
pyrtlsdr Documentation

Release 0.2.92

Roger <<https://github.com/roger-/pyrtlsdr>>

Sep 07, 2021

CONTENTS

1 Overview	1
1.1 Description	1
1.2 Links	1
1.3 Usage	1
1.4 Experimental features	4
1.5 Dependencies	5
1.6 Todo	5
1.7 Troubleshooting	5
1.8 License	6
1.9 Credit	6
2 Reference	7
2.1 <code>rtlsdr.rtlsdr</code>	7
2.2 <code>rtlsdr.rtlsdraio</code>	12
2.3 <code>rtlsdr.rtlsdrtcp</code>	14
2.4 <code>rtlsdr.helpers</code>	19
3 Indices and tables	21
Python Module Index	23
Index	25

OVERVIEW

1.1 Description

pyrtlsdr is a simple Python interface to devices supported by the RTL-SDR project, which turns certain USB DVB-T dongles employing the Realtek RTL2832U chipset into low-cost, general purpose software-defined radio receivers. It wraps many of the functions in the `librtlsdr` library including asynchronous read support and also provides a more Pythonic API.

1.2 Links

- Documentation:
 - <https://pyrtlsdr.readthedocs.io/>
- Releases:
 - <https://pypi.org/project/pyrtlsdr/>
- Source code and project home:
 - <https://github.com/roger-/pyrtlsdr>
- Releases for `librtlsdr`:
 - <https://github.com/librtlsdr/librtlsdr/releases>

1.3 Usage

pyrtlsdr can be installed by downloading the source files and running `python setup.py install`, or using `pip` and `pip install pyrtlsdr`.

All functions in `librtlsdr` are accessible via `librtlsdr.py` and a Pythonic interface is available in `rtlsdr.py` (recommended). Some documentation can be found in docstrings in the latter file.

1.3.1 Examples

Simple way to read and print some samples:

```
from rtlsdr import RtlSdr

sdr = RtlSdr()

# configure device
sdr.sample_rate = 2.048e6 # Hz
sdr.center_freq = 70e6    # Hz
sdr.freq_correction = 60  # PPM
sdr.gain = 'auto'

print(sdr.read_samples(512))
```

Plotting the PSD with matplotlib:

```
from pylab import *
from rtlsdr import *

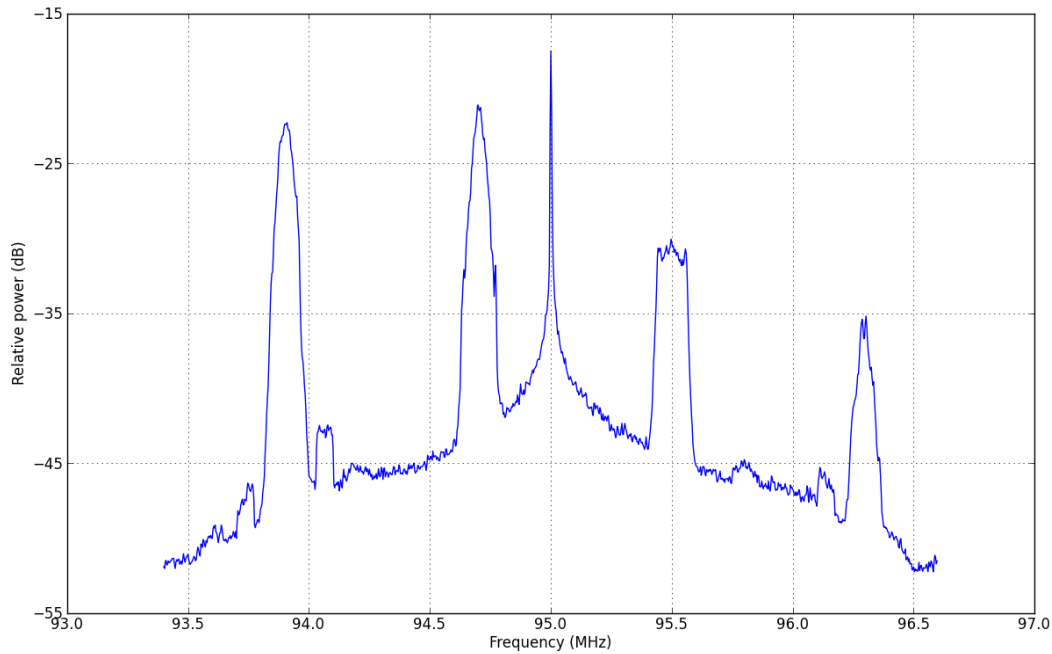
sdr = RtlSdr()

# configure device
sdr.sample_rate = 2.4e6
sdr.center_freq = 95e6
sdr.gain = 4

samples = sdr.read_samples(256*1024)
sdr.close()

# use matplotlib to estimate and plot the PSD
psd(samples, NFFT=1024, Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6)
xlabel('Frequency (MHz)')
ylabel('Relative power (dB)')

show()
```

Resulting Plot:

See the files ‘demo_waterfall.py’ and ‘test.py’ for more examples.

1.3.2 Handling multiple devices:

(added in v2.5.6)

```
from rtlsdr import RtlSdr

# Get a list of detected device serial numbers (str)
serial_numbers = RtlSdr.get_device_serial_addresses()

# Find the device index for a given serial number
device_index = RtlSdr.get_device_index_by_serial('00000001')

sdr = RtlSdr(device_index)

# Or pass the serial number directly:
sdr = RtlSdr(serial_number='00000001')
```

Note

Most devices by default have the same serial number: '0000001'. This can be set to a custom value by using the `rtl_eeprom` utility packaged with `librtlsdr`.

1.4 Experimental features

Two new submodules are available for testing: `rtlsdraio`, which adds native Python 3 asynchronous support (asyncio module), and `rtlsdrtcp` which adds a TCP server/client for accessing a device over the network. See the respective modules in the `rtlsdr` folder for more details and feel free to test and report any bugs!

1.4.1 rtlsdraio

Note that the `rtlsdraio` module is automatically imported and adds `stream()` and `stop()` methods to the normal `RtlSdr` class. It also requires the new `async/await` syntax introduced in Python 3.5+.

The syntax is basically:

```
import asyncio
from rtlsdr import RtlSdr

async def streaming():
    sdr = RtlSdr()

    async for samples in sdr.stream():
        # do something with samples
        # ...

    # to stop streaming:
    await sdr.stop()

    # done
    sdr.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(streaming())
```

1.4.2 rtsdrtcp

The `RtlSdrTcpServer` class is meant to be connected physically to an SDR dongle and communicate with an instance of `RtlSdrTcpClient`. The client is intended to function as closely as possible to the base `RtlSdr` class (as if it had a physical dongle attached to it).

Both of these classes have the same arguments as the base `RtlSdr` class with the addition of `hostname` and `port`:

```
server = RtlSdrTcpServer(hostname='192.168.1.100', port=12345)
server.run_forever()
# Will listen for clients until Ctrl-C is pressed
```



```
# On another machine (typically)
client = Rt1SdrTcpClient(hostname='192.168.1.100', port=12345)
client.center_freq = 2e6
data = client.read_samples()
```

1.4.3 TCP Client Mode

On platforms where the `librtlsdr` library cannot be installed/compiled, it is possible to import the `Rt1SdrTcpClient` only by setting the environment variable "RTLSDR_CLIENT_MODE" to "true". If this is set, no other modules will be available.

Feature added in v0.2.4

1.5 Dependencies

- Windows/Linux/OSX
- Python 2.7.x/3.3+
- `librtlsdr`
- **Optional:** NumPy (wraps samples in a more convenient form)

`matplotlib` is also useful for plotting data. The `librtlsdr` binaries (`rtlsdr.dll` in Windows and `librtlsdr.so` in Linux) should be in the `pyrtlsdr` directory, or a system path. Note that these binaries may have additional dependencies.

1.6 Todo

There are a few remaining functions in `librtlsdr` that haven't been wrapped yet. It's a simple process if there's an additional function you need to add support for, and please send a pull request if you'd like to share your changes.

1.7 Troubleshooting

- Some operating systems (Linux, OS X) seem to result in `libusb` buffer issues when performing small reads. Try reading 1024 (or higher powers of two) samples at a time if you have problems.
- If you're having `librtlsdr` import errors:
 - **Windows:** Make sure all the `librtlsdr` DLL files (`librtlsdr.dll`, `libusb-1.0.dll`) are in your system path, or the same folder as this README file. Also make sure you have all of *their* dependencies (e.g. `libgcc_s_dw2-1.dll` or possibly the Visual Studio runtime files). If `rtl_sdr.exe` works, then you should be okay. Also note that you can't mix the 64 bit version of Python with 32 bit builds of `librtlsdr`, and vice versa.
 - **Linux:** Make sure your `LD_LIBRARY_PATH` environment variable contains the directory where the `librtlsdr.so.0` library is located. You can do this in a shell with (for example): `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib`. See [this issue](#) for more details.

1.8 License

All of the code contained here is licensed by the GNU General Public License v3.

1.9 Credit

Credit to dbasden for his earlier wrapper [python-librtlsdr](#) and all the contributors on GitHub.

Copyright (C) 2013 by Roger <https://github.com/roger->

2.1 rtl_sdr.rtl_sdr

exception `rtl_sdr.rtl_sdr.LibUSBError(errno, msg=")`

Bases: `OSError`

class `rtl_sdr.rtl_sdr.BaseRtlSdr(device_index=0, test_mode_enabled=False, serial_number=None)`

Bases: `object`

Core interface for most API functionality

Parameters

- **device_index** (`int`, optional) – The device index to use if there are multiple dongles attached. If only one is being used, the default value (0) will be used.
- **test_mode_enabled** (`bool`, optional) – If True, enables a special test mode, which will return the value of an internal RTL2832 8-bit counter with calls to `read_bytes()`.
- **serial_number** (`str`, optional) – If not None, the device will be searched for by the given serial_number by `get_device_index_by_serial()` and the device_index returned will be used automatically.

DEFAULT_GAIN

Default *gain* value used on initialization: 'auto'

DEFAULT_FC

Default *center_freq* value used on initialization: 80e6 (80 Mhz)

Type `float`

DEFAULT_RS

Default *sample_rate* value used on initialization: 1.024e6 (1024 Msps)

Type `float`

DEFAULT_READ_SIZE

Default number of samples or bytes to read if no arguments are supplied for `read_bytes()` or `read_samples()`. Default value is 1024

Type `int`

gain_values

The valid gain parameters supported by the device (in tenths of dB). These are stored as returned by `librtl_sdr`.

Type `list(int)`

valid_gains_db

The valid gains in dB

Type `list(float)`

static get_device_index_by_serial(serial)

Retrieves the device index for a device matching the given serial number

Parameters `serial (str)` – The serial number to search for

Returns The device_index as reported by librtlsdr

Return type `int`

Notes

Most devices by default have the same serial number: '0000001'. This can be set to a custom value by using the `rtl_eeeprom` utility packaged with librtlsdr.

static get_device_serial_addresses()

Get serial numbers for all attached devices

Returns A list of all detected serial numbers (`str`)

Return type `list(str)`

get_gains()

Get all supported gain values from driver

Returns Gains in tenths of a dB

Return type `list(int)`

get_tuner_type()

Get the tuner type.

Returns The tuner type as reported by the driver. See the [tuner enum definition](#) for more information.

Return type `int`

init_device_values()

Retrieves information from the device

This method acquires the values for [gain_values](#). Also sets the device to the default [center frequency](#), the [sample rate](#) and [gain](#)

open(device_index=0, test_mode_enabled=False, serial_number=None)

Connect to the device through the underlying wrapper library

Initializes communication with the device and retrieves information from it with a call to [init_device_values\(\)](#).

Parameters

- **device_index** (`int`, optional) – The device index to use if there are multiple dongles attached. If only one is being used, the default value (0) will be used.
- **test_mode_enabled** (`bool`, optional) – If True, enables a special test mode, which will return the value of an internal RTL2832 8-bit counter with calls to [read_bytes\(\)](#).
- **serial_number** (`str`, optional) – If not None, the device will be searched for by the given serial_number by [get_device_index_by_serial\(\)](#) and the device_index returned will be used automatically.

Notes

The arguments used here are passed directly from object initialization.

Raises `IOError` – If communication with the device could not be established.

packed_bytes_to_iq(*bytes*)

Unpack a sequence of bytes to a sequence of normalized complex numbers

This is called automatically by `read_samples()`.

Returns The unpacked iq values as either a `list` or `numpy.ndarray` (if available).

read_bytes(*num_bytes=1024*)

Read specified number of bytes from tuner.

Does not attempt to unpack complex samples (see `read_samples()`), and data may be unsafe as buffer is reused.

Parameters `num_bytes` (`int`, optional) – The number of bytes to read. Defaults to `DEFAULT_READ_SIZE`.

Returns A buffer of `len(num_bytes)` containing the raw samples read.

Return type `ctypes.Array[c_ubyte]`

read_samples(*num_samples=1024*)

Read specified number of complex samples from tuner.

Real and imaginary parts are normalized to be in the range `[-1, 1]`. Data is safe after this call (will not get overwritten by another one).

Parameters `num_samples` (`int`, optional) – Number of samples to read. Defaults to `DEFAULT_READ_SIZE`.

Returns The samples read as either a `list` or `numpy.ndarray` (if available).

set_agc_mode(*enabled*)

Enable RTL2832 AGC

Parameters `enabled` (`bool`) –

set_direct_sampling(*direct*)

Enable direct sampling.

Parameters `direct` – If `False` or `0`, disable direct sampling. If `'i'` or `1`, use ADC I input. If `'q'` or `2`, use ADC Q input.

set_manual_gain_enabled(*enabled*)

Enable or disable manual gain control of tuner.

Parameters `enabled` (`bool`) –

Notes

If `enabled` is `False`, then AGC should also be used by calling `set_agc_mode()`. It is recommended to use `set_gain()` instead of calling this method directly.

property `bandwidth`

Get/Set bandwidth value (in Hz)

Set value to 0 (default) for automatic bandwidth selection.

Notes

This value is stored locally and may not reflect the real tuner bandwidth

Type `int`

property `center_freq`

Get/Set the center frequency of the device (in Hz)

Type `int`

property `fc`

Get/Set the center frequency of the device (in Hz)

Type `int`

property `freq_correction`

Get/Set frequency offset of the tuner (in PPM)

Type `int`

property `gain`

Get/Set gain of the tuner (in dB)

Notes

If set to 'auto', AGC mode is enabled; otherwise gain is in dB. The actual gain used is rounded to the nearest value supported by the device (see the values in `valid_gains_db`).

Type `float` or `str`

property `rs`

Get/Set the sample rate of the tuner (in Hz)

Type `int`

property `sample_rate`

Get/Set the sample rate of the tuner (in Hz)

Type `int`

class `rtlsdr.rtlsdr.Rt1Sdr`(*device_index=0, test_mode_enabled=False, serial_number=None*)

Bases: `rtlsdr.rtlsdr.BaseRt1Sdr`

This adds async read support to `BaseRt1Sdr`

`_bytes_converter_callback`(*raw_buffer, num_bytes, context*)

Converts the raw buffer used in `rtlsdr_read_async` to a usable type

This method is used internally by `read_bytes_async()` to convert the raw data from `rtlsdr_read_async` into a memory-safe array.

The callback given in `read_bytes_async()` will then be called with the signature:

```
callback(values, context)
```

Parameters

- **raw_buffer** – Buffer of type `unsigned char`
- **num_bytes** (*int*) – Length of `raw_buffer`
- **context** – User-defined value passed to `rtlsdr_read_async`. In most cases, will be a reference to the *RtISdr* instance

Notes

This method is not meant to be called directly or overridden by subclasses.

_samples_converter_callback(*buffer, context*)

Converts the raw buffer used in `rtlsdr_read_async` to a usable type

This method is used internally by `read_samples_async()` to convert the data into a sequence of complex numbers.

The callback given in `read_samples_async()` will then be called with the signature:

```
callback(samples, context)
```

Parameters

- **buffer** – Buffer of type `unsigned char`
- **context** – User-defined value passed to `rtlsdr_read_async`. In most cases, will be a reference to the *RtISdr* instance

Notes

This method is not meant to be called directly or overridden by subclasses.

cancel_read_async()

Cancel async read. This should be called eventually when using async reads (`read_bytes_async()` or `read_samples_async()`), or callbacks will never stop.

See also:

`limit_time()` and `limit_calls()`

read_bytes_async(*callback, num_bytes=1024, context=None*)

Continuously read bytes from tuner

Parameters

- **callback** – A function or method that will be called with the result. See `_bytes_converter_callback()` for the signature.
- **num_bytes** (*int*) – Number of bytes to read for each callback. Defaults to `DEFAULT_READ_SIZE`.
- **context** (*Optional*) – Object to be passed as an argument to the callback. If not supplied or `None`, the *RtISdr* instance will be used.

Notes

As with `read_bytes()`, the data passed to the callback may be overwritten.

`read_samples_async(callback, num_samples=1024, context=None)`

Continuously read 'samples' from the tuner

This is a combination of `read_samples()` and `read_bytes_async()`

Parameters

- **callback** – A function or method that will be called with the result. See `_samples_converter_callback()` for the signature.
- **num_samples** (*int*) – The number of samples read into each callback. Defaults to `DEFAULT_READ_SIZE`.
- **context** (*Optional*) – Object to be passed as an argument to the callback. If not supplied or `None`, the `RtlSdr` instance will be used.

2.2 rtlsdr.rtlsdraio

This module adds `asyncio` support for reading samples from the device.

The main functionality can be found in the `stream()` method of `rtlsdr.rtlsdraio.RtlSdrAio`.

Example

```
import asyncio
from rtlsdr import RtlSdr

async def streaming():
    sdr = RtlSdr()

    async for samples in sdr.stream():
        # do something with samples
        # ...

    # to stop streaming:
    await sdr.stop()

    # done
    sdr.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(streaming())
```

`class rtlsdr.rtlsdraio.AsyncCallbackIter(func_start, func_stop=None, queue_size=20, *, loop=None)`
Bases: `object`

Convert a callback-based legacy async function into one supporting `asyncio` and Python 3.5+

The queued data can be iterated using `async for`

Parameters

- **func_start** – A callable which should take a single callback that will be passed data. Will be run in a separate thread in case it blocks.
- **func_stop** (*optional*) – A callable to stop **func_start** from calling the callback. Will be run in a separate thread in case it blocks.
- **queue_size** (*int*, *optional*) – The maximum amount of data that will be buffered.
- **loop** (*optional*) – The `asyncio.event_loop` to use. If not supplied, `asyncio.get_event_loop()` will be used.

async add_to_queue(*args)

Add items to the queue

Parameters *args – Arguments to be added

This method is a `coroutine`

async start()

Start the execution

The callback given by **func_start** will be called by `asyncio.AbstractEventLoop.run_in_executor()` and will continue until **stop()** is called.

This method is a `coroutine`

async stop()

Stop the running executor task

If **func_stop** was supplied, it will be called after the queue has been exhausted.

This method is a `coroutine`

class `rtlsdr.rtlsdraio.Rt1SdrAio`(*device_index=0, test_mode_enabled=False, serial_number=None*)

Bases: `rtlsdr.rtlsdr.Rt1Sdr`

stop()

Stop async stream

Stops the `read_samples_async` and `Executor` task created by **stream()**.

stream(*num_samples_or_bytes=131072, format='samples', loop=None*)

Start async streaming from SDR and return an async iterator (Python 3.5+).

The `read_samples_async()` method is called in an `Executor` instance using `asyncio.AbstractEventLoop.run_in_executor()`.

The returned asynchronous iterable can then be used to retrieve sample data using `async for` syntax.

Calling the **stop()** method will stop the `read_samples_async` session and close the `Executor` task.

Parameters

- **num_samples_or_bytes** (*int*) – The number of bytes/samples that will be returned each iteration
- **format** (*str*, *optional*) – Specifies whether raw data (“bytes”) or IQ samples (“samples”) will be returned
- **loop** (*optional*) – An `asyncio` event loop

Returns An asynchronous iterator to yield sample data

2.3 rtlsdr.rtlsdrtcp

This module allows client/server communication.

The `RtlSdrTcpServer` class is meant to be connected physically to an SDR dongle and communicate with an instance of `RtlSdrTcpClient`.

The client is intended to function as closely as possible to the base `RtlSdr` class (as if it had a physical dongle attached to it).

Both of these classes have the same arguments as the base `RtlSdr` class with the addition of `hostname` and `port`.

Examples

```
server = RtlSdrTcpServer(hostname='192.168.1.100', port=12345)
server.run_forever()
# Will listen for clients until Ctrl-C is pressed
```

```
# On another machine (typically)
client = RtlSdrTcpClient(hostname='192.168.1.100', port=12345)
client.center_freq = 2e6
data = client.read_samples()
```

Note: On platforms where the `librtlsdr` library cannot be installed/compiled, it is possible to import `RtlSdrTcpClient` only by setting the environment variable "RTLSDR_CLIENT_MODE" to "true". If this is set, no other modules will be available.

Feature added in v0.2.4

2.3.1 rtlsdr.rtlsdrtcp.server

```
class rtlsdr.rtlsdrtcp.server.RequestHandler(request, client_address, server)
```

Bases: `socketserver.BaseRequestHandler`

```
close()
```

```
finish()
```

```
handle(rx_message=None)
```

```
handle_method_call(rx_message)
```

```
handle_prop_get(rx_message)
```

```
handle_prop_set(rx_message)
```

```
setup()
```

```
class rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer(device_index=0, test_mode_enabled=False,
                                              serial_number=None, hostname='127.0.0.1',
                                              port=None)
```

Bases: `rtlsdr.rtlsdr.RtlSdr`, `rtlsdr.rtlsdrtcp.base.RtlSdrTcpBase`

Server that connects to a physical dongle to allow client connections.

close()

Stops the server (if it's running) and closes the connection to the dongle.

open(*device_index=0, test_mode_enabled=False, serial_number=None*)

Connect to the device through the underlying wrapper library

Initializes communication with the device and retrieves information from it with a call to `init_device_values()`.

Parameters

- **device_index** (`int`, optional) – The device index to use if there are multiple dongles attached. If only one is being used, the default value (0) will be used.
- **test_mode_enabled** (`bool`, optional) – If True, enables a special test mode, which will return the value of an internal RTL2832 8-bit counter with calls to `read_bytes()`.
- **serial_number** (`str`, optional) – If not None, the device will be searched for by the given `serial_number` by `get_device_index_by_serial()` and the `device_index` returned will be used automatically.

Notes

The arguments used here are passed directly from object initialization.

Raises `IOError` – If communication with the device could not be established.

read_bytes(*num_bytes=1024*)

Return a packed string of bytes read along with the `struct_fmt`.

read_samples(*num_samples=1024*)

This overrides the base implementation so that the raw data is sent. It will be unpacked to I/Q samples on the client side.

run()

Runs the server thread and returns. Use this only if you are running mainline code afterwards. The server must explicitly be stopped by the `stop` method before exit.

run_forever()

Runs the server and begins a mainloop. The loop will exit with Ctrl-C.

class `rtlsdr.rtlsdrtcp.server.Server`(*rtl_sdr*)

Bases: `socketserver.TCPServer`

server_close()

Called to clean-up the server.

May be overridden.

REQUEST_RECV_SIZE = 1024

class `rtlsdr.rtlsdrtcp.server.ServerThread`(*rtl_sdr*)

Bases: `threading.Thread`

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard `run()` method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the `args` and `kwargs` arguments, respectively.

stop()

`rtlsdr.rtlsdrtcp.server.run_server()`

Convenience function to run the server from the command line with options for hostname, port and device index.

2.3.2 `rtlsdr.rtlsdrtcp.client`

class `rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient`(*device_index=0, test_mode_enabled=False, hostname='127.0.0.1', port=None*)

Bases: `rtlsdr.rtlsdrtcp.base.RtlSdrTcpBase`

Client object that connects to a remote server.

Exposes most of the methods and descriptors that are available in the `RtlSdr` class in a transparent manner allowing an interface that is nearly identical to the core API.

`close()`

`get_bandwidth()`

`get_center_freq()`

`get_freq_correction()`

`get_gain()`

`get_gains()`

`get_sample_rate()`

`get_tuner_type()`

`open(*args)`

`read_bytes(num_bytes=1024)`

`read_bytes_async(*args)`

`read_samples(num_samples=1024)`

`read_samples_async(*args)`

`set_bandwidth(value)`

`set_center_freq(value)`

`set_direct_sampling(value)`

`set_freq_correction(value)`

`set_gain(value)`

`set_sample_rate(value)`

property `bandwidth`

property `center_freq`

property `fc`

property `freq_correction`

property `gain`

property `rs`

property `sample_rate`

2.3.3 rtlsdr.rtlsdrtcp.base

exception `rtlsdr.rtlsdrtcp.base.CommunicationError(msg, source_exc=None)`

Bases: `Exception`

class `rtlsdr.rtlsdrtcp.base.AckMessage(**kwargs)`

Bases: `rtlsdr.rtlsdrtcp.base.MessageBase`

Simple message type meant for ACKnowledgemnt of message receipt

get_header(kwargs)**

Builds the header data for the message

The timestamp is added to the header if not already present.

Returns

Return type `dict`

class `rtlsdr.rtlsdrtcp.base.ClientMessage(**kwargs)`

Bases: `rtlsdr.rtlsdrtcp.base.MessageBase`

get_header(kwargs)**

Builds the header data for the message

The timestamp is added to the header if not already present.

Returns

Return type `dict`

get_response_class()

send_message(sock)

Serializes and sends the message

Parameters `sock` – The socket object to write to

class `rtlsdr.rtlsdrtcp.base.MessageBase(**kwargs)`

Bases: `object`

Base class for messages sent between clients and servers.

Handles serialization/deserialization and communication with socket type objects.

timestamp

Timestamp given from `time.time()`

Type `float`

header

A dict containing message type and payload information

Type `dict`

data

The payload containing either the request or response data

classmethod `from_remote(sock)`

Reads data from the socket and parses an instance of `MessageBase`

Parameters `sock` – The socket object to read from

get_ack_response(sock)

get_data(kwargs)**

get_header(**kwargs)

Builds the header data for the message

The *timestamp* is added to the header if not already present.

Returns

Return type dict

get_response(sock)

Waits for a specific response message

The message class returned from `get_response_class()` is used to parse the message (called from `from_remote()`)

Parameters sock – The socket object to read from

send_message(sock)

Serializes and sends the message

Parameters sock – The socket object to write to

```
class rtlsdr.rtlsdrtcp.base.Rt1SdrTcpBase(device_index=0, test_mode_enabled=False,
                                         hostname='127.0.0.1', port=None)
```

Bases: object

Base class for all rtlsdrtcp functionality

Parameters

- **device_index** (int, optional) –
- **test_mode_enabled** (bool, optional) –
- **hostname** (str, optional) –
- **port** (int, optional) –

packed_bytes_to_iq(bytes)

A direct copy of `rtlsdr.BaseRt1Sdr.packed_bytes_to_iq()`

DEFAULT_PORT = 1235

```
class rtlsdr.rtlsdrtcp.base.ServerMessage(**kwargs)
```

Bases: `rtlsdr.rtlsdrtcp.base.MessageBase`

classmethod from_remote(sock)

Reads data for the socket buffer and reconstructs the appropriate message that was sent by the other end.

This method is used by clients to reconstruct `ServerMessage` objects and if necessary, use multiple read calls to get the entire message (if the message size is greater than the buffer length)

get_data(**kwargs)

get_header(**kwargs)

Builds the header data for the message

The *timestamp* is added to the header if not already present.

Returns

Return type dict

get_response_class()

send_message(*sock*)

Sends the message data to clients.

If necessary, uses multiple calls to send to ensure all data has actually been sent through the socket objects's buffer.

2.4 rtlsdr.helpers

rtlsdr.helpers.limit_calls(*max_calls*)

Decorator to cancel async reads after the given number of calls.

Parameters **max_calls** (*int*) – Number of calls to wait for before cancelling

Examples

Stop reading after 10 calls:

```
>>> @limit_calls(10)
>>> def read_callback(data, context):
>>>     print('signal mean:', sum(data)/len(data))
>>> sdr = RtlSdr()
>>> sdr.read_samples_async(read_callback)
```

Notes

See notes in [limit_time\(\)](#)

rtlsdr.helpers.limit_time(*max_seconds*)

Decorator to cancel async reads after a specified time period.

Parameters **max_seconds** – Number of seconds to wait before cancelling

Examples

Stop reading after 10 seconds:

```
>>> @limit_time(10)
>>> def read_callback(data, context):
>>>     print('signal mean:', sum(data)/len(data))
>>> sdr = RtlSdr()
>>> sdr.read_samples_async(read_callback)
```

Notes

The context in either `read_bytes_async()` or `read_samples_async()` is relied upon and must use the default value (the `Rt1Sdr` instance)

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

rtlsdr.helpers, 19
rtlsdr.rtlsdr, 7
rtlsdr.rtlsdraio, 12
rtlsdr.rtlsdrtcp, 14
rtlsdr.rtlsdrtcp.base, 17
rtlsdr.rtlsdrtcp.client, 16
rtlsdr.rtlsdrtcp.server, 14

Symbols

`_bytes_converter_callback()` (*rtlsdr.rtlsdr.RtlSdr* method), 10
`_samples_converter_callback()` (*rtlsdr.rtlsdr.RtlSdr* method), 11

A

`AckMessage` (class in *rtlsdr.rtlsdrtcp.base*), 17
`add_to_queue()` (*rtlsdr.rtlsdraio.AsyncCallbackIter* method), 13
`AsyncCallbackIter` (class in *rtlsdr.rtlsdraio*), 12

B

`bandwidth` (*rtlsdr.rtlsdr.BaseRtlSdr* property), 10
`bandwidth` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* property), 16
`BaseRtlSdr` (class in *rtlsdr.rtlsdr*), 7

C

`cancel_read_async()` (*rtlsdr.rtlsdr.RtlSdr* method), 11
`center_freq` (*rtlsdr.rtlsdr.BaseRtlSdr* property), 10
`center_freq` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* property), 16
`ClientMessage` (class in *rtlsdr.rtlsdrtcp.base*), 17
`close()` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* method), 16
`close()` (*rtlsdr.rtlsdrtcp.server.RequestHandler* method), 14
`close()` (*rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer* method), 14
`CommunicationError`, 17

D

`data` (*rtlsdr.rtlsdrtcp.base.MessageBase* attribute), 17
`DEFAULT_FC` (*rtlsdr.rtlsdr.BaseRtlSdr* attribute), 7
`DEFAULT_GAIN` (*rtlsdr.rtlsdr.BaseRtlSdr* attribute), 7
`DEFAULT_PORT` (*rtlsdr.rtlsdrtcp.base.RtlSdrTcpBase* attribute), 18
`DEFAULT_READ_SIZE` (*rtlsdr.rtlsdr.BaseRtlSdr* attribute), 7
`DEFAULT_RS` (*rtlsdr.rtlsdr.BaseRtlSdr* attribute), 7

F

`fc` (*rtlsdr.rtlsdr.BaseRtlSdr* property), 10
`fc` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* property), 16
`finish()` (*rtlsdr.rtlsdrtcp.server.RequestHandler* method), 14
`freq_correction` (*rtlsdr.rtlsdr.BaseRtlSdr* property), 10
`freq_correction` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* property), 16
`from_remote()` (*rtlsdr.rtlsdrtcp.base.MessageBase* class method), 17
`from_remote()` (*rtlsdr.rtlsdrtcp.base.ServerMessage* class method), 18

G

`gain` (*rtlsdr.rtlsdr.BaseRtlSdr* property), 10
`gain` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* property), 16
`gain_values` (*rtlsdr.rtlsdr.BaseRtlSdr* attribute), 7
`get_ack_response()` (*rtlsdr.rtlsdrtcp.base.MessageBase* method), 17
`get_bandwidth()` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* method), 16
`get_center_freq()` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* method), 16
`get_data()` (*rtlsdr.rtlsdrtcp.base.MessageBase* method), 17
`get_data()` (*rtlsdr.rtlsdrtcp.base.ServerMessage* method), 18
`get_device_index_by_serial()` (*rtlsdr.rtlsdr.BaseRtlSdr* static method), 8
`get_device_serial_addresses()` (*rtlsdr.rtlsdr.BaseRtlSdr* static method), 8
`get_freq_correction()` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* method), 16
`get_gain()` (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient* method), 16
`get_gains()` (*rtlsdr.rtlsdr.BaseRtlSdr* method), 8

- get_gains() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 get_header() (*rtlsdr.rtlsdrtcp.base.AckMessage method*), 17
 get_header() (*rtlsdr.rtlsdrtcp.base.ClientMessage method*), 17
 get_header() (*rtlsdr.rtlsdrtcp.base.MessageBase method*), 17
 get_header() (*rtlsdr.rtlsdrtcp.base.ServerMessage method*), 18
 get_response() (*rtlsdr.rtlsdrtcp.base.MessageBase method*), 18
 get_response_class() (*rtlsdr.rtlsdrtcp.base.ClientMessage method*), 17
 get_response_class() (*rtlsdr.rtlsdrtcp.base.ServerMessage method*), 18
 get_sample_rate() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 get_tuner_type() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 8
 get_tuner_type() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
- ## H
- handle() (*rtlsdr.rtlsdrtcp.server.RequestHandler method*), 14
 handle_method_call() (*rtlsdr.rtlsdrtcp.server.RequestHandler method*), 14
 handle_prop_get() (*rtlsdr.rtlsdrtcp.server.RequestHandler method*), 14
 handle_prop_set() (*rtlsdr.rtlsdrtcp.server.RequestHandler method*), 14
 header (*rtlsdr.rtlsdrtcp.base.MessageBase attribute*), 17
- ## I
- init_device_values() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 8
- ## L
- LibUSBError, 7
 limit_calls() (*in module rtlsdr.helpers*), 19
 limit_time() (*in module rtlsdr.helpers*), 19
- ## M
- MessageBase (*class in rtlsdr.rtlsdrtcp.base*), 17
 module
 rtlsdr.helpers, 19
 rtlsdr.rtlsdr, 7
 rtlsdr.rtlsdraio, 12
 rtlsdr.rtlsdrtcp, 14
 rtlsdr.rtlsdrtcp.base, 17
- rtlsdr.rtlsdrtcp.client, 16
 rtlsdr.rtlsdrtcp.server, 14
- ## O
- open() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 8
 open() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 open() (*rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer method*), 15
- ## P
- packed_bytes_to_iq() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 9
 packed_bytes_to_iq() (*rtlsdr.rtlsdrtcp.base.RtlSdrTcpBase method*), 18
- ## R
- read_bytes() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 9
 read_bytes() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 read_bytes() (*rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer method*), 15
 read_bytes_async() (*rtlsdr.rtlsdr.RtlSdr method*), 11
 read_bytes_async() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 read_samples() (*rtlsdr.rtlsdr.BaseRtlSdr method*), 9
 read_samples() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 read_samples() (*rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer method*), 15
 read_samples_async() (*rtlsdr.rtlsdr.RtlSdr method*), 12
 read_samples_async() (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient method*), 16
 REQUEST_RECV_SIZE (*rtlsdr.rtlsdrtcp.server.Server attribute*), 15
 RequestHandler (*class in rtlsdr.rtlsdrtcp.server*), 14
 rs (*rtlsdr.rtlsdr.BaseRtlSdr property*), 10
 rs (*rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient property*), 16
 RtlSdr (*class in rtlsdr.rtlsdr*), 10
 rtlsdr.helpers
 module, 19
 rtlsdr.rtlsdr
 module, 7
 rtlsdr.rtlsdraio
 module, 12
 rtlsdr.rtlsdrtcp
 module, 14
 rtlsdr.rtlsdrtcp.base
 module, 17
 rtlsdr.rtlsdrtcp.client
 module, 16

`rtlsdr.rtlsdrtcp.server`
module, 14

`RtlSdrAio` (class in `rtlsdr.rtlsdraio`), 13

`RtlSdrTcpBase` (class in `rtlsdr.rtlsdrtcp.base`), 18

`RtlSdrTcpClient` (class in `rtlsdr.rtlsdrtcp.client`), 16

`RtlSdrTcpServer` (class in `rtlsdr.rtlsdrtcp.server`), 14

`run()` (`rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer` method), 15

`run()` (`rtlsdr.rtlsdrtcp.server.ServerThread` method), 15

`run_forever()` (`rtlsdr.rtlsdrtcp.server.RtlSdrTcpServer` method), 15

`run_server()` (in module `rtlsdr.rtlsdrtcp.server`), 15

S

`sample_rate` (`rtlsdr.rtlsdr.BaseRtlSdr` property), 10

`sample_rate` (`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` property), 16

`send_message()` (`rtlsdr.rtlsdrtcp.base.ClientMessage` method), 17

`send_message()` (`rtlsdr.rtlsdrtcp.base.MessageBase` method), 18

`send_message()` (`rtlsdr.rtlsdrtcp.base.ServerMessage` method), 18

`Server` (class in `rtlsdr.rtlsdrtcp.server`), 15

`server_close()` (`rtlsdr.rtlsdrtcp.server.Server` method), 15

`ServerMessage` (class in `rtlsdr.rtlsdrtcp.base`), 18

`ServerThread` (class in `rtlsdr.rtlsdrtcp.server`), 15

`set_agc_mode()` (`rtlsdr.rtlsdr.BaseRtlSdr` method), 9

`set_bandwidth()` (`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`set_center_freq()` (`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`set_direct_sampling()` (`rtlsdr.rtlsdr.BaseRtlSdr` method), 9

`set_direct_sampling()`
(`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`set_freq_correction()`
(`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`set_gain()` (`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`set_manual_gain_enabled()` (`rtlsdr.rtlsdr.BaseRtlSdr` method), 9

`set_sample_rate()` (`rtlsdr.rtlsdrtcp.client.RtlSdrTcpClient` method), 16

`setup()` (`rtlsdr.rtlsdrtcp.server.RequestHandler` method), 14

`start()` (`rtlsdr.rtlsdraio.AsyncCallbackIter` method), 13

`stop()` (`rtlsdr.rtlsdraio.AsyncCallbackIter` method), 13

`stop()` (`rtlsdr.rtlsdraio.RtlSdrAio` method), 13

`stop()` (`rtlsdr.rtlsdrtcp.server.ServerThread` method), 15

`stream()` (`rtlsdr.rtlsdraio.RtlSdrAio` method), 13

T

`timestamp` (`rtlsdr.rtlsdrtcp.base.MessageBase` attribute), 17

V

`valid_gains_db` (`rtlsdr.rtlsdr.BaseRtlSdr` attribute), 7