonex** – There are 11 zones, zone1 is the first. The zone attribute represents the control loop zone table parameters. *(<top>, <p>, <i>, <d>, <mout>, <range>)*.

class `slave.lakeshore.ls340.`**`Output`**(*transport*, *protocol*, *channel*)
Bases: `slave.core.InstrumentBase`

Represents a LS340 analog output.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **channel** – The analog output channel. Valid are either 1 or 2.

Variables analog – The analog output parameters, represented by the tuple *(<bipolar>, <mode>, <input>, <source>, <high>, <low>, <manual>)*, where:

- *<bipolar>* Enables bipolar output.
- *<mode>* Valid entries are *'off'*, *'input'*, *'manual'*, *'loop'*. *'loop'* is only valid for the output channel 2.
- *<input>* Selects the input to monitor (Has no effect if mode is not *'input'*).
- *<source>* Selects the input data, either *'kelvin'*, *'celsius'*, *'sensor'* or *'linear'*.
- *<high>* Represents the data value at which 100% is reached.
- *<low>* Represents the data value at which the minimum value is reached (-100% for bipolar, 0% otherwise).
- *<manual>* Represents the data value of the analog output in manual mode.

class `slave.lakeshore.ls340.`**`Program`**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

Represents a LS340 program.

Parameters

- **transport** – A transport object.

- **protocol** – A protocol object.
- **idx** – The program index.

Note: There is currently no parsing done on program lines. Lines are read and written as strings according to the LS340 manual.

---

append_line(*new_line*)
Appends the new_line to the LS340 program.

delete()
Deletes this program.

line(*idx*)
Return the i'th program line.

Parameters **i** – The i'th program line.

run()
Runs this program.

class slave.lakeshore.ls370.**Curve**(*transport*, *protocol*, *idx*, *length*)
Bases: slave.core.InstrumentBase

A LS370 curve.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The curve index.
- **length** – The curve buffer length.

Variables **header** – The curve header configuration. *(<name><serial><format><limit><coefficient>)*, where

- <*name*> The name of the curve, a string limited to 15 characters.
- <*serial*> The serial number, a string limited to 10 characters.
- <*format*> Specifies the curve data format. Valid entries are 'Ohm/K' and 'logOhm/K'.
- <*limit*> The curve temperature limit in Kelvin.
- <*coefficient*> The curves temperature coefficient. Valid entries are *'negative'* and *'positive'*.

The Curve is implementing the *collections.sequence* protocoll. It models a sequence of points. These are tuples with the following structure *(<units value>, <temp value>)*, where

- <*units value*> specifies the sensor units for this point.
- <*temp value*> specifies the corresponding temperature in kelvin.

To access the points of this curve, use indexing and slicing operations, e.g.:

```
# assuming an LS30 instance named ls30, the following will print the
# sixth point of the first user curve.
curve = ls370.user_curve[0]
print curve[1]   # print second point
print curve[-1]  # print last point
print curve[::2] # print every second point
```

```
# Set the fifth data point to 0.10191 sensor units and 470.000 K.
curve[5] = 0.10191, 470.000
```

Note: Be aware that the builtin `len()` function returns the buffer length, **not** the number of points.

Warning: In contrast to the LS370 device, point indices start at 0 **not** 1.

`delete()`
Deletes this curve.

class `slave.lakeshore.ls370.`**`Display`**(*transport*, *protocol*, *location*)
Bases: `slave.core.InstrumentBase`

A LS370 Display at the chosen location.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **location** – The display location.

Variables **config** – The configuration of the display. *(<channel>, <source>, <resolution>)*, where

- <*channel*> The index of the displayed channel, 0-16, where 0 activates channel scanning.
- <*source*> The displayed data. Valid entries are ‘kelvin’, ‘ohm’, ‘linear’, ‘min’ and ‘max’
- <*resolution*> The displayed resolution in number of digits, 4-6.

class `slave.lakeshore.ls370.`**`Heater`**(*transport*, *protocol*)
Bases: `slave.core.InstrumentBase`

An LS370 Heater.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

Variables

- **manual_output** – The manual heater output, a float representing the percent of current or actual power depending on the heater output selection.
- **output** – The heater output in percent of current or actual power dependant on the heater output selection.
- **range** – The heater current range. Valid entries are ‘off’, ‘31.6 uA’, ‘100 uA’, ‘316 uA’, ‘1 mA’, ‘3.16 mA’, ‘10 mA’, ‘31.6 mA’, and ‘100 mA’
- **status** – The heater status, either ‘no error’ or ‘heater open error’.

`RANGE` **= [u’off’, u‘31.6 uA’, u‘100 uA’, u‘316 uA’, u‘1 mA’, u‘3.16 mA’, u‘10 mA’, u‘31.6 mA’, u‘100 mA’]**
The supported heater ranges.

class `slave.lakeshore.ls370.`**`Input`**(*transport*, *protocol*, *channels*)
Bases: `slave.core.InstrumentBase`, `_abcoll.Sequence`

The LS370 Input.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

Variables scan –

It is a sequence like interface to each `InputChannel`.

E.g. to access the kelvin reading of channel 5, Assuming an instance of `LS370` named *ls370*, one would simply write.:

```
>>> ls370.input[5].kelvin
```

To scan the second channel, and activate the autoscan one would write:

```
>>> ls370.input.scan = 2, True
```

Note: In contrast to the LS370 internal commands the channel indexing is zero based.

class `slave.lakeshore.ls370.`**`InputChannel`**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

A LS370 input channel.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The channel index.

Variables

- **alarm** – The alarm configuration. *(<enabled>, <source>, <high>, <low>, <deadband>, <latch>)*, where
  - *<enable>* enables/disables the alarm, valid are *True*, *False*.
  - *<source>* The data channel against which the alarm condition is checked. Either 'kelvin', 'ohm' or 'linear'.
  - *<high>* The high alarm value.
  - *<low>* The low alarm value.
  - *<deadband>* The value the source value must change to deactivate the non-latched alarm.
  - *<latch>* Enables/disables the latched alarm. (A latched alarm stays active, even if the alarm condition isn't met anymore).
- **alarm_status** – The status of the high and low alarm. *(<high state>, <low state>)*, where
  - *<high state>* is either *True'or 'False*.
  - *<low state>* is either *True'or 'False*.
- **config** – The input channel configuration. *(<enabled>, <dwell>, <pause>, <curve>, <co-efficient>)*, where
  - *<enabled>* enables/disables the channel, valid are *True*, *False*.
  - *<dwell>* The autoscanning dwell time in seconds, 1-200.
  - *<pause>* The change pause time in seconds, 3-200.

  - <*curve*> The curve used by the channel, valid are 'no curve' or 0-19 the index of the user curves.
  - <*coefficient*> The temperature coefficient used if no curve is selected. Valid are 'negative' and 'positive'.

- **excitation_power** – The current excitation power.
- **filter** – The filter parameters. *(<enabled>, <settle time>, <window>)*, where
  - <*enabled*> A boolean enabling/disabling the filtering.
  - <*settle time*> The settle time in seconds, 1-200.
  - <*window*> The filtering window, 1-80 in precent of the fullscale reading.
- **index** – The index of the input channel.
- **kelvin** – The input channel reading in kelvin.

Note: If no curve is present, the reading will be *0.*.

- **linear** – Linear equation data.
- **linear_equation** – The input linear equation parameters. *(<equation>, <m>, <x source>, <b source>, <b>)*, where
  - <*equation*> is either *'slope-intercept'* or *'point-slope'*, meaning 'y = mx + b' or 'y = m(x + b)'.
  - <*m*> The slope.
  - <*x source*> The input data to use, either 'kelvin', 'celsius' or 'sensor units'.
  - <*b source*> Either 'value', '+sp1', '-sp1', '+sp2' or '-sp2'.
  - <*b*> The b value if <*b source*> is set to 'value'.
- **minmax** – The min max data, *(<min>, <max>)*, where
  - <*min*> Is the minimum input data.
  - <*max*> Is the maximum input data.
- **minmax_param** – Configures the source data to use with the minmax filter. Valid are 'kelvin', 'ohm' and 'linear'.
- **reading_status** – The channel reading status. A register with the following keys
  - 'cs overload' Current source overload.
  - 'vcm overload' Common mode voltage overload.
  - 'vmix overload' Mixer overload.
  - 'vdif overload' Differential overload.
  - 'range over' The selected resistance range is too low.
  - 'range under' The the polarity (+/-) of the current or voltage leads is wrong and the selected resistance range is too low.
- **resistance** – The input reading in ohm.
- **resistance_range** – The resistance range configuration. *(<mode>, <excitation>, <range>, <autorange>, <excitation_enabled>)*
  - <*mode*> The excitation mode, either 'current' or 'voltage'.

– <*excitation*> The excitation range, either 1-22 for current excitation or 1-12 for voltage excitation.

class `slave.lakeshore.ls370.`**`LS370`**(*transport*, *scanner=None*)

Bases: `slave.iec60488.IEC60488`

A lakeshore mode ls370 resistance bridge.

Represents a Lakeshore model ls370 ac resistance bridge.

Parameters **transport** – A transport object.

Variables

- **baud** – The baud rate of the rs232 interface. Valid entries are *300*, *1200* and *9600*.
- **beeper** – A boolean value representing the beeper mode. *True* means enabled, *False* means disabled.
- **brightness** – The brightness of the frontpanel display in percent. Valid entries are *25*, *50*, *75* and *100*.
- **common_mode_reduction** – The state of the common mode reduction.
- **control_mode** – The temperature control mode, valid entries are 'closed', 'zone', 'open' and 'off'.
- **control_params** – The temperature control parameters. *(<channel>, <filter>, <units>, <delay>, <output>, <limit>, <resistance>)*, where
  - <*channel*> The input channel used for temperature control.
  - <*filter*> The filter mode, either 'filtered' or 'unfiltered'.
  - <*units*> The setpoint units, either 'kelvin' or 'ohm'.
  - <*delay*> The delay in seconds used for the setpoint change during autoscan. An integer between 1 and 255.
  - <*output*> The heater output display, either 'current' or 'power'.
  - <*limit*> The maximum heater range. See `Heater.RANGE`.
  - <*resistance*> The heater load in ohms. Valid entries are 1. to 100000.
- **digital_output** – A register enabling/disabling the digital output lines.
- **displays** – A tuple of `Display` instances, representing the 7 available display locations.
- **display_locations** – The number of displayed locations, between 1 and 8.
- **frequency** – The excitation frequency. Valid entries are '9.8 Hz', '13.7 Hz' and '16.2 Hz'.

  Note: This commands takes several seconds to complete

- **heater** – An instance of the `Heater` class.
- **ieee** – The IEEE-488 interface parameters, represented by the following tuple *(<terminator>, <EOI enable>, <address>)*, where
  - <*terminator*> is *None*, *\r\n*, *\n\r* or *\n*.
  - <*EOI enable*> A boolean.
  - <*address*> The IEEE-488.1 address of the device, an integer between 0 and 30.
- **input** – An instance of `Input`.

- **input_change** – Defines if range and excitation keys affects all or only one channel. Valid entries are 'all', 'one'.
- **mode** – Represents the interface mode. Valid entries are *"local"*, *"remote"*, *"lockout"*.
- **monitor** – The monitor output selection, one of 'off', 'cs neg', 'cs pos', 'vad', 'vcm neg', 'vcm pos', 'vdif' or 'vmix'.
- **output** – A tuple, holding two `Output` objects
- **pid** – The pid loop settings. *(<p>, <i>, <d>)*, where
  - *<p>* The proportional gain, a float in the range 0.001 to 1000.
  - *<i>* The integral action, a float in the range 0 to 10000.
  - *<d>* The derivative action, a float in the range 0 to 2500.
- **polarity** – The polarity of the temperature control. Valid entries are 'unipolar' and 'bipolar'.
- **ramp** – The setpoint ramping parameters. *(<enabled>, <rate>)*, where
  - *<enabled>* Is a boolean, enabling/disabling the ramping.
  - *<rate>* A float representing the ramping rate in kelvin per minute in the range 0.001 to 10.
- **ramping** – The ramping status, either *True* or *False*.
- **low_relay** – The low relay, an instance of `Relay`.
- **high_relay** – The high relay, an instance of `Relay`.
- **setpoint** – The temperature control setpoint.
- **still** – The still output value.

  Note: The still only works, if it's properly configured in the analog output 2.

- **all_curves** – A `Curve` instance that can be used to configure all user curves simultaneously. Instead of iterating of the `user_curve` attribute one can use this command. This way less commands will be send.
- **user_curve** – A tuple of 20 `Curve` instances.
- **zones** – A sequence of 10 Zones. Each zone is represented by a tuple *(<top>, <p>, <i>, <d>, <manual>, <heater>, <low>, <high>, <analog1> , <analog2>)*, where
  - *<top>* The setpoint limit of this zone.
  - *<p>* The proportional action, 0.001 to 1000.
  - *<i>* The integral action, 0 to 10000.
  - *<d>* The derivative action, 0 to 10000.
  - *<manual>* The manual output in percent, 0 to 100.
  - *<heater>* The heater range.
  - *<low>* The low relay state, either *True* or *False*.
  - *<high>* The high relay state, either *True* or *False*.
  - *<analog1>* The output value of the first analog output in percent. From -100 to 100.
  - *<analog2>* The output value of the second analog output in percent. From -100 to 100.

clear_alarm()

Clears the alarm status for all inputs.

reset_minmax()

Resets Min/Max functions for all inputs.

scanner

The scanner option in use.

Changing the scanner option changes number of input channels available. Valid values are

scanner	channels
None	1
'3708'	8
'3716'	16
'3716L'	16

class `slave.lakeshore.ls370.`**`Output`**(*transport*, *protocol*, *channel*)

Bases: `slave.core.InstrumentBase`

Represents a LS370 analog output.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **channel** – The analog output channel. Valid are either 1 or 2.

Variables

- **analog** – The analog output parameters, represented by the tuple *(<bipolar>, <mode>, <input>, <source>, <high>, <low>, <manual>)*, where:
  - *<bipolar>* Enables bipolar output.
  - ***<mode>* Valid entries are 'off', 'channel', 'manual', 'zone',** 'still'. 'still' is only valid for the output channel 2.
  - *<input>* Selects the input to monitor (Has no effect if mode is not *'input'*).
  - *<source>* Selects the input data, either 'kelvin', 'ohm', 'linear'.
  - *<high>* Represents the data value at which 100% is reached.
  - *<low>* Represents the data value at which the minimum value is reached (-100% for bipolar, 0% otherwise).
  - *<manual>* Represents the data value of the analog output in manual mode.
- **value** – The value of the analog output.

class `slave.lakeshore.ls370.`**`Relay`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

A LS370 relay.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The relay index.

Variables

- **config** – The relay configuration. *(<mode>, <channel>, <alarm>)*, where
  - *<mode>* The relay mode either 'off' 'on', 'alarm' or 'zone'.
  - ***<channel>* Specifies the channel, which alarm triggers the relay.** Valid entries are 'scan' or an integer in the range 1-16.
  - ***<alarm>* The alarm type triggering the relay. Valid are 'low' 'high'** or 'both'.
- **status** – The relay status.

3.1.8 `misc` Module

class `slave.misc.`**`ForwardSequence`**`(`*iterable*, *get*, *set=None*`)`

Bases: `_abcoll.Sequence`

Sequence forwarding item access and write operations.

Parameters

- **iterable** – An iterable of items to be stored.
- **get** – A callable used on item access, receiving the item. It's result is returned.
- **set** – A callable receiving the item and a value on item set operations.

Implements a immutable sequence, which forwards item access and write operations to the stored items.

class `slave.misc.`**`Measurement`**`(`*path*, *measurables*`)`

Bases: `object`

Small measurement helper class.

For each call to `__call__()` a comma separated row, representing the return values for each callable item in measurables is written to the file specified by path.

`close``()`

`open``()`

`slave.misc.`**`index`**`(`*index*, *length*`)`

Generates an index.

Parameters

- **index** – The index, can be positive or negative.
- **length** – The length of the sequence to index.

Raises IndexError

Negative indices are typically used to index a sequence in reverse order. But to use them, the indexed object must convert them to the correct, positive index. This function can be used to do this.

3.1.9 `quantum_design` Module

class `slave.quantum_design.ppms.`**`AnalogOutput`**`(`*transport*, *protocol*, *id*`)`

Bases: `slave.core.InstrumentBase`

Represents an analog output.

Variables

- **id** – The analog output id.

- **voltage** – The voltage present at the analog output channel.

Note: Setting the voltage removes any linkage.

- **link** – Links a parameter to this analog output. It has the form *(<link>, <full>, <mid>)*, where
  - *<link>* is the parameter to link. See `STATUS_LINK` for valid links.
  - *<full>* the value of the parameter corresponding to full scale output (10 V).
  - *<mid>* the value of the parameter corresponding to mid sclae output (0 V).

`link`

class `slave.quantum_design.ppms.`**`BridgeChannel`**(*transport*, *protocol*, *id*)

Bases: `slave.core.InstrumentBase`

Represents the user bridge configuration.

Variables

- **id** – The user bridge channel id.
- **config** – The bridge configuration, represented by a tuple of the form *(<excitation>, <power limit>, <dc flag>, <mode>)*, where
  - *<excitation>* The excitation current in microamps from 0.01 to 5000.
  - ***<power limit>* The maximum power to be applied in microwatts from** 0.001 to 1000.
  - ***<dc flag>* Selects the excitation type. Either 'AC' or 'DC'. 'AC'** corresponds to a square wave excitation of 7.5 Hz.
  - ***<mode>* Configures how often the internal analog-to-digital converter** recalibrates itself. Valid are 'standart', 'fast' and 'high res'.
- **resistance** – The resistance of the user channel in ohm.
- **current** – The current of the user channel in microamps.

class `slave.quantum_design.ppms.`**`PPMS`**(*transport*, *max_field=None*)

Bases: `slave.iec60488.IEC60488`

A Quantum Design Model 6000 PPMS.

Parameters

- **transport** – A transport object.
- **max_field** – The maximum magnetic field allowed in Oersted. If *None*, the default, it's read back from the ppms.

Note: The ppms needs a newĺine '\n' character as message terminator. Using delay between read and write operations is recommended as well.

Variables

- **advisory_number** – The advisory code number, a read only integer in the range 0 to 999.
- **chamber** – The configuration of the sample chamber. Valid entries are 'seal', 'purge seal', 'vent seal', 'pump' and 'vent', where
  - 'seal' seals the chamber immediately.

  - 'purge seal' purges and then seals the chamber.
  - 'vent seal' ventilates and then seals the chamber.
  - 'pump' pumps the chamber continuously.
  - 'vent' ventilates the chamber continuously.
- **sample_space_pressure** – The pressure of the sample space in user units. (read only)
- **system_status** – The general system status.

Configuration

Variables

- **bridges** – A list of `BridgeChannel` instances representing all four user bridges.

  Note: Python indexing starts at 0, so the first bridge has the index 0.

- **date** – The configured date of the ppms computer represented by a python *datetime.date* object.
- **time** – The configured time of the ppms computer, represented by a python *datetime.time* object.
- **analog_output** – A tuple of `AnalogOutput` instances corresponding to the four analog outputs.
- **digital_input** – The states of the digital input lines. A dict with the following keys

Keys	Pinouts
'Motor Port - Limit 1'	P10-4,5
'Motor Port - Limit 2'	P10-9,5
'Aux Port - Sense 1'	P8-18,19
'Aux Port - Sense 2'	P8-6,19
'Ext Port - Busy'	P11-9
'Ext Port - User'	P11-5

  A dict value of True means the line is asserted.

  (read only)

- **digital_output** – The state of the digital output lines. A dict with the following keys

Keys	Connector Port	Pinouts
'Drive Line 1'	Auxiliary Port	P8-1,14
'Drive Line 2'	Auxiliary Port	P8-2,15
'Drive Line 3'	Auxiliary Port	P8-3,16
'Actuator Drive'	Motor Port	P10-3,8

  A dict value of *True* means the line is set to -24 V output. Setting it with a dict containing only some keys will only change these. The other lines will be left unchanged.

- **driver_output** – A `CommandSequence` representing the driver outputs of channel 1 and 2. Each channel is represented by a tuple of the form *(<current>, <power limit>)*, where
  - *<current>* is the current in mA, in the range 0 to 1000.
  - *<power limit>* is the power limit in W, in the range 0 to 20.

  Note: Python indexing starts with 0. Therefore channel 1 has the index 0.

- **external_select** – The state of the external select lines. A dict with the following keys

Key	Connector Port	Pinouts
Select	1 External Port	P11-1,6
Select	2 External Port	P11-2,7
Select	3 External Port	P11-3,8

  A dict value of *True* means the line is asserted (switch closed). Setting it with a dict containing only some keys will only change these. The other lines will be left unchanged.

- **revision** – The revision number. (read only)

Helium Level Control

Variables **level** – The helium level, represented by a tuple of the form *(<level>, <age>)*, where

- *<level>* The helium level in percent.
- *<age>* is the age of the reading. Either '>1h', '<1h' or 'continuous'.

Magnet Control

Variables

- **field** – The current magnetic field in Oersted(read only).
- **target_field** – The magnetic field configuration, represented by the following tuple *(<field>, <rate>, <approach mode>, <magnet mode>)*, where
  - *<field>* is the magnetic field setpoint in Oersted with a resolution of 0.01 Oersted. The min and max fields depend on the magnet used.
  - *<rate>* is the ramping rate in Oersted/second with a resolution of 0.1 Oersted/second. The min and max values depend on the magnet used.
  - *<approach mode>* is the approach mode, either 'linear', 'no overshoot' or 'oscillate'.
  - *<magnet mode>* is the state of the magnet at the end of the charging process, either 'persistent' or 'driven'.
- **magnet_config** – The magnet configuration represented by the following tuple *(<max field>, <B/I ratio>, <inductance>, <low B charge volt>, <high B charge volt>, <switch heat time>, <switch cool time>)*, where
  - *<max field>* is the max field of the magnet in Oersted.
  - *<B/I ratio>* is the field to current ratio in Oersted/A.
  - *<inductance>* is the inductance in Henry.
  - *<low B charge volt>* is the charging voltage at low B fields in volt.
  - *<high B charge volt>* is the chargin voltage at high B fields in volt.
  - *<switch heat time>* is the time it takes to open the persistent switch in seconds.
  - *<switch cool time>* is the time it takes to close the persistent switch in seconds.

Sample Position

Variables

- **move_config** – The move configuration, a tuple consisting of *(<unit>, <unit/step>, <range>)*, where
  - *<unit>* The unit, valid are 'steps', 'degree', 'radian', 'mm', 'cm', 'mils' and 'inch'.
  - *<unit/step>* the units per step.
  - *<range>* The allowed travel range.
- **move_limits** – The position of the limit switch and the max travel limit, represented by the following tuple *(<lower limit>, <upper limit>)*, where
  - *<lower limit>* The lower limit represents the position of the limit switch in units specified by the move configuration.
  - *<upper limit>* The upper limit in units specified by the move configuration. It is defined by the position of the limit switch and the configured travel range.

  (read only)
- **position** – The current sample position.

Temperature Control

Variables

- **temperature** – The temperature at the sample position in Kelvin (read only).
- **target_temperature** – The temperature configuration, a tuple consisting of *(<temperature>, <rate>, <approach mode>)*, where
  - *<temperature>* The temperature setpoint in kelvin in the range 1.9 to 350.
  - *<rate>* The sweep rate in kelvin per minute in the range 0 to 20.
  - *<approach mode>* The approach mode, either 'fast' or 'no overshoot'.

`beep` (*duration*, *frequency*)
Generates a beep.

Parameters

- **duration** – The duration in seconds, in the range 0.1 to 5.
- **frequency** – The frequency in Hz, in the range 500 to 5000.

`date`

`digital_output`

`external_select`

`field`
The field at sample position.

`levelmeter` (*rate*)
Changes the measuring rate of the levelmeter.

Parameters **rate** – Valid are ‘on’, ‘off’, ‘continuous’ and ‘hourly’. ‘on’ turns on the level meter, takes a reading and turns itself off. In ‘continuous’ mode, the readings are constantly updated. If no reading is requested within 60 seconds, the levelmeter will be turned off. ‘off’ turns off hourly readings.

Note: It takes approximately 10 seconds until a measured level is available.

`move` (*position*, *slowdown=0*)
Move to the specified sample position.

Parameters

- **position** – The target position.
- **slowdown** – The slowdown code, an integer in the range 0 to 14, used to scale the stepper motor speed. 0, the default, is the fastest rate and 14 the slowest.

`move_to_limit` (*position*)
Move to limit switch and define it as position.

Parameters **position** – The new position of the limit switch.

`redefine_position` (*position*)
Redefines the current position to the new position.

Parameters **position** – The new position.

`scan_field` (*measure*, *field*, *rate*, *mode=u’persistent’*, *delay=1*)
Performs a field scan.

Measures until the target field is reached.

Parameters

- **measure** – A callable called repeatedly until stability at the target field is reached.
- **field** – The target field in Oersted.

  Note: The conversion is 1 Oe = 0.1 mT.

- **rate** – The field rate in Oersted per minute.
- **mode** – The state of the magnet at the end of the charging process, either ‘persistent’ or ‘driven’.
- **delay** – The time delay between each call to measure in seconds.

Raises TypeError if measure parameter is not callable.

`scan_temperature` (*measure*, *temperature*, *rate*, *delay=1*)
Performs a temperature scan.

Measures until the target temperature is reached.

Parameters

- **measure** – A callable called repeatedly until stability at target temperature is reached.
- **temperature** – The target temperature in kelvin.
- **rate** – The sweep rate in kelvin per minute.
- **delay** – The time delay between each call to measure in seconds.

set_field (*field*, *rate*, *approach=u'linear'*, *mode=u'persistent'*, *wait_for_stability=True*, *delay=1*)
Sets the magnetic field.

Parameters

- **field** – The target field in Oersted.

  Note: The conversion is 1 Oe = 0.1 mT.

- **rate** – The field rate in Oersted per minute.
- **approach** – The approach mode, either 'linear', 'no overshoot' or 'oscillate'.
- **mode** – The state of the magnet at the end of the charging process, either 'persistent' or 'driven'.
- **wait_for_stability** – If *True*, the function call blocks until the target field is reached and stable.
- **delay** – Specifies the frequency in seconds how often the magnet status is checked. (This has no effect if wait_for_stability is *False*).

set_temperature (*temperature*, *rate*, *mode=u'fast'*, *wait_for_stability=True*, *delay=1*)
Sets the temperature.

Parameters

- **temperature** – The target temperature in kelvin.
- **rate** – The sweep rate in kelvin per minute.
- **mode** – The sweep mode, either 'fast' or 'no overshoot'.
- **wait_for_stability** – If wait_for_stability is *True*, the function call blocks until the target temperature is reached and stable.
- **delay** – The delay specifies the frequency how often the status is checked.

shutdown ()
The temperature controller shutdown.

Invoking this method puts the PPMS in standby mode, both drivers used to control the system temperature are turned off and helium flow is set to a minimum value.

system_status
The system status codes.

temperature
The current temperature at the sample position.

time

slave.quantum_design.ppms.**STATUS_CHAMBER = {0: u'unknown', 1: u'purged, sealed', 2: u'vented, sealed', 3: u'seale**
Chamber status codes.

slave.quantum_design.ppms.**STATUS_DIGITAL_INPUT = {1: u'Motor Port - Limit 1', 2: u'Motor Port - Limit 2', 3: u**
Status of digital input lines.

slave.quantum_design.ppms.**STATUS_DIGITAL_OUTPUT = {0: u'Drive Line 1', 1: u'Drive Line 2', 2: u'Drive Line 3',**
Status of the digital output lines.

slave.quantum_design.ppms.**STATUS_EXTERNAL_SELECT = {0: u'Select 1', 1: u'Select 2', 2: u'Select 3'}**
Status of the external select lines.

`slave.quantum_design.ppms.`**`STATUS_LINK`** **= {0: None, 1: u'Temperature', 2: u'Tield', 3: u'Position', 4: u'User Bridg**
The linking status.

`slave.quantum_design.ppms.`**`STATUS_MAGNET`** **= {0: u'unknown', 1: u'persistent, stable', 2: u'persist switch warming',**
Magnet status codes.

`slave.quantum_design.ppms.`**`STATUS_SAMPLE_POSITION`** **= {0: u'unknown', 1: u'stopped', 5: u'moving', 8: u'limit',**
Sample Position status codes.

`slave.quantum_design.ppms.`**`STATUS_TEMPERATURE`** **= {0: u'unknown', 1: u'normal stability at target temperature', 2**
Temperature controller status code.

3.1.10 `signal_recovery` Module

class `slave.signal_recovery.sr7225.`**`Float`**(*min=None*, *max=None*, **args*, ***kw*)
Bases: `slave.types.Float`

Custom float class used to correct a bug in the SR7225 firmware.

When the SR7225 is queried in floating point mode and the value is exactly zero, it appends a *x00* value, a null byte. To workaround this firmware bug, the null byte is stripped before the conversion to float happens.

class `slave.signal_recovery.sr7225.`**`SR7225`**(*transport*)
Bases: `slave.core.InstrumentBase`

Represents a Signal Recovery SR7225 lock-in amplifier.

Parameters **transport** – A transport object.

Signal Channel

Variables

- **current_mode** – The current mode, either *'off'*, *'high bandwidth'* or *'low noise'*.
- **voltage_mode** – The voltage mode, either *'test'*, *'A'* or *'A-B'*. The *'test'* mode corresponds to both inputs grounded.
- **fet** – The voltage mode input device control. Valid entries are *'bipolar'* and *'fet'*, where
  - *'bipolar'* is a bipolar device with 10kOhm input impedance and 2 nV/sqrt(Hz) voltage noise at 1 kHz.
  - *'fet'* 10MOhm input impedance and 5nV/sqrt(Hz) voltage noise at 1kHz.
- **grounding** – The input connector shield grounding mode. Valid entries are *'ground'* and *'float'*.
- **coupling** – The input connector coupling, either *'ac'* or *'dc'*.
- **sensitivity** – The full-scale sensitivity. The valid entries depend on the current mode.

‘off’	‘high bandwidth’	‘low noise’
‘2 nV’	‘2 fA’	‘2 fA’
‘5 nV’	‘5 fA’	‘5 fA’
‘10 nV’	‘10 fA’	‘10 fA’
‘20 nV’	‘20 fA’	‘20 fA’
‘50 nV’	‘50 fA’	‘50 fA’
‘100 nV’	‘100 fA’	‘100 fA’
‘200 nV’	‘200 fA’	‘200 fA’
‘500 nV’	‘500 fA’	‘500 fA’
‘1 uV’	‘1 pA’	‘1 pA’
‘2 uV’	‘2 pA’	‘2 pA’
‘5 uV’	‘5 pA’	‘5 pA’
‘10 uV’	‘10 pA’	‘10 pA’
‘20 uV’	‘20 pA’	‘20 pA’
‘50 uV’	‘50 pA’	‘50 pA’
‘100 uV’	‘100 pA’	‘100 pA’
‘200 uV’	‘200 pA’	200 pA’
‘500 uV’	‘500 pA’	‘500 pA’
‘1 mV’	‘1 nA’	‘1 nA’
‘2 mV’	‘2 nA’	‘2 nA’
‘5 mV’	‘5 nA’	‘5 nA’
‘10 mV’	‘10 nA’	—
‘20 mV’	‘20 nA’	—
‘50 mV’	‘50 nA’	—
‘100 mV’	‘100 nA’	—
‘200 mV’	‘200 nA’	—
‘500 mV’	‘500 nA’	—
‘1 V’	‘1 uA’	—

- **ac_gain** – The gain of the signal channel amplifier. See `SR7230.AC_GAIN` for valid values.
- **ac_gain_auto** – A boolean corresponding to the ac gain automatic mode. It is *False* if the ac_gain is under manual control, and *True* otherwise.
- **line_filter** – The line filter configuration. *(<filter>, <frequency>)*, where
  - *<filter>* Is the filter mode. Valid entries are *‘off’*, *‘notch’*, *‘double’* or *‘both’*.
  - *<frequency>* Is the notch filter center frequency, either *‘60Hz’* or *‘50Hz’*.
- **sample_frequency** – The sampling frequency. An integer between 0 and 2, corresponding to three different sampling frequencies near 166kHz.

Reference channel

Variables

- **reference** – The reference input mode, either *‘internal’*, *‘ttl’* or *‘analog’*.
- **harmonic** – The reference harmonic mode, an integer between 1 and 32 corresponding to the first to 32. harmonic.
- **reference_phase** – The phase of the reference signal, a float ranging from -360.00 to 360.00 corresponding to the angle in degrees.
- **reference_frequency** – A float corresponding to the reference frequency in Hz. (read only)

Note: If `reference` is not *'internal'* the reference frequency value is zero if the reference channel is unlocked.

Signal channel output filters

Variables

- **slope** – The output lowpass filter slope in dB/octave, either *'6 dB'*, *'12 dB'*, *'18 dB'* or *'24 dB'*.
- **time_constant** – A float representing the time constant in seconds. See `TIME_CONSTANT` for the available values.
- **sync** – A boolean value, representing the state of the synchronous time constant mode.

Signal channel output amplifiers

Variables

- **x_offset** – The x-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **y_offset** – The y-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **expand** – The expansion control, either *'off'*, *'x'*, *'y'* or *'both'*.
- **channel1_output** – The output of CH1 connector of the rear panel Either *'x'*, *'y'*, *'r'*, *'phase1'*, *'phase2'*, *'noise'*, *'ratio'* or *'log ratio'*.
- **channel2_output** – The output of CH2 connector of the rear panel Either *'x'*, *'y'*, *'r'*, *'phase1'*, *'phase2'*, *'noise'*, *'ratio'* or *'log ratio'*.

Instrument outputs

Variables

- **x** – A float representing the X-channel output in either volt or ampere. (read only)
- **y** – A float representing the Y-channel output in either volt or ampere. (read only)
- **xy** – X and Y-channel output with the following format *(<x>, <y>)*. (read only)
- **r** – The signal magnitude, as float. (read only)
- **theta** – The signal phase, as float. (read only)
- **r_theta** – The magnitude and the signal phase. *(<r>, <theta>)*. (read only)
- **ratio** – The ratio equivalent to X/ADC1. (read only)
- **log_ratio** – The ratio equivalent to log(X/ADC1). (read only)

- **noise** – The square root of the noise spectral density measured at the Y channel output. (read only)
- **noise_bandwidth** – The noise bandwidth. (read only)
- **noise_output** – The noise output, the mean absolute value of the Y channel. (read only)
- **star** – The star mode configuration, one off *'x'*, *'y'*, *'r'*, *'theta'*, *'adc1'*, *'xy'*, *'rtheta'*, *'adc12'*

Internal oscillator

Variables

- **amplitude** – A float between 0. and 5. representing the oscillator amplitude in V rms.
- **amplitude_start** – Amplitude sweep start value.
- **amplitude_stop** – Amplitude sweep end value.
- **amplitude_step** – Amplitude sweep amplitude step.
- **frequency** – The oscillator frequency in Hz. Valid entries are 0. to 1.2e5.
- **frequency_start** – The frequency sweep start value.
- **frequency_stop** – The frequency sweep stop value.
- **frequency_step** – The frequency sweep step size and sweep type. *(<step>, <mode>)*, where
  - *<step>* The step size in Hz.
  - *<mode>* The sweep mode, either 'log' or 'linear'.
- **sync_oscillator** – The state of the syncronous oscillator (demodulator) mode.
- **sweep_rate** – The frequency and amplitude sweep step rate in time per step. Valid entries are 0.05 to 1000. in 0.005 steps representing the time in seconds.

Auxiliary outputs

Variables

- **dac1** – The voltage of the auxiliary output 1 on the rear panel, a float between +/- 12.00.
- **dac2** – The voltage of the auxiliary output 2 on the rear panel, a float between +/- 12.00.
- **output_port** – The bits to be output on the rear panel digital output port, an Integer between 0 and 255.

Auxiliary inputs

Variables

- **adc1** – The auxiliary analog-to-digital input 1.
- **adc2** – The auxiliary analog-to-digital input 2.
- **adc_trigger_mode** – The trigger mode of the auxiliary ADC inputs, represented by an integer between 0 and 13.

- **burst_time** – The burst time per point rate for the ADC1 and ADC2. An integer between 25 and 5000 when storing only to ADC1 and 56 to 5000 when storing to ADC1 and ADC2.

  Note: The lower boundary of 56 in the second case is not tested by slave itself.

Output data curve buffer

Variables

- **curve_buffer_settings** – The curve buffer settings define what is to be stored in the curve buffer.
- **curve_buffer_length** – The length of the curve buffer. The max value depends on the the `curve_buffer_settings`.
- **storage_intervall** – The storage intervall, an integer representing the time between datapoints in miliseconds.
- **event_marker** – If the *'event'* flag of the `curve_buffer_settings` is *True*, the content of the event marker variable, an integer between 0 and 32767, is stored for each data point.
- **measurement_status** – The curve acquisition status. *(<acquisition status>, <sweeps>, <lockin status>, <points>)*, where
  - *<acquisition status>* is the curve acquisition status. It is either *'no activity'*, *'td running'*, *'tdc running'*, *'td halted'* or *'tdc halted'*.
  - *<sweeps>* The number of sweeps acquired.
  - *<lockin status>* The content of the status register, equivalent to `status`.
  - *<points>* The number of points acquired.

Computer interfaces

Variables

- **rs232** – The rs232 settings. *(<baud rate>, <settings>)*, where
  - *<baud rate>* The baud rate in bits per second. Valid entries are *75*, *110*, *134.5*, *150*, *300*, *600*, *1200*, *1800*, *2000*, *2400*, *4800*, *9600* and *19200*.
  - *<settings>* The sr7225 uses a 5bit register to configure the rs232 interface.
    - *'9bit'* If it is *True*, data + parity use 9 bits and 8 bits otherwise.
    - *'parity'* If it's *True*, a single parity bit is used. If it's *False* no parity bit is used.
    - *'odd parity'* If it is *True* odd parity is used and even otherwise.
    - *'echo'* If it's *True*, echo is enabled.
    - *'promt'* If it's *True*, promt is enabled.
- **gpib** – The gpib configuration. *(<channel>, <terminator>)*, where
  - *<channel>* Is the gpib communication channel, an integer between 0 and 31.
  - *<terminator>* The command terminator, either *'CR'*, *'CR echo'*, *'CRLF'*, *'CRLF echo'*, *'None'* or *'None echo'*. *'CR'* is the carriage return, *'LF'* the linefeed. When echo is on, every command or response of the gpib interface is echoed to the rs232 interface.

- **delimiter** – The response data separator. Valid entries are 13 or 32 to 125 representing the ascii value of the character in use.

  Warning: This command does **not** change the response data separator of the commands in use. Using it might lead to errors.

- **status** – The sr7225 status register.
- **status_enable** – The status enable register is used to mask the bits of the status register, which generate a service request.
- **overload_status** – The overload status register.
- **remote** – The remote mode of the front panel.

Instrument identification

Variables

- **identification** – The identification, it responds with *'7225'*.
- **revision** – The firmware revision. A multiline string.

  Warning: Not every transport can handle multiline responses.

- **version** – The firmware version.

Frontpanel

Variables **lights** – The status of the front panel lights. *True* if these are enabled, *False* otherwise.

`AC_GAIN` = **[u'0 dB', u'10 dB', u'20 dB', u'30 dB', u'40 dB', u'50 dB', u'60 dB', u'70 db', u'80 dB', u'90 dB']**

`BAUD_RATE` = **[75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 4800, 9600, 19200]**
All valid baud rates of the rs232 interface.

`CURVE_BUFFER` = **{0: u'x', 1: u'y', 2: u'r', 3: u'theta', 4: u'sensitivity', 5: u'adc1', 6: u'adc2', 7: u'7', 8: u'dac1', 9: u'da**
The definition of the curve buffer register bits. To change the curve buffer settings use the `curve_buffer_settings` attribute.

`OVERLOAD_BYTE` = **{1: u'ch1 output overload', 2: u'ch2 output overload', 3: u'y output overload', 4: u'x output overloa**
The overload byte definition.

`RS232` = **{0: u'9bit', 1: u'parity', 2: u'odd parity', 3: u'echo', 4: u'promt'}**
The rs232 settings register definition.

`SENSITIVITY_CURRENT_HIGHBW` = **[u'2 fA', u'5 fA', u'10 fA', u'20 fA', u'50 fA', u'100 fA', u'200 fA', u'500 fA', u'1 pA**

`SENSITIVITY_CURRENT_LOWNOISE` = **[u'2 fA', u'5 fA', u'10 fA', u'20 fA', u'50 fA', u'100 fA', u'200 fA', u'500 fA', u'1**

`SENSITIVITY_VOLTAGE` = **[u'2 nV', u'5 nV', u'10 nV', u'20 nV', u'50 nV', u'100 nV', u'200 nV', u'500 nV', u'1 uV', u**

`STATUS_BYTE` = **{0: u'cmd complete', 1: u'invalid cmd', 2: u'cmd param error', 3: u'reference unlock', 4: u'overload', 5**

`TIME_CONSTANT` = **[1e-05, 2e-05, 4e-05, 8e-05, 0.00016, 0.00032, 0.00064, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20**
All valid time constant values.

`auto_measure`()
Triggers the auto measure mode.

`auto_offset``()`
Triggers the auto offset mode.

`auto_phase``()`
Triggers the auto phase mode.

`auto_sensitivity``()`
Triggers the auto sensitivity mode.

When the auto sensitivity mode is triggered, the SR7225 adjustes the sensitivity so that the signal magnitude lies in between 30% and 90% of the full scale sensitivity.

`halt``()`
Halts the data acquisition.

`init_curves``()`
Initializes the curve storage memory and its status variables.

Warning: All records of previously taken curves is removed.

`lock``()`
Updates all frequency-dependent gain and phase correction parameters.

`reset``(`*complete=False*`)`
Resets the lock-in to factory defaults.

Parameters **complete** – If True all settings are reseted to factory defaults. If it's False, all settings are reseted to factory defaults with the exception of communication and LCD contrast settings.

`sensitivity`

`start_afsweep``()`
Starts a frequency and amplitude sweep.

`start_asweep``(`*start=None*, *stop=None*, *step=None*`)`
Starts a amplitude sweep.

Parameters

- **start** – Sets the start frequency.
- **stop** – Sets the target frequency.
- **step** – Sets the frequency step.

`start_fsweep``(`*start=None*, *stop=None*, *step=None*`)`
Starts a frequency sweep.

Parameters

- **start** – Sets the start frequency.
- **stop** – Sets the target frequency.
- **step** – Sets the frequency step.

`stop``()`
Stops/Pauses the current sweep.

`take_data``(`*continuously=False*`)`
Starts data acquisition.

Parameters **continuously** – If False, data is written until the buffer is full. If its True, the data buffer is used as a circular buffer. That means if data acquisition reaches the end of the buffer it continues at the beginning.

take_data_triggered(*mode*)

Starts triggered data acquisition.

Parameters **mode** – If mode is 'curve', a trigger signal starts the acquisition of a complete curve or set of curves. If its 'point', only a single data point is stored.

Implements the signal recovery sr7230 driver.

The following example shows how to use the fast curve buffer to acquire data.

:: import time from slave.transport import Socket from slave.signal_recovery import SR7230

lockin = SR7230(Socket(address=('192.168.178.1', 50000)))

lockin.fast_buffer.enabled = True # Use fast curve buffer. lockin.fast_buffer.storage_interval = 8 # Take date every 8 us. lockin.fast_buffer.length = 100000 # Store the max number of points lockin.take_data() # Start data acquisition immediately.

while lockin.acquisition_status[0] == 'on': time.sleep(0.1)

x, y = lockin.fast_buffer['x'], lockin.fast_buffer['y']

The fast buffer can store just a limited amount of variables. The standard buffer is a lot more flexible. The following examples shows how to use it to store the sensitivity, x and y values.

```
lockin.standard_buffer.enabled = True
lockin.standard_buffer.definition = 'X', 'Y', 'sensitivity'
lockin.standard_buffer.storage_interval = 1000
lockin.standard_buffer.length = 1000
lockin.take_data()

while lockin.acquisition_status[0] == 'on':
    time.sleep(0.1)

# Note: The x and y values are not stored in absolute units. They are in
# relative units compared to the chosen senitivity.
sensitivity = sr7230.standard_buffer['sensitivity']
x = sr7230.standard_buffer['x']
y = sr7230.standard_buffer['y']
```

class slave.signal_recovery.sr7230.**AmplitudeModulation**(*transport*, *protocol*)

Bases: slave.core.InstrumentBase

Represents the amplitude modulation commands.

Variables

- **center** (*float*) – The amplitude modulation center voltage. A floating point in the range -10 to 10 in volts.
- **depth** (*integer*) – The amplitude modulation depth in percent in the range 0 - 100.
- **filter** (*int*) – The amplitude modulation filter control. An integer in the range 0 - 10, where 0 is the widest bandwidth and 10 is the lowest.
- **source** – The amplitude modulation source voltage used to modulate the oscillator amplitude. Valid are 'adc1' and 'external'.
- **span** (*float*) – The amplitude modulation span voltage in volts. A float in the range -10 to 10.

Note: The sum of center and span voltage can't exceed +/- 10V.

class `slave.signal_recovery.sr7230.`**`DAC`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

Variables

- **voltage** – The user specified DAC output voltage.
- **output** – Defines which output apears on the DAC. The allowed values depend on the DAC channel. See `OUTPUT`.

`OUTPUT` = **[(u'x1', u'noise', u'ratio', u'logratio', u'equation1', u'equation2', u'user', u'demod1', u'r2'), (u'y1', u'noise', u**

class `slave.signal_recovery.sr7230.`**`Demodulator`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

Implements the dual reference mode commands.

Note: These commands only work if the lockin is in dual reference mode. See `reference_mode`.

Variables

- **x** – A float representing the X-channel output in either volt or ampere. (read only)
- **y** – A float representing the Y-channel output in either volt or ampere. (read only)
- **x_offset** – The x-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **y_offset** – The y-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **xy** – X and Y-channel output with the following format *(<x>, <y>)*. (read only)
- **r** – The signal magnitude, as float. (read only)
- **theta** – The signal phase, as float. (read only)
- **r_theta** – The magnitude and the signal phase. *(<r>, <theta>)*. (read only)
- **reference_phase** – The phase of the reference signal, a float ranging from -360 to 360 corresponding to the angle in degrees with a resolution of mili degree.
- **harmonic** – The reference harmonic mode, an integer between 1 and 128 corresponding to the first to 127 harmonics.
- **slope** – The output lowpass filter slope in dB/octave, either '6 dB', '12 dB', '18 dB' or '24 dB'.

  Note: If :attr:¸.noise_measurement‘ or `fastmode` is enabled, only '6 dB' and '12 dB' are valid.

- **time_constant** – The filter time constant. See `TIME_CONSTANT` for proper values.

Note: If `noise_measurement` is enabled, only '500 us', '1 ms', '2 ms', '5 ms' and '10 ms' are valid.

- **sensitivity** – The full-scale sensitivity. The valid entries depend on the current mode. See `sensitivity` for valid entries.

auto_offset()
Triggers the auto offset mode.

auto_phase()
Triggers the auto phase mode.

auto_sensitivity()
Triggers the auto sensitivity mode.

When the auto sensitivity mode is triggered, the SR7225 adjustes the sensitivity so that the signal magnitude lies in between 30% and 90% of the full scale sensitivity.

sensitivity

class `slave.signal_recovery.sr7230.`**DigitalPort**(*transport*, *protocol*)
Bases: `slave.core.InstrumentBase`

The digital port configuration.

Variables

- **output** – Defines which ports are configured as outputs.
- **value** – Reads the bit state of all lines but writes only to output lines.

DIGITAL_OUTPUT = {0: u'DO', 1: u'D1', 2: u'D2', 3: u'D3', 4: u'D4', 5: u'D5', 6: u'D6', 7: u'D7'}

class `slave.signal_recovery.sr7230.`**Equation**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

The equation commands.

An equation is defined as:

```
(A +/- B) * C
-------------
      D
```

Variables

- **value** – The value of the equation calculation. (read only)
- **define** – A tuple defining the equation parameter; *(<A>, <op>, <B>, <C>, <D>)* where
  - *<op>* is either '+' or '-'.
  - <A>, <B>, <C>, <D> is one of `INPUT`.
- **c1** – Equation constant c1, a float in the range -30. to 30.
- **c2** – Equation constant c2, a float in the range -30. to 30.

INPUT = [u'x1', u'y1', u'r', u'theta', u'adc1', u'adc2', u'adc3', u'adc4', u'c1', u'c2', u'0', u'1', u'frequency', u'oscillator'

class `slave.signal_recovery.sr7230.`**`FastBuffer`**(*transport*, *protocol*)
Bases: `slave.core.InstrumentBase`

Represents the fast curve buffer command group.

The fast curve buffer is similar to the `StandardBuffer`. It is less flexible but allows for the fastest data acquisition rate.

Variables

- **length** – The length of the fast curve buffer, at most 100000 can be stored.
- **enabled** – A boolean flag enabling/disabling the fast curve buffer.

  Note: Enabling the fast curve buffer disables the standard curve buffer.

- **storage_interval** – The storage interval in microseconds. The smallest value is 1.

`KEYS` **= [u'x', u'y', u'demod1', u'adc1', u'adc2', u'x2', u'y2', u'demod2']**

class `slave.signal_recovery.sr7230.`**`FrequencyModulation`**(*transport*, *protocol*, *option=None*)
Bases: `slave.core.InstrumentBase`

Represents the frequency modulation commands.

Variables

- **center_frequency** (*float*) – The center frequency of the oscillator frequency modulation. A float in the range 0. to 120e3 (250e3 if 250 kHz option is installed).
- **center_voltage** (*float*) – The center voltage of the oscillator frequency modulation. A float in the range -10 to 10.
- **filter** (*int*) – The amplitude modulation filter control. An integer in the range 0 - 10, where 0 is the widest bandwidth and 10 is the lowest.
- **span_frequency** (*float*) – The oscillator frequency modulation span frequency. A float in the range 0 up to 60e3 (125e3 if 250kHz option is installed).
- **span_voltage** (*float*) – The oscillator frequency modulation span voltage. A float in the range -10 to 10.

Note: The center frequency must be larger than the span frequency. Invalid values raise a *ValueError*.

`center_frequency`

`span_frequency`

class `slave.signal_recovery.sr7230.`**`SR7230`**(*transport*, *option=None*)
Bases: `slave.core.InstrumentBase`

Represents a Signal Recovery SR7230 lock-in amplifier.

Parameters

- **transport** – A transport object.
- **option** – Specifies if an optional card is installed. Valid are *None* or '250kHz'. This changes some frequency related limits.

Signal Channel

Variables

- **current_mode** – The current mode, either ‘off’, ‘high bandwidth’ or ‘low noise’.
- **voltage_mode** – The voltage mode, either ‘test’, ‘A’ , ‘-B’ or ‘A-B’. The ‘test’ mode corresponds to both inputs grounded.
- **demodulator_source** – It sets the source of the signal for the second stage demodulators in dual reference mode. Valid are ‘main’, ‘adc1’ and ‘tandem’.

value	description
‘main’	The main signal channel adc is used.
‘adc1’	The rear pannel auxiliary input adc1 is used as source.
‘tandem’	The demodulator 1 X-channel output is used as source.

- **fet** – The voltage mode input device control. Valid entries are ‘bipolar’ and ‘fet’, where
  - ‘bipolar’ is a bipolar device with 10kOhm input impedance. It allows for the lowest possible voltage noise.

    Note: It is not possible to use bipolar and ac coupling together.

  - ‘fet’ 10MOhm input impedance. It is the default setting.
- **shield** – The input connector shield grounding mode. Valid entries are ‘ground’ and ‘float’.
- **coupling** – The input connector coupling, either ‘ac’ or ‘dc’.
- **sensitivity** – The full-scale sensitivity. The valid entries depend on the current mode.

‘off’	‘high bandwidth’	‘low noise’
‘10 nV’	‘10 fA’	—
‘20 nV’	‘20 fA’	—
‘50 nV’	‘50 fA’	—
‘100 nV’	‘100 fA’	—
‘200 nV’	‘200 fA’	‘2 fA’
‘500 nV’	‘500 fA’	‘5 fA’
‘1 uV’	‘1 pA’	‘10 fA’
‘2 uV’	‘2 pA’	‘20 fA’
‘5 uV’	‘5 pA’	‘50 fA’
‘10 uV’	‘10 pA’	‘100 fA’
‘20 uV’	‘20 pA’	‘200 fA’
‘50 uV’	‘50 pA’	‘500 fA’
‘100 uV’	‘100 pA’	‘1 pA’
‘200 uV’	‘200 pA’	2 pA’
‘500 uV’	‘500 pA’	‘5 pA’
‘1 mV’	‘1 nA’	‘10 pA’
‘2 mV’	‘2 nA’	‘20 pA’
‘5 mV’	‘5 nA’	‘50 pA’
‘10 mV’	‘10 nA’	‘100 pA’
‘20 mV’	‘20 nA’	‘200 pA’
‘50 mV’	‘50 nA’	‘500 pA’
‘100 mV’	‘100 nA’	‘1 nA’
‘200 mV’	‘200 nA’	‘2 nA’
‘500 mV’	‘500 nA’	‘5 nA’
‘1 V’	‘1 uA’	‘10 nA’

- **ac_gain** – The gain of the signal channel amplifier. See `SR7230.AC_GAIN` for valid values.
- **ac_gain_auto** – A boolean corresponding to the ac gain automatic mode. It is *False* if the ac_gain is under manual control, and *True* otherwise.
- **line_filter** – The line filter configuration. *(<filter>, <frequency>)*, where
  - *<filter>* Is the filter mode. Valid entries are *'off'*, *'notch'*, *'double'* or *'both'*.
  - *<frequency>* Is the notch filter center frequency, either *'60Hz'* or *'50Hz'*.

Reference channel

Variables

- **reference_mode** – The instruments reference mode. Valid are 'single', 'dual harmonic' and 'dual reference'.
- **reference** – The reference input mode, either 'internal', 'ttl' or 'analog'.
- **internal_reference_channel** – In dual reference mode, selects the reference channel, which is operated in internal reference mode. Valid are 'ch1' or 'ch2'.
- **harmonic** – The reference harmonic mode, an integer between 1 and 128 corresponding to the first to 127 harmonics.
- **trigger_output** – Set's the rear pannel trigger output.

Value	Description
'curve'	A trigger signal is generated by curve buffer triggering
'reference'	A ttl signal at the reference frequency

- **reference_phase** – The phase of the reference signal, a float ranging from -360 to 360 corresponding to the angle in degrees with a resolution of mili degree.
- **reference_frequency** – A float corresponding to the reference frequency in Hz. (read only)

Note: If `reference` is not *'internal'* the reference frequency value is zero if the reference channel is unlocked.

- **virtual_reference** – A boolean enabling/disabling the virtual reference mode.

Signal Channel Output Filters

Variables

- **noise_measurement** – A boolean representing the noise measurement mode.
- **noise_buffer_length** – The length of the noise buffer in seconds. Valid are 'off', '1 s', '2 s', '3 s' and '4 s'.
- **time_constant** – The filter time constant. See `TIME_CONSTANT` for valid values.

Note: If `noise_measurement` is enabled, only '500 us', '1 ms', '2 ms', '5 ms' and '10 ms' are valid.

- **sync** – A boolean value, representing the state of the synchronous time constant mode.

- **slope** – The output lowpass filter slope in dB/octave, either ‘6 dB’, ‘12 dB’, ‘18 dB’ or ‘24 dB’.

Note: If :attr:¸.noise_measurement‘ or `fastmode` is enabled, only ‘6 dB’ and ‘12 dB’ are valid.

Signal Channel Output Amplifiers

Variables

- **x_offset** – The x-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **y_offset** – The y-channel output offset control. *(<enabled>, <range>)*, where
  - *<enabled>* A boolean enabling/disabling the output offset.
  - *<range>* The range of the offset, an integer between -30000 and 30000 corresponding to +/- 300%.
- **expand** – The expansion control, either ‘off’, ‘x’, ‘y’ or ‘both’.
- **fastmode** – Enables/disables the fastmode of the output filter. In normal mode (*False*), the instruments analog outputs are derived from the output processor. The update rate is 1 kHz.

  In fastmode, the analog outputs are derived directly from the core FPGA running the demodulator algorithms. This increases the update rate to 1 Mhz for time constants 10 us to 500 ms. It remains at 1 kHz for longer time constants.

Instrument outputs

Variables

- **x** – A float representing the X-channel output in either volt or ampere. (read only)
- **y** – A float representing the Y-channel output in either volt or ampere. (read only)
- **xy** – X and Y-channel output with the following format *(<x>, <y>)*. (read only)
- **r** – The signal magnitude, as float. (read only)
- **theta** – The signal phase, as float. (read only)
- **r_theta** – The magnitude and the signal phase. *(<r>, <theta>)*. (read only)
- **ratio** – The ratio equivalent to X/ADC1. (read only)
- **log_ratio** – The ratio equivalent to log(X/ADC1). (read only)
- **noise** – The square root of the noise spectral density measured at the Y channel output. (read only)
- **noise_bandwidth** – The noise bandwidth. (read only)
- **noise_output** – The noise output, the mean absolute value of the Y channel. (read only)
- **equation** – The equation configuration, a list of two `Equation` instances.

Internal oscillator

Variables

- **amplitude** – A float between 0. and 5. representing the oscillator amplitude in V rms.
- **amplitude_start** – Amplitude sweep start value.
- **amplitude_stop** – Amplitude sweep end value.
- **amplitude_step** – Amplitude sweep amplitude step.
- **frequency** – The oscillator frequency in Hz. Valid entries are 0. to 1.2e5.
- **frequency_start** – The frequency sweep start value.
- **frequency_stop** – The frequency sweep stop value.
- **frequency_step** – The frequency sweep step size and sweep type. *(<step>, <mode>)*, where
  - *<step>* The step size in Hz.
  - *<mode>* The sweep mode, either 'log', 'linear' or 'seek'. In 'seek' mode the sweep stops automatically when the signal magnitude exceeds 50% of full scale. It's most commonly used to set up virtual reference mode.
- **sweep_rate** – The frequency and amplitude sweep step rate in time per step in seconds. Valid entries are 0.001 to 1000. with a resolution of 0.001.
- **modulation** – The state of the oscillator amplitude/frequency modulation. Valid are *False*, 'amplitude' and 'frequency'.
- **amplitude_modulation** – The amplitude modulation commands, an instance of `AmplitudeModulation`.
- **frequency_modulation** – The frequency modulation commands, an instance of `FrequencyModulation`.

Analog Outputs

Variables **dac** – A sequence of four `DAC` instances, representing all four analog outpus.

Digital I/O

Variables **digital_ports** – The digital port configuration, an instance of `DigitalPort`.

Auxiliary Inputs

Variables

- **aux** – A `CommandSequence` instance, providing access to all four analog to digital input channel voltage readings. E.g.:

```
# prints voltage reading of first aux input channel.
print(sr7230.aux[0])
```

- **aux_trigger_mode** – The trigger modes of the auxiliary input channels. Valid are 'internal', 'external', 'burst' and 'fast burst'

mode	description
'internal'	The internal
'external'	The ADC TRIG IN connector is used to trigger readings.
'burst'	Allows sampling rates of 40 kHz, but only ADC1 and ADC2 can be used.
'fast burst'	Sampling rates up to 200 kHz are possible, but only ADC1 can be used.

Output Data Curve Buffer

Variables

- **acquisition_status** – The state of the curve acquisition. A tuple corresponding to *(<state>, <sweeps>, <status byte>, <points>)*, where
  - *<state>* is the curve acquisition state. Possible values are

Value	Description
'off'	No curve acquisition in progress.
'on'	Curve acquisition via `SR7230.take_data()` in progress.
'continuous'	Curve acquisition via `take_data_continuously()` in progress.
'halted'	Curve acquisition via `SR7230.take_data()` in progress but halted.
'continuous halted'	Curve acquisition via `take_data_continuously()` in progress but halted.

  - *<sweeps>* the number of sweeps acquired.
  - *<status byte>* the status byte, see `SR7230.status_byte`.
  - *<points>* The number of points acquired.
- **fast_buffer** – An instance of `FastBuffer`, representing the fast curve buffer related commands.
- **standard_buffer** – An instance of `FastBuffer`, representing the fast curve buffer related commands.
- **buffer_trigger_output** – The trigger output generated when buffer acquisition is running.

Value	Description
'curve'	A trigger is generated once per curve.
'point'	A trigger is generated once per point.

- **buffer_trigger_output_polarity** – The polarity of the trigger output. Valid are 'rising', 'falling'.

Computer Interfaces

Variables

- **baudrate** – The baudrate of the rs232 interface. See `BAUDRATE` for valid values.
- **delimiter** – The data delimiter. See `DELIMITER` for valid values.

Note: In normal operation, there is no need to change the delimiter because the communication is handled by the protocol. If the delimiter is changed, the *msg_data_sep* and *resp_data_sep* values of the protocol must be changed manually.

- **status** – The status byte. (read only)
- **overload_status** – The overload status. (read only)
- **ip_address** – Four integer values representing the ip address. E.g. 169.254.0.10 would translate to a tuple (169, 254, 0, 10)

Note: Setting the IP address sets the gateway and subnet mask to a default value. If non default values are required set them afterwards.

- **subnet_mask** – A tuple of four ints representing the subnet mask.
- **gateway_address** – A tuple of four ints representing the gateway address.

Instrument Identification

Variables

- **identification** – The model number *7230*. (read only)
- **version** – The firmware version. (read only)
- **date** – The last calibration date. (read only)
- **name** – The name of the lock-in amplifier, a string with up 64 chars.

Dual Mode

In dual reference mode, two demodulator stages are used instead of one. The standard commands such as *x*, *y* or *sensitivity* won't work. To access the parameters of the two stages use the `demod` attribute instead. E.g.:

```
# Set lockin into dual reference mode.
sr7230.reference_mode = 'dual'
# get x value of first demod stage (zero index based).
x = sr7230.demod[0].x
# set the sensitivity of the second demod stage.
sr7230.demod[1].sensitivity = '50 nV'
```

Variables demod – A tuple of two `Demodulator` instances. The first with index 0 represents the first demodulator, the second item with index 1 represents the second demodulator.

`AC_GAIN` = [u'0 dB', u'6 dB', u'12 dB', u'18 dB', u'24 dB', u'30 dB', u'36 dB', u'42 dB', u'48 dB', u'54 dB', u'60 dB', u'

`BAUDRATE` = [75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 4800, 9600, 19200, 38400]

`DELIMITER` = [u'\r', u' ', u'!', u'"', u'#', u'$', u'%', u'&', u"'", u'(', u')', u'*', u'+', u',', u'-', u'.', u'/', u'0', u'1', u'2', u'

`OVERLOAD_BYTE` = {0: u'x1', 1: u'y1', 2: u'x2', 3: u'y2', 4: u'adc1', 5: u'adc2', 6: u'adc3', 7: u'adc4'}

`SENSITIVITY_CURRENT_HIGHBW` = [u'10 fA', u'20 fA', u'50 fA', u'100 fA', u'200 fA', u'500 fA', u'1 pA', u'2 pA', u'5 p

`SENSITIVITY_CURRENT_LOWNOISE` = [u'2 fA', u'5 fA', u'10 fA', u'20 fA', u'50 fA', u'100 fA', u'200 fA', u'500 fA', u'1

`SENSITIVITY_VOLTAGE` = [u'10 nV', u'20 nV', u'50 nV', u'100 nV', u'200 nV', u'500 nV', u'1 uV', u'2 uV', u'5 uV', u

`STATUS_BYTE` = {0: u'command complete', 1: u'invalid command', 2: u'command parameter error', 3: u'reference unlo

TIME_CONSTANT = [u'10 us', u'20 us', u'50 us', u'100 us', u'200 us', u'500 us', u'1 ms', u'2 ms', u'5 ms', u'10 ms', u'20

auto_measure()
Triggers the auto measure mode.

auto_offset()
Triggers the auto offset mode.

auto_phase()
Triggers the auto phase mode.

auto_sensitivity()
Triggers the auto sensitivity mode.

When the auto sensitivity mode is triggered, the SR7225 adjustes the sensitivity so that the signal magnitude lies in between 30% and 90% of the full scale sensitivity.

clear_buffer()
Initialises the curve buffer and related status variables.

date

factory_defaults(*full=False*)
Resets the device to factory defaults.

Parameters full – If full is *True*, all settings are returned to factory defaults, otherwise the communication settings are not changed.

halt()
Halts curve acquisition in progress.

If a sweep is linked to curve buffer acquisition it is halted as well.

i = 125

link_afsweep()
Links dual amplitude/frequency sweep to curve buffer acquisition.

link_asweep()
Links amplitude sweep to curve buffer acquisition.

link_fsweep()
Links frequency sweep to curve buffer acquisition.

lock_ip()
Locks the ip address.

Only commands of the locked ip are accepted.

pause_afsweep()
Pauses dual frequency/amplitude sweep.

pause_asweep()
Pauses amplitude sweep.

pause_fsweep()
Pauses frequency sweep.

sensitivity

start_afsweep()
Starts a frequency and amplitude sweep.

start_asweep(*start=None*, *stop=None*, *step=None*)
Starts a amplitude sweep.

Parameters

- **start** – Sets the start frequency.
- **stop** – Sets the target frequency.
- **step** – Sets the frequency step.

start_fsweep(*start=None*, *stop=None*, *step=None*)
Starts a frequency sweep.

Parameters

- **start** – Sets the start frequency.
- **stop** – Sets the target frequency.
- **step** – Sets the frequency step.

stop()
Stops the current sweep.

take_data()
Starts data acquisition.

take_data_continuously(*trigger*, *stop*)
Starts continuous data acquisition.

Parameters

- **trigger** – The trigger condition, either 'curve' or 'point'.

Value	Description
'curve'	Each trigger signal starts a curve acquisition. The max trigger frequency in this mode is 1 kHz.
'point'	A point is stored for each trigger signal.

- **edge** – Defines wether a 'rising' or 'falling' edge is interpreted as a trigger signal.
- **stop** – The stop condition. Valid are 'buffer', 'halt', 'rising' and 'falling'.

Value	Description
'buffer'	Data acquisition stops when the number of point specified in `length` is acquired.
'halt'	Data acquisition stops when the halt command is issued.
'rising'	Data acquisition stops on the rising edge of a trigger signal.
'falling'	Data acquisition stops on the falling edge of a trigger signal.

Note: The internal buffer is used as a circular buffer.

take_data_triggered(*trigger*, *edge*, *stop*)
Configures data acquisition to start on various trigger conditions.

Parameters

- **trigger** – The trigger condition, either 'curve' or 'point'.

Value	Description
'curve'	Each trigger signal starts a curve acquisition. The max trigger frequency in this mode is 1 kHz.
'point'	A point is stored for each trigger signal.

- **edge** – Defines wether a 'rising' or 'falling' edge is interpreted as a trigger signal.

- **stop** – The stop condition. Valid are 'buffer', 'halt', 'rising' and 'falling'.

Value	Description
'buffer'	Data acquisition stops when the number of point specified in `length` is acquired.
'halt'	Data acquisition stops when the halt command is issued.
'trigger'	Takes data for the period of a trigger event. If edge is 'rising' then teh acquisition starts on the rising edge of the trigger signal and stops on the falling edge and vice versa

`unlock_ip`()
Unlocks the ip address.

`update_correction`()
Updates all frequency-dependant gain and phase correction parameters.

class `slave.signal_recovery.sr7230.`**`StandardBuffer`**(*transport*, *protocol*)
Bases: `slave.core.InstrumentBase`

Represents the standard buffer command group.

Variables

- **length** – The size of the standard curve buffer is 100000 points. These are shared equally between all define curves.
- **enabled** – A boolean flag enabling/disabling the fast curve buffer.

Note: Enabling the fast curve buffer disables the standard curve buffer.

- **storage_interval** – The storage interval in microseconds. The smallest value is 1000.
- **define** – Selects which curves should be stored. See `KEYS` for allowed values.

`KEYS` = **[u'x', u'y', u'r', u'theta', u'sensitivity', u'noise', u'ratio', u'log ratio', u'adc1', u'adc2', u'adc3', u'adc4', u'dac1',**

`define`

`event`(*value*)
Set an event marker.

If the event curve is defined and data acquisition is running, a call to event stores the value in the event curve.

`length`

3.1.11 `srs` Module

The sr830 module implements an interface for the Stanford Research Systems SR830.

The SR830 lock in amplifier is an IEEE Std. 488-2 1987 compliant device.

class `slave.srs.sr830.`**`SR830`**(*transport*)
Bases: `slave.core.InstrumentBase`

Stanford Research SR830 Lock-In Amplifier instrument class.

The SR830 provides a simple, yet powerful interface to a Stanford Research SR830 lock-in amplifier.

E.g.:

```
import visa
from slave.sr830 import SR830
# create a transport with a sr830 instrument via GPIB on channel 8
transport = visa.Instrument('GPIB::8')
# instantiate the lockin interface.
lockin = SR830(transport)
# execute a simple measurement
for i in range(100):
    print 'X:', lockin.x
    time.sleep(1)
```

`alarm` = **None**
Sets or queries the alarm state.

`amplitude` = **None**
Sets or queries the amplitude of the sine output.

`auto_gain`()
Executes the auto gain command.

`auto_offset`(*signal*)
Executes the auto offset command for the selected signal.

Parameters **i** – Can either be 'X', 'Y' or 'R'.

`auto_phase`()
Executes the auto phase command.

`auto_reserve`()
Executes the auto reserve command.

`ch1` = **None**
Reads the value of channel 1.

`ch1_display` = **None**
Set or query the channel 1 display settings.

`ch1_output` = **None**
Sets the channel1 output.

`ch2` = **None**
Reads the value of channel 2.

`ch2_display` = **None**
Set or query the channel 2 display settings.

`ch2_output` = **None**
Sets the channel2 output.

`clear`()
Clears all status registers.

`clear_on_poweron` = **None**
Enables or disables the clearing of the status registers on poweron.

`coupling` = **None**
Sets or queries the input coupling.

`data_points` = **None**
Queries the number of data points stored in the internal buffer.

`delayed_start`()
Starts data storage after a delay of 0.5 sec.

`fast_mode` = **None**
Sets or queries the data transfer mode. .. note:

```
Do not use :class:`~SR830.start() to execute the scan, use
:class:`~SR830.delayed_start instead.
```

`filter` = **None**
Sets or queries the input line notch filter status.

`frequency` = **None**
Sets or queries the internal reference frequency.

`ground` = **None**
Sets or queries the input shield grounding.

`harmonic` = **None**
Sets or queries the detection harmonic.

`idn` = **None**
Queries the device identification string

`input` = **None**
Sets or queries the input configuration.

`key_click` = **None**
Sets or queries the key click state.

`output_interface` = **None**
Sets or queries the output interface.

`overide_remote` = **None**
Sets the remote mode override.

`pause`()
Pauses data storage.

`phase` = **None**
Sets and queries the reference phase

`r` = **None**
Reads the value of r.

`r_offset_and_expand` = **None**
Sets or queries the x value offset and expand

`recall_setup`(*id*)
Recalls the lock-in setup from the setup buffer.

Parameters **id** – Represents the buffer id (0 < buffer < 10). If no lock-in setup is stored under this id, recalling results in an error in the hardware.

`reference` = **None**
Sets or queries the reference source

`reference_trigger` = **None**
Sets or triggers the reference trigger mode.

`reserve` = **None**
Sets or queries the dynamic reserve.

`reset_buffer`()
Resets internal data buffers.

reset_configuration()
Resets the SR830 to it's default configuration.

save_setup(*id*)
Saves the lock-in setup in the setup buffer.

send_mode = None
The send command sets or queries the end of buffer mode. .. note:

```
If loop mode is used, the data storage should be paused to avoid
confusion about which point is the most recent.
```

sensitivity = None
Sets or queries the sensitivity in units of volt.

slope = None
Sets or queries the low-pass filter slope.

snap(**args*)
Records up to 6 parameters at a time.

Parameters **args** – Specifies the values to record. Valid ones are 'X', 'Y', 'R', 'theta', 'AuxIn1', 'AuxIn2', 'AuxIn3', 'AuxIn4', 'Ref', 'CH1' and 'CH2'. If none are given 'X' and 'Y' are used.

start()
Starts or resumes data storage.

state = None
Queries or sets the state of the frontpanel.

sync = None
Sets or queries the synchronous filtering mode.

theta = None
Reads the value of theta.

time_constant = None
Sets or queries the time constant in seconds.

trace(*buffer*, *start*, *length=1*)
Reads the points stored in the channel buffer.

Parameters

- **buffer** – Selects the channel buffer (either 1 or 2).
- **start** – Selects the bin where the reading starts.
- **length** – The number of bins to read.

trigger()
Emits a trigger event.

x = None
Reads the value of x.

x_offset_and_expand = None
Sets or queries the x value offset and expand.

y = None
Reads the value of y.

y_offset_and_expand = None
Sets or queries the x value offset and expand.

class `slave.srs.sr850.`**`Cursor`**(*transport*, *protocol*)

Bases: `slave.core.InstrumentBase`

Represents the SR850 cursor of the active display.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

Note: The cursor commands are only effective if the active display is a chart display.

Variables

- **seek_mode** – The cursor seek mode, valid are 'max', 'min' and 'mean'.
- **width** – The cursor width, valid are 'off', 'narrow', 'wide' and 'spot'.
- **vertical_division** – The vertical division of the active display. Valid are 8, 10 or *None*.
- **control_mode** – The cursor control mode. Valid are 'linked', 'separate'.
- **readout_mode** – The cursor readout mode. Valid are 'delay', 'bin', 'fsweep' and 'time'.
- **bin** – The cursor bin position of the active display. It represents the center of the cursor region. This is not the same as the cursor readout position. To get the actual cursor location, use `Display.cursor`.

`move`()

Moves the cursor to the max or min position of the data, depending on the seek mode.

`next_mark`()

Moves the cursor to the next mark to the right.

`previous_mark`()

Moves the cursor to the next mark to the left.

class `slave.srs.sr850.`**`Display`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

Represents a SR850 display.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The display id.

Note: The SR850 will generate an error if one tries to set a parameter of an invisible display.

Variables

- **type** – The display type, either 'polar', 'blank', 'bar' or 'chart'.
- **trace** – The trace number of the displayed trace.
- **range** – The displayed range, a float between 10^-18 and 10^18.

  Note: Only bar and chart displays are affected.

- **offset** – The display center value in units of the trace in the range 10^-12 to 10^12.
- **horizontal_scale** – The display's horizontal scale. Valid are '2 ms', '5 ms', '10 ms', '20 ms', '50 ms', '0.1 s', '0.2 s', '0.5 s', '1 s', '2 s', '5 s', '10 s', '20 s', '50 s', '1 min', '100 s', '2 min', '200 s', '5 min', '500 s', '10 min', '1 ks', '20 min', '2 ks', '1 h', '10 ks', '3 h', '20 ks', '50 ks', '100 ks' and '200 ks'.
- **bin** – The bin number at the right edge of the chart display. (read only)

  Note: The selected display must be a chart display.

- **cursor** – The cursor position of this display (read only), represented by the tuple *(<horizontal>, <vertical>)*, where
  - *<horizontal>* is the horizontal position in bin, delay, time or sweep frequency.
  - *<vertical>* is the vertical position.

class `slave.srs.sr850.`**`FitParameters`**(*transport*, *protocol*)

Bases: `slave.core.InstrumentBase`

The calculated fit parameters.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

The meaning of the fit parameters depends on the fit function used to obtain them. These are

Function	Definition

line *y = a + b * (t - t0)* exp *y = a * exp(-(t - t0) / b) + c* gauss *y = a * exp(0.5 * (t / b)^2) + c* ======== ==============================

Variables

- **a** – The a parameter.

Function	Meaning
linear	Vertical offset in trace units.
exp	Amplitude in trace units.
gauss	Amplitude in trace units.

- **b** – The b parameter.

Function	Meaning
linear	Slope in trace units per second.
exp	Time constant in time.
gauss	Line width in time.

- **c** – The c parameter.

Function	Meaning
linear	Unused.
exp	Vertical offset in trace units.
gauss	Vertical offset in trace units.

- **t0** – The t0 parameter.

Function	Meaning
linear	Horizontal offset in time.
exp	Horizontal offset in time.
gauss	Peak center position in time.

class `slave.srs.sr850.`**`Mark`**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

A SR850 mark.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The mark index.

`active`
The active state of the mark.

Returns True if the mark is active, False otherwise.

`bin`
The bin index of this mark.

Returns An integer bin index or None if the mark is inactive.

`label`
The label string of the mark.

Note: The label should not contain any whitespace charactes.

class `slave.srs.sr850.`**`MarkList`**(*transport*, *protocol*)
Bases: `slave.core.InstrumentBase`

A sequence like structure holding the eight SR850 marks.

`active`()
The indices of the active marks.

class `slave.srs.sr850.`**`Output`**(*transport*, *protocol*, *idx*)
Bases: `slave.core.InstrumentBase`

Represents a SR850 analog output.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The output id.

Variables

- **mode** – The analog output mode. Valid are ‘fixed’, ‘log’ and ‘linear’.
- **voltage** – The output voltage in volt, in the range -10.5 to 10.5.
- **limits** – The output voltage limits and offset, represented by the following tuple *(<start>, <stop>, <offset>)*, where
  - *<start>* is the start value of the sweep. A float between 1e-3 and 21.
  - *<stop>* is the stop value of the sweep. A float between 1e-3 and 21.

– *<offset>* is the sweep offset value. A float between -10.5 and 10.5.

Note: If the output is in fixed mode, setting the limits will generate a lock-in internal error.

class `slave.srs.sr850.`**`SR850`**(*transport*)

Bases: `slave.iec60488.IEC60488`, `slave.iec60488.PowerOn`

A Stanford Research SR850 lock-in amplifier.

Parameters **transport** – A transport object.

Reference and Phase Commands

Variables

- **phase** – The reference phase shift. A float between -360. and 719.999 in degree.
- **reference_mode** – The reference source mode, either 'internal', 'sweep' or 'external'.
- **frequency** – The reference frequency in Hz. A float between 0.001 and 102000.

Note: For harmonics greater than 1, the sr850 limits the max frequency to 102kHz / harmonic.

- **frequency_sweep** – The type of the frequency sweep, either 'linear' or 'log', when in 'internal' reference_mode.
- **start_frequency** – The start frequency of the internal frequency sweep mode. A float in the range 0.001 to 102000.
- **stop_frequency** – The stop frequency of the internal frequency sweep mode. A float in the range 0.001 to 102000.
- **reference_slope** – The reference slope in the external mode. Valid are:

'sine'	sine zero crossing
'rising'	TTL rising edge
'falling'	TTL falling edge

- **harmonic** – The detection harmonic, an integer between 1 and 32767.
- **amplitude** – The amplitude of the sine output in volts. Valid entries are floats in the range 0.004 to 5.0 with a resolution of 0.002.

Reference and Phase Commands

Variables

- **input** – The input configuration, either 'A', 'A-B' or 'I'.
- **input_gain** – The conversion gain of the current input. Valid are '1 MOhm' and '100 MOhm'.
- **ground** – The input grounding, either 'float' or 'ground'.
- **coupling** – The input coupling, either 'AC' or 'DC'.
- **filter** – The input line filter configuration. Valid are 'unfiltered', 'notch', '2xnotch' and 'both'.

- **sensitivity** – The input sensitivity in volt or microamps. Valid are 2e-9, 5e-9, 10e-9, 20e-9, 50e-9, 100e-9, 200e-9, 500e-9, 1e-6, 2e-6, 5e-6, 10e-6, 20e-6, 50e-6, 100e-6, 200e-6, 500e-6, 1e-3, 2e-3, 5e-3, 10e-3, 20e-3, 50e-3, 100e-3, 200e-3, 500e-3, and 1.
- **reserve_mode** – The reserve mode configuration. Valid entries are 'max', 'manual' and 'min'.
- **reserve** – The dynamic reserve. An Integer between 0 and 5 where 0 is the minimal reserve available.
- **time_constant** – The time constant in seconds. Valid are 10e-6, 30e-6, 100e-6, 300e-6, 1e-3, 3e-3, 10e-3, 30e-3, 100e-3, 300e-3, 1., 3., 10, 30, 100, 300, 1e3, 3e3, 10e3, and 30e3.
- **filter_slope** – The lowpass filter slope in db/oct. Valid are 6, 12, 18 and 24.
- **sync_filter** – The state of the syncronous filtering, either True or False.

Note: Syncronous filtering is only vailable if the detection frequency is below 200Hz

Output and Offset Commands

Variables

- **ch1_display** – The channel 1 frontpanel output source. Valid are 'x', 'r', 'theta', 'trace1', 'trace2', 'trace3' and 'trace4'.
- **ch2_display** – The channel 2 frontpanel output source. Valid are 'y', 'r', 'theta', 'trace1', 'trace2', 'trace3' and 'trace4'.
- **x_offset_and_expand** – The output offset and expand of the x quantity. A tuple (<offset>, <expand>), where
  - <offset> the offset in percent, in the range -105.0 to 105.0.
  - <expand> the expand in the range 1 to 256.
- **y_offset_and_expand** – The output offset and expand of the y quantity. A tuple (<offset>, <expand>), where
  - <offset> the offset in percent, in the range -105.0 to 105.0.
  - <expand> the expand in the range 1 to 256.
- **r_offset_and_expand** – The output offset and expand of the r quantity. A tuple (<offset>, <expand>), where
  - <offset> the offset in percent, in the range -105.0 to 105.0.
  - <expand> the expand in the range 1 to 256.

Trace and Scan Commands

Variables

- **traces** – A sequence of four `Trace` instances.
- **scan_sample_rate** – The scan sample rate, valid are 62.5e-3, 125e-3, 250e-3, 500e-3, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 in Hz or 'trigger'.

- **scan_length** – The scan length in seconds. It well be set to the closest possible time. The minimal scan time is 1.0 seconds. The maximal scan time is determined by the scanning sample rate and the number of stored traces.

Traces	Buffer Size
1	64000
2	32000
3	48000
4	16000

- **scan_mode** – The scan mode, valid are 'shot' or 'loop'.

Display and Scale Commands

Variables

- **active_display** – Sets the active display. Valid are 'full', 'top' or 'bottom'.
- **selected_display** – Selects the active display, either 'top' or 'bottom'. If the active display 'full' it is already selected.
- **screen_format** – Selects the screen format. Valid are
  - 'single', The complete screen is used.
  - 'dual', A up/down split view is used.
- **monitor_display** – The monitor display mode, valid are 'settings' and 'input/output'.
- **full_display** – An instance of `Display`, representing the full display.
- **top_display** – An instance of `Display`, representing the top display.
- **bottom_display** – An instance of `Display`, representing the bottom display.

Cursor Commands

Variables **cursor** – The cursor of the active display, an instance of `Cursor`.

Aux Input and Output Commands

Variables

- **aux_input** – A sequence of four read only items representing the aux inputs in volts.
- **aux_input** – A sequence of four `Output` instances, representing the analog outputs.
- **start_on_trigger** (*bool*) – If start on trigger is True, a trigger signal will start the scan.

Variables **marks** – An instance of `MarkList`, a sequence like structure giving access to the SR850 marks.

Math Commands

Variables

- **math_operation** – Sets the math operation used by the `calculate()` operation. Valid are '+', '-', '*', '/', 'sin', 'cos', 'tan', 'sqrt', '^2', 'log' and '10^x'.

- **math_argument_type** – The argument type used in the `calculate()` method, either 'trace' or 'constant'.
- **math_constant** (*float*) – Specifies the constant value used by the `calculate()` operation if the `math_argument_type` is set to 'constant'.
- **math_trace_argument** – Specifies the trace number used by the `calculate()` operation if the `math_argument_type` is set to 'trace'.
- **fit_function** – The function used to fit the data, either 'linear', 'exp' or 'gauss'.
- **fit_params** – An instance of `FitParameter` used to access the fit parameters.
- **statistics** – An instance of `Statistics` used to access the results of the statistics calculation.

Store and Recall File Commands

Variables **filename** – The active filename.

Warning: The SR850 supports filenames with up to eight characters and an optional extension with up to three characters. The filename must be in the DOS format. Slave does not validate this yet.

Setup Commands

Variables

- **interface** – The communication interface in use, either 'rs232' or 'gpib'.
- **overide_remote** (*bool*) – The gpib remote overide mode.
- **key_click** (*bool*) – Enables/disables the key click.
- **alarm** (*bool*) – Enables/disables the alarm.
- **hours** (*int*) – The hours of the internal clock. An integer in the range 0 - 23.
- **minutes** (*int*) – The minutes of the internal clock. An integer in the range 0 - 59.
- **seconds** (*int*) – The seconds of the internal clock. An integer in the range 0 - 59.
- **days** (*int*) – The days of the internal clock. An integer in the range 1 - 31.
- **month** (*int*) – The month of the internal clock. An integer in the range 1 - 12.
- **years** (*int*) – The year of the internal clock. An integer in the range 0 - 99.
- **plotter_mode** – The plotter mode, either 'rs232' or 'gpib'.
- **plotter_baud_rate** – The rs232 plotter baud rate.
- **plotter_address** – The gpib plotter address, an integer between 0 and 30.
- **plotting_speed** – The plotting speed mode, either 'fast' or 'slow'.
- **trace_pen_number** (*int*) – The trace pen number in the range 1 to 6.
- **grid_pen_number** (*int*) – The grid pen number in the range 1 to 6.
- **alphanumeric_pen_number** (*int*) – The alphanumeric pen number in the range 1 to 6.
- **cursor_pen_number** (*int*) – The cursor pen number in the range 1 to 6.

- **printer** – The printer type. Valid are 'epson', 'hp' and 'file'.

Data Transfer Commands

Variables

- **x** (*float*) – The in-phase signal in volt (read only).
- **y** (*float*) – The out-of-phase signal in volt (read only).
- **r** (*float*) – The amplitude signal in volt (read only).
- **theta** (*float*) – The in-phase signal in degree (read only).
- **fast_mode** – The fast mode. When enabled, data is automatically transmitted over the gpib interface. Valid are 'off', 'dos' or 'windows'.

Note: Use `SR850.start()` with *delay=True* to start the scan.

Warning: When enabled, the user is responsible for reading the transmitted values himself.

Interface Commands

Variables **access** – The frontpanel access mode.

Value	Description
'local'	Frontpanel operation is allowed.
'remote'	Frontpanel operation is locked out except the *HELP* key. It returns the lock-in into the local state.
'lock-out'	All frontpanel operation is locked out.

Status Reporting Commands

Variables

- **status** – The status byte register, a dictionary with the following entries

Key	Description
SCN	No scan in progress.
IFC	No command execution in progress.
ERR	An enabled bit in the error status byte has been set.
LIA	An enabled bit in the LIA status byte has been set.
MAV	The message available byte.
ESB	An enabled bit in the event status byte has been set.
SRQ	A service request has occured.

- **event_status** – The event status register, a dictionairy with the following keys:

Key	Description
'INP'	An input queue overflow occured.
'QRY'	An output queue overflow occured.
'EXE'	A command execution error occured.
'CMD'	Received an illegal command.
'URQ'	A key press or knob rotation occured.
'PON'	Set by power on.

If any bit is *True* and enabled, the 'ESB' bit in the `status` is set to *True*.

- **event_status_enable** – The event status enable register. It has the same keys as `SR850.event_status`.
- **error_status** – The event status register, a dictionairy with the following keys

Key	Description
'print/plot error'	A printing/plotting error occured.
'backup error'	Battery backup failed.
'ram error'	Ram memory test failed.
'disk error'	A disk or file operation failed.
'rom error'	Rom memory test failed.
'gpib error'	GPIb fast data transfer aborted.
'dsp error'	DSP test failed.
'math error'	Internal math error occured.

If any bit is *True* and enabled, the 'ERR' bit in the `status` is set to *True*.

- **error_status_enable** – The error status enable register. It has the same keys as `SR850.error_status`.
- **lia_status** – The LIA status register, a dictionairy with the following keys

Key	Description
'input overload'	An input or reserve overload occured.
'filter overload'	A filter overload occured.
'output overload'	A output overload occured.
'reference unlock'	A reference unlock is detected.
'detection freq change'	The detection frequency changed its range.
'time constant change'	The time constant changed indirectly, either by changing the frequency range, dynamic reserve or filter slope.
'triggered'	A trigger event occured.
'plot'	Completed a plot.

`LIA_BYTE` **= {0: u'input overload', 1: u'filter overload', 2: u'output overload', 3: u'reference unlock', 4: u'detection freq**

`auto_gain`()
Performs a auto gain action.

`auto_offset`(*quantity*)
Automatically offsets the given quantity.

Parameters **quantity** – The quantity to offset, either 'x', 'y' or 'r'

`auto_phase`()
Automatically selects the best matching phase.

`auto_reserve`()
Automatically selects the best dynamic reserve.

`auto_scale`()
Autoscales the active display.

Note: Just Bar and Chart displays are affected.

`calculate`(*operation=None*, *trace=None*, *constant=None*, *type=None*)
Starts the calculation.

The calculation operates on the trace graphed in the active display. The math operation is defined by the `math_operation`, the second argument by the `math_argument_type`.

For convenience, the operation and the second argument, can be specified via the parameters

Parameters

- **operation** – Set's the math operation if not *None*. See `math_operation` for details.
- **trace** – If the trace argument is used, it sets the `math_trace_argument` to it and sets the `math_argument_type` to 'trace'
- **constant** – If constant is not *None*, the `math_argument_type` is set to 'constant'
- **type** – If type is not *None*, the `math_argument_type` is set to this value.

E.g. instead of:

```
lockin.math_operation = '*'
lockin.math_argument_type = 'constant'
lockin.math_constant = 1.337
lockin.calculate()
```

one can write:

```
lockin.calculate(operation='*', constant=1.337)
```

Note: Do not use trace, constant and type together.

Note: The calculation takes some time. Check the status byte to see when the operation is done. A running scan will be paused until the operation is complete.

Warning: The SR850 will generate an error if the active display trace is not stored when the command is executed.

`calculate_statistics`(*start*, *stop*)
Starts the statistics calculation.

Parameters

- **start** – The left limit of the time window in percent.
- **stop** – The right limit of the time window in percent.

Note: The calculation takes some time. Check the status byte to see when the operation is done. A running scan will be paused until the operation is complete.

Warning: The SR850 will generate an error if the active display trace is not stored when the command is executed.

delete_mark()
Deletes the nearest mark to the left.

fit(*range*, *function=None*)
Fits a function to the active display's data trace within a specified range of the time window.

E.g.:

```
# Fit's a gaussian to the first 30% of the time window.
lockin.fit(range=(0, 30), function='gauss')
```

Parameters

- **start** – The left limit of the time window in percent.
- **stop** – The right limit of the time window in percent.
- **function** – The function used to fit the data, either 'line', 'exp', 'gauss' or None, the default. The configured fit function is left unchanged if function is None.

Note: Fitting takes some time. Check the status byte to see when the operation is done. A running scan will be paused until the fitting is complete.

Warning: The SR850 will generate an error if the active display trace is not stored when the fit command is executed.

pause()
Pauses a scan and all sweeps in progress.

place_mark()
Places a mark in the data buffer at the next sample.

Note: This has no effect if no scan is running.

plot_all()
Generates a plot of all data displays.

plot_cursors()
Generates a plot of the enabled cursors.

plot_trace()
Generates a plot of the data trace.

print_screen()
Prints the screen display with an attached printer.

recall(*mode=u'all'*)
Recalls from the file specified by `filename`.

Parameters **mode** – Specifies the recall mode.

Value	Description
'all'	Recalls the active display's data trace, the trace definition and the instrument state.
'state'	Recalls the instrument state.

reset_scan()

Resets a scan.

Warning: This will erase the data buffer.

save(*mode=u'all'*)

Saves to the file specified by `filename`.

Parameters mode – Defines what to save.

Value	Description
'all'	Saves the active display's data trace, the trace definition and the instrument state.
'data'	Saves the active display's data trace.
'state'	Saves the instrument state.

smooth(*window*)

Smooths the active display's data trace within the time window of the active chart display.

Parameters window – The smoothing window in points. Valid are 5, 11, 17, 21 and 25.

Note: Smoothing takes some time. Check the status byte to see when the operation is done. A running scan will be paused until the smoothing is complete.

Warning: The SR850 will generate an error if the active display trace is not stored when the smooth command is executed.

snap(**args*)

Records multiple values at once.

It takes two to six arguments specifying which values should be recorded together. Valid arguments are 'x', 'y', 'r', 'theta', 'aux1', 'aux2', 'aux3', 'aux4', 'frequency', 'trace1', 'trace2', 'trace3' and 'trace4'.

snap is faster since it avoids communication overhead. 'x' and 'y' are recorded together, as well as 'r' and 'theta'. Between these pairs, there is a delay of approximately 10 us. 'aux1', 'aux2', 'aux3' and 'aux4' have am uncertainty of up to 32 us. It takes at least 40 ms or a period to calculate the frequency.

E.g.:

```
lockin.snap('x', 'theta', 'trace3')
```

start(*delay=False*)

Starts or resumes a scan.

Parameters delay (*bool*) – If *True*, starts the scan with a delay of 0.5 seconds.

Note: It has no effect if the scan is already running.

class slave.srs.sr850.**Statistics**(*transport*, *protocol*)

Bases: `slave.core.InstrumentBase`

Provides access to the results of the statistics calculation.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.

Variables

- **mean** – The mean value.
- **standard_deviation** – The standart deviation.
- **total_data** – The sum of all the data points within the range.
- **time_delta** – The time delta of the range.

class `slave.srs.sr850.`**`Trace`**(*transport*, *protocol*, *idx*)

Bases: `slave.core.InstrumentBase`

Represents a SR850 trace.

Parameters

- **transport** – A transport object.
- **protocol** – A protocol object.
- **idx** – The trace id.

Variables

- **value** (*float*) – The value of the trace (read only).
- **traces** – A sequence of four traces represented by the following tuple *(<a>, <b>, <c>, <store>)* where
  - *<a>, <b>, <c>* define the trace which is calculated as *<a>* * *<b>* / *<c>*. Each one of them is one of the following quantities '1', 'x', 'y', 'r', 'theta', 'xn', 'yn', 'rn', 'Al1', 'Al2', 'Al3', 'Al4', 'F', 'x**2', 'y**2', 'r**2', 'theta**2', 'xn**2', 'yn**2', 'rn**2', 'Al1**2', 'Al2**2', 'Al3**2', 'Al4**2' or 'F**2'
  - *<store>* is a boolean defining if the trace is stored.

  Traces support a subset of the slicing notation. To get the number of points stored, use the builtin `len()` method. E.g.:

```
# get point at bin 17.
print trace[17]
# get point 17, 18 and 19
print trace[17:20]
```

  If the upper bound exceeds the number of store points, an internal lock-in error is generated.

3.1.12 `types` Module

Contains the type factory classes used to load and dump values to string.

The type module contains several type classes used by the `Command` class to load and dump values.

Custom Types

The `Command` class needs an object with three methods:

- `load(value)()`, takes the value and returns the userspace representation.
- `dump(value)()`, returns the device space representation of value.
- `simulate()`, generates a valid user space value.

The abstract `Type` class implements this interface but most of the time it is sufficient to inherit from `SingleType`.

`SingleType` provides a default implementation, as well as three hooks to modify the behaviour.

class `slave.types.`**`Boolean`**(*fmt=None*)
Bases: `slave.types.SingleType`

Represents a Boolean type.

Parameters **fmt** – Boolean uses a default format string of *'{0:d}'*. This means *True* will get serialized to *'1'* and *False* to *'0'*.

`simulate`()

class `slave.types.`**`Enum`**(**args*, ***kw*)
Bases: `slave.types.Mapping`

Represents a one to one mapping to an integer range.

`load`(*value*)

class `slave.types.`**`Float`**(*min=None*, *max=None*, **args*, ***kw*)
Bases: `slave.types.Range`

Represents a floating point type.

`simulate`()

class `slave.types.`**`Integer`**(*min=None*, *max=None*, **args*, ***kw*)
Bases: `slave.types.Range`

Represents an integer type.

`simulate`()
Generates a random integer in the available range.

class `slave.types.`**`Mapping`**(*mapping*)
Bases: `slave.types.SingleType`

Represents a one to one mapping of keys and values.

The Mapping represents a one to one mapping of keys and values. The keys represent the value on the user side, and the values represent the value on the instrument side, e.g:

```
type_ = Mapping({'UserValue': 'DeviceValue'})
print type_.load('DeviceValue')  # prints 'UserValue'
print type_.dump('UserValue')  # prints 'DeviceValue'
```

Note:

1.Keys do not need to be strings, they just need to be hashable.

2.Values will be converted to strings using *str()*.

`load`(*value*)

`simulate`()
Returns a randomly chosen key of the mapping.

class `slave.types.`**`Range`**(*min=None*, *max=None*, **args*, ***kw*)
Bases: `slave.types.SingleType`

Abstract base class for types representing ranges.

Parameters

- **min** – The minimal included value.
- **max** – The maximal included value.

The Range base class extends the `SingleType` class with a range checking validation.

class `slave.types.`**`Register`**`(`*mapping*`)`
Bases: `slave.types.SingleType`

Represents a binary register, where bits are mapped to a key.

Parameters **mapping** – The mapping defines the mapping between bits and keys, e.g.

```
mapping = {
    0: 'First bit',
    1: 'Second bit',
}
reg = Register(mapping)
```

`load``(`*value*`)`

`simulate``()`
Returns a dictionary representing the mapped register with random values.

class `slave.types.`**`Set`**`(`**args*, ***kw*`)`
Bases: `slave.types.Mapping`

Represents a one to one mapping of each value to its string representation.

class `slave.types.`**`SingleType`**`(`*fmt=None*`)`
Bases: `slave.types.Type`

A simple yet easily customizable implementation of the Type interface

Parameters **fmt** – A format string used in `__serialize__()` to convert the value to string. Advanced string formatting syntax is used. Default: *'{0}'*

The SingleType provides a default implementation of the `Type` interface. It provides three hooks to modify it's behavior.

- `__convert__()`
- `__serialize__()`
- `__validate__()`

Only `__convert__()` is required. It should convert the value to the represented python type. Both `__serialize__()` and `__validate__()` have a default implementation, which can be overwritten to provide custom behaviour.

`dump``(`*value*`)`
Dumps the value to string.

Returns Returns the stringified version of the value.

Raises TypeError, ValueError

`load``(`*value*`)`
Create the value from a string.

Returns The value loaded from a string.

Raises TypeError

class `slave.types.`**`String`**(*min=None*, *max=None*, **args*, ***kw*)

Bases: `slave.types.SingleType`

Represents a string type.

Parameters

- **min** – Minimum number of characters required.
- **max** – Maximum number of characters allowed.

`simulate`()

Returns a randomly constructed string.

Simulate randomly constructs a string with a length between min and max. If min is not present, a minimum length of 1 is assumed, if max is not present a maximum length of 10 is used.

class `slave.types.`**`Type`**

Bases: `future.types.newobject.newobject`

The type class defines the interface for all type factory classes.

`dump`(*value*)

`load`(*value*)

`simulate`()

Return a valid, randomly calculated value.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*