python-saleae Documentation

Release 0.1.0

Pat Pannuto

Dec 08, 2018

Contents

This library implements the control protocol for the Saleae Logic Analyzer. It is based off of the documentation and example here: https://github.com/saleae/SaleaeSocketApi

IMPORTANT: You must enable the 'Remote Scripting Server' in Saleae. Click on "Options" in the top-right, the "Developer" tab, and check "Enable scripting socket server". This should not require a restart.

This library requires Saleae Logic 1.2.x or greater. Unfortunately there is no way to check the version of Logic running using the scripting protocol so this is difficult to check at runtime.

Currently, this is basically a direct mapping of API calls with some small sanity checking and conveniences. It has not been extensively tested beyond my immediate needs, but it also should not have any known problems.

To get a feel for how the library works and what it can do, try the built-in demo:

```
#!/usr/bin/env python3
import saleae
saleae.demo()
```

Issues, updates, pull requests, etc should be directed to github.

CHAPTER 1

Installation

The easiest method is to simply use pip:

```
(sudo) pip install saleae
```

CHAPTER 2

Usage

```
import saleae
s = saleae.Saleae()
s.capture_to_file('/tmp/test.logicdata')
```

CHAPTER 3

The Saleae class

class saleae.**Saleae**(*host='localhost'*, *port=10429*, *quiet=False*)

exception CommandNAKedError

exception ImpossibleSettings

exception SaleaeError

capture_start()
Start a new capture and immediately return.

capture_start_and_wait_until_finished()
Convenience method that blocks until capture is complete.

```
>>> s.set_capture_seconds(.5)
>>> s.capture_start_and_wait_until_finished()
>>> s.is_processing_complete()
True
```

capture_stop()
Stop a capture and return whether any data was captured.

Returns True if any data collected, False otherwise

```
>>> s.set_capture_seconds(5)
>>> s.capture_start()
>>> time.sleep(1)
>>> s.capture_stop()
True
```

export_analyzer(*analyzer_index*, *save_path*, *wait_for_processing=True*, *data_response=False*)
Export analyzer index N and save to absolute path save_path. The analyzer must be finished processing

export_data2(*file_path_on_target_machine*, *digital_channels=None*, *analog_channels=None*, *time_span=None*, *format='csv'*, ***export_args*)

Export command: EXPORT_DATA2, <filename>, [ALL_CHANNELS|SPECIFIC_CHANNELS, [DIGITAL_ONLY|ANALOG_ONLY|ANALOG_AND_DIGITAL], <channel index> [ANALOG|DIGITAL], ..., <channel index> [ANALOG|DIGITAL]], [ALL_TIME|TIME_SPAN, <(double)start>, <(double)end>], [BINARY, <binary settings>|CSV, <csv settings>|VCD|MATLAB, <matlab settings>]

```
>>> s.export_data2('/tmp/test.csv')
```

get_active_channels()
Get the active digital and analog channels.

Returns A 2-tuple of lists of integers, the active digital and analog channels respectively

```
>>> s.get_active_channels()
([0, 1, 2, 3], [0])
```

get_active_device()
Get the current active Saleae device.

Returns A `saleae.ConnectedDevice` object for the active Saleae

```
>>> s.get_active_device() #doctest:+ELLIPSIS
<saleae.ConnectedDevice #1 LOGIC_4_DEVICE Logic 4 (...) **ACTIVE**>
```

get_all_sample_rates()
Get available sample rate combinations for the current performance level and channel combination.

```
>>> s.get_all_sample_rates()
[(12000000, 6000000), (12000000, 125000), (12000000, 5000), (12000000, 1000),␣
↪(12000000, 100), (12000000, 10), (12000000, 0), (6000000, 0), (3000000, 0),␣
↪(1000000, 0)]
```

get_analyzers()
Return a list of analyzers currently in use, with indexes.

get_bandwidth(*sample_rate*, *device=None*, *channels=None*)
Compute USB bandwidth for a given configuration.

Must supply sample_rate because Saleae API has no get_sample_rate method.

```
>>> s.get_bandwidth(s.get_all_sample_rates()[0])
96000000
```

get_capture_pretrigger_buffer_size()
The number of samples saleae records before the trigger.

Returns An integer number descripting the pretrigger buffer size

```
>>> s.get_capture_pretrigger_buffer_size() #doctest:+ELLIPSIS
1...
```

get_connected_devices()
Get a list of attached Saleae devices.

Note, this will never be an empty list. If no actual Saleae devices are connected, then Logic will return the four fake devices shown in the example.

Returns A list of `saleae.ConnectedDevice` objects

```
>>> s.get_connected_devices() #doctest:+ELLIPSIS
[<saleae.ConnectedDevice #1 LOGIC_4_DEVICE Logic 4 (...) **ACTIVE**>, <saleae.
↪ConnectedDevice #2 LOGIC_8_DEVICE Logic 8 (...)>, <saleae.ConnectedDevice
↪#3 LOGIC_PRO_8_DEVICE Logic Pro 8 (...)>, <saleae.ConnectedDevice #4 LOGIC_
↪PRO_16_DEVICE Logic Pro 16 (...)>]
```

get_digital_voltage_options()
Get a list of available digital I/O voltage thresholds for the device and the currently active selection.

This feature is only supported on the original Logic 16, Logic Pro 8, and Logic Pro 16.

```
>>> s.get_digital_voltage_options() #doctest:+SKIP
[(0, '1.2 Volts', <DigitalVoltageFlags.Selected: 1>), (1, '1.8 Volts',
↪<DigitalVoltageFlags.NotSelected: 0>), (2, '3.3+ Volts',
↪<DigitalVoltageFlags.NotSelected: 0>)]
```

get_performance()
Get performance value. Performance controls USB traffic and quality.

Returns A `saleae.PerformanceOption`

```
>>> s.get_performance() #doctest:+SKIP
<PerformanceOption.Full: 100>
```

is_analyzer_complete(*analyzer_index*)
check to see if analyzer with index N has finished processing.

static kill_logic(*kill_all=False*)
Attempts to find and kill running Saleae Logic software

static launch_logic(*timeout=5*, *quiet=False*)
Attempts to open Saleae Logic software

reset_active_channels()
Set all channels to active.

```
>>> s.reset_active_channels()
```

select_active_device(*device_index*)

```
>>> s.select_active_device(2)
>>> s.get_active_device() #doctest:+ELLIPSIS
<saleae.ConnectedDevice #2 LOGIC_8_DEVICE Logic 8 (...) **ACTIVE**>
>>> s.select_active_device(1)
```

set_active_channels(*digital=None*, *analog=None*)
Set the active digital and analog channels.

Note: This feature is only supported on Logic 16, Logic 8(2nd gen), Logic Pro 8, and Logic Pro 16

Raises

- ***ImpossibleSettings*** – if used with a Logic 4 device
- ***ImpossibleSettings*** – if no active channels are given

```
>>> s.set_active_channels([0,1,2,3], [0]) #doctest:+SKIP
```

set_capture_pretrigger_buffer_size(*size*, *round=True*)
Set the number of samples saleae records before the trigger.

```
>>> s.set_capture_pretrigger_buffer_size(1e6)
```

set_capture_seconds(*seconds*)
Set the capture duration to a length of time.

Parameters **seconds** – Capture length. Partial seconds (floats) are fine.

```
>>> s.set_capture_seconds(1)
```

set_digital_voltage_option(*index*)
Set active digital I/O voltage threshold for the device.

This feature is only supported on the original Logic 16, Logic Pro 8, and Logic Pro 16.

Parameters **index** – digital I/O voltage threshold index from the getter function.

Raises ***ImpossibleSettings*** – raised if out of range index is requested

```
>>> s.set_digital_voltage_option(0) #doctest:+SKIP
```

set_num_samples(*samples*)
Set the capture duration to a specific number of samples.

Parameters **samples** – Number of samples to capture, will be coerced to `int`

From Saleae documentation Note: USB transfer chunks are about 33ms of data so the number of samples you actually get are in steps of 33ms.

```
>>> s.set_num_samples(1e6)
```

set_performance(*performance*)
Set performance value. Performance controls USB traffic and quality.

Parameters **performance** – must be of type saleae.PerformanceOption

Note: This will change the sample rate.

#>>> s.set_performance(saleae.PerformanceOption.Full)

set_sample_rate(*sample_rate_tuple*)
Set the sample rate. Note the caveats. Consider `set_sample_rate_by_minimum`.

Due to saleae software limitations, only sample rates exposed in the Logic software can be used. Use the `get_all_sample_rates` method to get all of the valid sample rates. The list of valid sample rates changes based on the number and type of active channels, so set up all channel configuration before attempting to set the sample rate.

Parameters **sample_rate_tuple** – A sample rate as returned from `get_all_sample_rates`

```
>>> s.set_sample_rate(s.get_all_sample_rates()[0])
```

set_sample_rate_by_minimum(*digital_minimum=0*, *analog_minimum=0*)
Set to a valid sample rate given current configuration and a target.

Because the avaiable sample rates are not known until runtime after all other configuration, this helper method takes a minimum digital and/or analog sampling rate and will choose the minimum sampling rate available at runtime. Setting digital or analog to 0 will disable the respective sampling method.

Parameters

- **digital_minimum** – Minimum digital sampling rate in samples/sec or 0 for don't care
- **analog_minimum** – Minimum analog sampling rate in samples/sec or 0 for don't care

Returns (digital_rate, analog_rate) the sample rate that was set

Raises *ImpossibleSettings* – rasied if sample rate cannot be met

```
>>> s.set_sample_rate_by_minimum(1e6, 1)
(12000000, 10)
```

set_trigger_one_channel(*digital_channel*, *trigger*)

Convenience method to set one trigger.

Parameters

- **channel** – Integer specifying channel
- **trigger** – saleae.Trigger indicating trigger type

Raises *ImpossibleSettings* – rasied if channel is not active

set_triggers_for_all_channels(*channels*)

Set the trigger conditions for all active digital channels.

Parameters **channels** – An array of saleae.Trigger for each channel

Raises *ImpossibleSettings* – rasied if configuration is not provided for all channels

Note: Calls to this function must always set all active digital channels. The Saleae protocol does not currently expose a method to read current triggers.

Index