PyRF3 API Reference Guide

Doc Version 1.0.3

November, 2020

CONTENTS

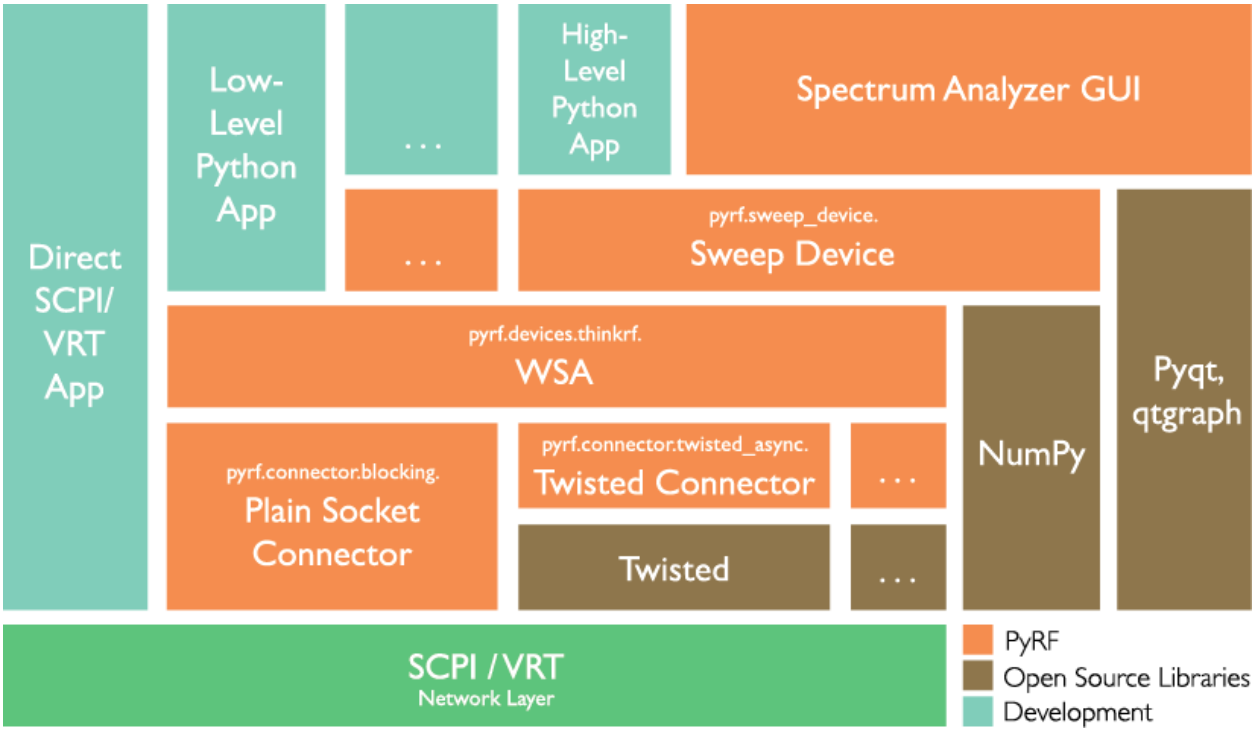
CHAPTER

ONE

API OVERVIEW

PyRF3, developed by thinkRF, is an openly available, comprehensive development environment for wireless signal analysis. It handles the low-level details of configuring an RF device, real-time data acquisition and signal processing, allowing you to concentrate on your analysis solutions. Hence, it enables rapid development of powerful applications that leverage the new generation of measurement-grade software-defined radio technology, such as thinkRF Real-Time Spectrum Analysis Software that works with thinkRF Real-Time Spectrum Analyzers (RTSA).

PyRF3 is built on the Python 3 Programming Language and includes feature-rich libraries, examples with visualization and source code, all specific to the requirements of signal analysis. It is openly available, allowing commercialization of solutions through BSD open licensing and offering device independence via standard hardware APIs.



1.1 Hardware Support

This library version currently supports the following thinkRF RTSA platforms:

- R5500 / R5550-408, 408P, 418, 427, WBIQ option
- R5700 / R5750-408, 418, 427, WBIQ option

CHAPTER

TWO

PYRF3 API

2.1 Installation Instructions

This section provides information on how to install the required python packages.

Note: Python v3.6.7 or higher is the required version for PyRF3.

2.1.1 Windows Setup

1. Set-up Python3

- Install python3.6.7 or higher if not already available in your system.
- During the Python3 installation, paths to Python3 and its Scripts are added automatically to Windows system's PATH environment. Double check to make sure these are indeed added (see under **Advanced System Properties > Enviroment Variables...**).

2. Install Dependencies

These installation steps make use of pip3 software to install required libraries. **pip3** is included with python 3.6.7 or higher installation (see above).

To verify, open a command prompt window and type **pip3**, if a help menu appears, **pip3** is ready in your system.

- Now use **pip3** to install the dependencies by typing into the command prompt window:

```
pip3 install numpy scipy netifaces qt-reactor flatdict pyside2 pywin32
```

- Optional:

```
pip3 install matplotlib Twisted
```

matplotlib and/or Twisted are only needed for some examples or if you intend to develop using these libraries.

Continue with *PyRF3 Installation* below.

2.1.2 Linux Setup

These instructions are tested on Debian/Ubuntu system, equivalent **apt** command might be needed for your Linux system. *sudo*/super user privilege might be needed in some of the steps.

1. Set-up Python3

- Update your system:

```
apt update
```

- Install python3.6.7 or higher if not already available in your system. If installing from source, additional packages might be needed to build the source. Check with python site or google search.

2. Install Dependencies

- Install **pip/pip3** if not already available (verify by typing **pip** or **pip3** & press enter in a command terminal):

```
apt install python3-pip
```

Might want to check the pip/pip3 version as newly installed pip/pip3 might not be the default "pip" command:

```
pip3 --version
```

- Install required libraries (sudo or super-user privilege might be needed):

```
pip3 install numpy scipy pyside2 netifaces qt-reactor flatdict
```

- Optional:

```
pip3 install matplotlib Twisted
```

matplotlib and/or Twisted are only needed for some examples or if you intend to develop using these libraries.

Continue with *PyRF3 Installation* below.

2.1.3 PyRF3 Installation

- Download the latest PyRF3 API and extract it.
- Open a command terminal and navigate to the extracted PyRF3 API directory.
- thinkRF provides a special modified **pyqtgraph** version that works with PySide2. This version is needed for the API and is included in the API folder. In the terminal, run:

```
easy_install-<version> pyqtgraph-0.11.0.dev0.egg
```

or:

```
pip3 install pyqtgraph-0.11.0.dev0.whl
```

(replace 'x' with the appropriate version number)

The <version> for **easy-install** refers to which python version that this tool associated with (ex. if you have used python v3.8.x, then it would be easy_install-3.8). Hint: Use the tab keyboard button. For Windows users, easy_install-<version>'.exe' might be needed.

- Now install the PyRF API by running:

```
easy_install-<version> pyrf-3.x.x.egg
```

or:

```
pip3 install pyrf-3.x.x.whl
```

(replace 'x' with the appropriate version number)

For Linux, sudo previlegde might be needed.

If the installation is successful, you should be able to run the examples in the **examples** folder.

2.2 API Usage

pyrf.devices.thinkrf.WSA is the class that provides interface and controls to thinkRF Real-Time Spectrum Analyzer (RTSA, also formerly known as WSA) devices. The methods of this class closely match the SCPI command set described in a RTSA's Programmer's Guide (available on thinkRF Resources).

This API may be used in a **blocking** mode (the default) or in an **asynchronous** mode with using the Twisted python library.

In **blocking** mode, all methods that read from the device will wait to receive a response before returning.

In **asynchronous** mode, all methods will send their commands to the device and then immediately return a Twisted Deferred object. If you need to wait for the response or completion of this command, you can attach a callback to the Deferred object and the Twisted reactor will call it when ready. You may choose to use Twisted's `inlineCallbacks` function decorator to write Twisted code that resembles synchronous code by yielding the Deferred objects returned from the API.

To use the **asynchronous**, when a WSA instance of a device (ex. `dut = WSA()`) is created, you must pass a *pyrf.connectors.twisted_async.TwistedConnector* instance as the connector parameter, as shown in *discovery.py / twisted_discovery.py*.

There are simple examples illustrating usage of this API under the *examples* directory included with the source code directory. Some are mentioned in the *Examples* section of this document.

2.2.1 Limitations:

- Sweep-device does not handle decimation.
- Frequency shift is not handled in DD mode.

2.3 API Reference

2.3.1 pyrf.devices

.thinkrf

class pyrf.devices.thinkrf.**WSA**(*connector=None*)

Interface for thinkRF's R55x0 and R57x0 & their variants.

Parameters **connector** – Connector object to use for SCPI/VRT connections, defaults to a new *PlainSocketConnector* instance

connect() must be called before other methods are used.

Note: The following methods will either block then return a result or if you passed a *TwistedConnector* instance to the constructor, they will immediately return a Twisted Deferred object.

The methods are grouped and listed by functionalities.

Connection Related Methods:

connect(*host*)
Connect to an RTSA (aka WSA).
Parameters **host** (*str*) – the hostname or IP to connect to
Usage:

```
dut.connect('123.456.789.1')
```

disconnect()
Close a connection to an RTSA (aka WSA).

async_connector()
Return True if the connector being used is asynchronous

set_async_callback(*callback*)
Set the asynchronous callback for a function when the device receives a VRT packet. Use with Twisted setup. :param callback: callback to set. Set to *None* to disable receiving packets.

Direct SCPI commands:

scpiget(*cmd*, *timeout*)
Send a SCPI *query* command and wait for the response.

This is the lowest-level interface provided. See the product's Programmer's Guide for the SCPI commands available.
Parameters
- **cmd** (*str*) – the SCPI command to send
- **timeout** (*int*) – the wait time (in seconds) for response

Returns the response output from the box if any

scpiset(*cmd*)
Send a SCPI command of set type (i.e. not query command).

This is the lowest-level interface provided. See the product's Programmer's Guide for the SCPI commands available.
Parameters **cmd** (*str*) – the command to send

errors()
Flush and return the list of errors from past commands sent to the RTSA. An empty list is returned when no errors are present.

Device System Related:

id()
Returns the RTSA's identification information string.
Returns "<Manufacturer>,<Model>,<Serial number>,<Firmware version>"

reset()
Resets the RTSA to its default configuration. It does not affect the registers or queues associated with the IEEE mandated commands.

locked(*modulestr*)
This command queries the lock status of the RF VCO (Voltage Control Oscillator) in the Radio Front End (RFE) or the lock status of the PLL reference clock in the digitizer card.
Parameters **modulestr** (*str*) – 'VCO' for rf lock status, 'CLKREF' for ref clock lock status
Returns True if locked

Data Acquisition Related Methods:

- Get permission:

has_data()
Check if there is VRT data to read.
Returns True if there is a packet to read, False if not

request_read_perm()
Acquire exclusive permission to read data from the RTSA.
Returns True if allowed to read, False if not

have_read_perm()
Check if we have permission to read data from the RTSA.
Returns True if allowed to read, False if not

- Set capture size for stream or block mode capture:

ppb(*packets=None*)
This command sets the number of IQ packets in a capture block
Parameters **packets** (*int*) – the number of packets for a block of capture, or *None* to query
Returns the current ppb value if the packets parameter is *None*

spp(*samples=None*)
This command sets or queries the number of Samples Per [VRT] Packet (SPP).

The upper bound of the *samples* is limited by the VRT's 16-bit packet size field. However, since the SPP must be a multiple of 32, the maximum is thus limited by (2**16 - 32) or 65504.
Parameters **samples** (*int*) – the number of samples in a VRT packet (256 to 65504, a multiple of 32), or *None* to query
Returns the current spp value if the samples parameter is *None*

- Stream setup

stream_start(*stream_id=None*)
This command begins the execution of the stream capture. It will also initiate data capturing. Data packets will be streamed (or pushed) from the RTSA whenever data is available.
Parameters **stream_id** (*int*) – optional unsigned 32-bit stream identifier

stream_stop()
This command stops the stream capture. After receiving the command, the RTSA system will stop when the current capturing VRT packet is completed. Recommend calling *flush()* after stopping.

- Sweep setup:

sweep_add(*entry*)
Add a sweep entry to the sweep list
Parameters **entry** (*pyrf.sweepDevice.sweepSettings*) – the sweep entry settings to add to the list

sweep_clear()
Remove all entries from the sweep list.

sweep_iterations(*count=None*)
Set or query the number of iterations to loop through a sweep list.
Parameters **count** (*int*) – the number of iterations, 0 for infinite, or *None* to query

Returns the current number of iterations if count is *None*

sweep_read(*index*)
Read a sweep entry at the given sweep *index* from the sweep list.
Parameters **index** – the index of the entry to read
Returns settings of that sweep entry
Return type *pyrf.config.SweepEntry*

sweep_start(*start_id=None*)
Start the sweep engine with an optional ID.
Parameters **start_id** (*int*) – An optional 32-bit ID to identify the sweep

sweep_stop()
Stop the sweep engine. Recommend calling *flush()* after stopping.

- VRT data acquisition related methods:

capture(*spp*, *ppb*)
This command will **start** the single block capture of *ppb* packets of *spp* samples in each packet. The data within a single block capture trace is continuous from one packet to the other, but not necessary between successive block capture commands issued. Used for **stream** or **block** capture mode. To read data back, use *read()* method. See example/show_i_q.py as an example.
Parameters
- **spp** (*int*) – the number of samples in a VRT packet
- **ppb** (*int*) – the number of packets in a block of capture

capture_mode()
This command queries the current capture mode
Returns the current capture mode

raw_read(*num*)
Raw read of VRT socket data of *num* bytes from the RTSA.
Parameters **num** (*int*) – the number of bytes to read
Returns bytes

read()
Read and return a single **parsed** VRT packet from the RTSA, either context or data.

read_data(*spp*)
Read and return a data packet, as well as computed power spectral density data, of *spp* (samples per packet) size, the associated context info and the computed power spetral data. If a block of data is requested (such as *ppb* is more than 1), loop through this function to retreive all data. See also data other capture functions: *capture_spectrum*, capture_time_domain
Parameters **spp** (*int*) – the number of samples in a VRT packet (256 to 65504) in a multiple of 32
Returns data, context dictionary, and power spectral data array

abort()
This command will cause the RTSA to stop the data capturing, whether in the manual trace block capture, triggering or sweeping mode. The RTSA will be put into the manual mode; in other words, process such as streaming, trigger and sweep will be stopped. The capturing process does not wait until the end of a packet to stop, it will stop immediately upon receiving the command.

flush()
This command clears the RTSA's internal data storage buffer of any data that is waiting to be sent. Thus, it is recommended that the flush command should be used when

switching between different capture modes to clear up the remnants of captured packets.

eof()
Check if the VRT stream has closed.
Returns True if no more data, False if more data

Device Configuration Methods for Non-Sweep Setup:

attenuator(*atten_val=None*)
This command enables, disables or queries the RTSA's RFE attenuation.

Parameters **atten_val** – see Programmer's Guide for the attenuation value to use for your product; *None* to query

Returns the current attenuation value if *None* is used

decimation(*value=None*)
This command sets or queries the rate of decimation of samples in a trace capture. The supported rate is 4 - 1024. When the rate is set to 1, no decimation is performed on the trace capture.

Parameters **value** (*int*) – new decimation value (1 or 4 - 1024); *None* to query

Returns the decimation value if *None* is used

freq(*freq=None*)
This command sets or queries the tuned center frequency of the RTSA.

Parameters **freq** (*int*) – the center frequency in Hz (range vary depnding on the product model); *None* to query

Returns the frequency in Hz if *None* is used

fshift(*shift=None*)
This command sets or queries the frequency shift value.

Parameters **freq** (*int*) – the frequency shift in Hz (0 - 125 MHz); *None* to query

Returns the amount of frequency shift if *None* is used

hdr_gain(*gain=None*)
This command sets or queries the HDR gain of the receiver. The gain has a range of -10 to 30 dB.

Parameters **gain** (*int*) – float between -10 and 30 to set; *None* to query

Returns the hdr gain in dB if *None* is used

iq_output_path(*path=None*)
This command sets or queries the RTSA's current IQ path. It is not applicable to R5700.

Parameters **path** (*str*) – 'DIGITIZER', 'CONNECTOR', 'HIF', or *None* to query

Returns the current IQ output path type if *None* is used

pll_reference(*src=None*)
This command sets or queries the RTSA's PLL reference source

Parameters **src** (*str*) – 'INT', 'EXT', 'GNSS' (when available with the model) or *None* to query

Returns the current PLL reference source if *None* is used

psfm_gain(*gain=None*)
This command sets or queries one of the Pre-Select Filter Modules's (PSFM) gain stages.

Parameters **gain** (*str*) – sets the gain value to 'high', 'medium', 'low', or *None* to query

Returns the RF gain value if *None* is used

Usage:

```
dut.psfm_gain('HIGH')
```

rfe_mode(*mode=None*)
This command sets or queries the RTSA's Receiver Front End (RFE) mode of operation.

Parameters **mode** (*str*) – 'ZIF', 'DD', 'HDR', 'SHN', 'SH', or *None* to query

Returns the current RFE mode if *None* is used

trigger(*settings=None*)
This command sets or queries the type of trigger event. Setting the trigger type to "NONE" is equivalent to disabling the trigger execution; setting to any other type will enable the trigger engine.

Parameters **settings** (*dict*) – the new trigger settings; *None* to query

Returns the trigger settings if *None* is used

apply_device_settings(*settings*, *force_change=False*)
This command takes a dict of device settings, and applies them to the RTSA

Parameters

- **settings** – dict containing device's settings such as attenuation, decimation, etc
- **force_change** (*bool*) – all settings must be applied

DSP and Data Processing Related Methods:

measure_noisefloor(*rbw=None*, *average=1*)
Returns a power level that represents the top edge of the noisefloor

Parameters

- **rbw** (*int*) – rbw of spectral capture (Hz) (will round to nearest native RBW) or None
- **average** (*int*) – number of capture iterations

Returns noise_power

peakfind(*n=1*, *rbw=None*, *average=1*)
Call capture_spectrum() to do a block capture & returns frequency and the power level of the peak (maximum) spectral point(s) computed using the current settings.

Parameters

- **n** (*int*) – determine the number of peaks to return
- **rbw** (*int*) – rbw of spectral capture (Hz) (will round to nearest native RBW) or None
- **average** (*int*) – number of capture iterations
- **dec** (*int*) – decimation factor applied

- **fshift** (*int*) – the fshift applied, in Hz

Returns [(peak_freq1, peak_power1), (peak_freq2, peak_power2) , ..., (peak_freqn, peak_powern)]

Data Recording Related Methods:

inject_recording_state(*state*)
Inject the current RTSA state into the recording stream when the next capture is received. Replaces previous data if not yet sent.

set_recording_output(*output_file=None*)
Dump a recording of all the received packets to output_file

Methods for Performing Spectral Flattening:

These methods only apply to devices that has been calibrated with flattening data during the factory installation.

configure_flattening(*loaded=None*)
Load calibration vectors using `calibration_vectors`. If present, call callback when loaded.

Parameters **loaded** – A callback to called when flattening vectors are loaded or None. When failed, an err object.

flattening_enabled(*enable=None*)
When called with enable=True, subsequent captures and sweeps will be flattened. When called with enable=False, subsequent captures and sweeps will NOT be flattened. When called with enable=None, current enable state is returned.

Parameters **enable** (*bool or None*) – A boolean to enable flattening or not. If 'None', default to True.

Returns The boolean state of the flattening or error.

has_flattening()
Determines the boolean state of the device's flattening option. :return: True if the device has calibration vectors and can flatten, else False.

flatten(*freq*, *pow_data*, *rbw*, *attenuation*, *invert=False*)
Applies the device's calibration vectors if available. Returns the flattened curve, or the original curve if the device has no vectors or something else goes wrong.

Parameters

- **freq** (*int*) – the center frequency in Hz (range vary depnding on the product model)
- **pow_data** (*list*) – spectral data (in dBm) to be analyzed
- **rbw** (*int*) – rbw of the spectral capture (Hz) (will round to nearest native RBW) or None
- **attenuation** (*int*) – the amount of attenuation to apply. Varies depend on products.
- **invert** (*bool*) – The spectral inversion status, as indicated in the VRT IF data's trailer word. Default to False.

Returns The flattened spectral data.

Device Discovery Functions:

pyrf.devices.thinkrf.**discover_wsa**(*wait_time=0.125*)

pyrf.devices.thinkrf.**parse_discovery_response**(*response*)
This function parses the RTSA's raw discovery response

Parameters **response** – The RTSA's raw response to a discovery query

Returns Return (model, serial, firmware version) based on a discovery response message

2.3.2 pyrf.connectors

.blocking

This module contains a class and functions for network socket block type handling, which includes sending/receiving packets, in relating to RTSA's SCPI commands and VRT data.

class pyrf.connectors.blocking.**PlainSocketConnector**
This connector makes SCPI/VRT socket connections using plain sockets, of blocking type. Timeout is defaulted to 8secs

connect(*host*, *timeout=8*)
Connection to both SCPI & VRT ports with a timeout. If after the *timeout* seconds, nothing has happened, throw an exception.

disconnect()
Attempt to disconnect safely from both SCPI and VRT ports

pyrf.connectors.blocking.**socketread**(*socket*, *count*, *flags=None*)
Retry socket read until *count* amount of data received, like reading from a file.

Parameters

- **count** (*int*) – the amount of data received
- **flags** – socket.recv() related flags

.twisted_async

This module contains Twisted classes and functions for asynchronous network socket handling, which includes sending/receiving packets, in relating to RTSA's SCPI commands and VRT data.

class pyrf.connectors.twisted_async.**SCPIClient**(*timeout*)

connectionMade()
Called when a connection is made.

This may be considered the initializer of the protocol, because it is called when the connection is completed. For clients, this is called once the connection to the server has been established; for servers, this is called after an accept() call stops blocking and a socket has been received. If you need to send any greeting or initial message, do it here.

dataReceived(*data*)
Called whenever data is received.

Use this method to translate to a higher-level message. Usually, some callback will be made upon the receipt of each complete protocol message.

@param data: a string of indeterminate length. Please keep in mind that you will probably need to buffer some data, as partial (or multiple) protocol messages may be received! I recommend that unit tests for protocols call through to this method with differing chunk sizes, down to one byte at a time.

timeoutConnection()
Called when the connection times out.

Override to define behavior other than dropping the connection.

class pyrf.connectors.twisted_async.**SCPIClientFactory**(*timeout*)

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

class pyrf.connectors.twisted_async.**TwistedConnector**(*reactor*, *vrt_callback=None*)
A connector that makes SCPI/VRT connections asynchronously using Twisted method.

Parameters

- **reactor** – a twisted reactor, (ex: "from twisted.internet import reactor")
- **vrt_callback** (*callback*) – A callback may be assigned to *vrt_callback* that will be called with VRT packets as they arrive. When *vrt_callback* is None (the default), arriving packets will be ignored.

exception pyrf.connectors.twisted_async.**TwistedConnectorError**

class pyrf.connectors.twisted_async.**VRTClient**(*receive_callback*)
A Twisted protocol for the VRT connection.

Parameters **receive_callback** – a function that will be passed a vrt DataPacket or ContextPacket when it is received

connectionLost(*reason*)
Called when the connection is shut down.

Clear any circular references here, and any external references to this Protocol. The connection has been closed.

@type reason: L{twisted.python.failure.Failure}

dataReceived(*data*)
Called whenever data is received.

Use this method to translate to a higher-level message. Usually, some callback will be made upon the receipt of each complete protocol message.

@param data: a string of indeterminate length. Please keep in mind that you will probably need to buffer some data, as partial (or multiple) protocol messages may be received! I recommend that unit tests for protocols call through to this method with differing chunk sizes, down to one byte at a time.

makeConnection(*transport*)
Make a connection to a transport and a server.

This sets the 'transport' attribute of this Protocol, and calls the connectionMade() callback.

class pyrf.connectors.twisted_async.**VRTClientFactory**(*receive_callback*)

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

2.3.3 pyrf.capture_device

This module provides classes which have data capture functions, including quick device configuration.

class pyrf.capture_device.**CaptureDevice**(*real_device*, *async_callback=None*, *device_settings=None*)
Virtual device that returns power levels generated from a single data packet

Parameters

- **real_device** – the device that will be used for capturing data, typically a `pyrf.thinkrf.WSA` instance.
- **async_callback** – callback to use for async operation (not used if *real_device* is using a blocking `PlainSocketConnector`)
- **device_settings** – initial device settings to use, passed to *configure_device()* if given

capture_time_domain(*rfe_mode*, *freq*, *rbw*, *device_settings=None*, *min_points=256*, *force_change=False*)
Initiate a block capture of raw time domain IQ or I-only data

Parameters

- **rfe_mode** (*str*) – radio front end mode, e.g. 'ZIF', 'SH', ...
- **freq** (*int*) – center frequency in Hz to set
- **rbw** (*float*) – the resolution bandwidth (RBW) in Hz of the data to be captured (output RBW may be smaller than requested)
- **device_settings** (*dict or None*) – rfe_mode, freq, decimation, fshift and other device settings
- **min_points** (*int*) – smallest number of data points per capture from the device
- **force_change** (*bool*) – force the configuration to apply device_settings changes or not

Returns (fstart, fstop, data) where fstart & fstop are frequencies in Hz & data is a list

configure_device(*device_settings*, *force_change=False*)
Configure the device settings on the next capture

Parameters

- **device_settings** (*dict*) – rfe mode, attenuation, decimation and other device settings
- **force_change** (*bool*) – force the configuration to apply device_settings changes or not

exception pyrf.capture_device.**CaptureDeviceError**

2.3.4 pyrf.sweep_device

This module contains Sweep-device classes which provide functions for performing sweep capture with the thinkRF's RTSAs.

class pyrf.sweep_device.**SweepDevice**(*real_device*, *async_callback=None*)
Virtual device that generates power spectrum from a given frequency range by sweeping the frequencies with a real device and piecing together the FFT results.

Parameters

- **real_device** – the RF device that will be used for capturing data, typically a *pyrf.devices.thinkrf.WSA* instance.
- **async_callback** – a callback to use for async operation (not used if *real_device* is using a blocking `PlainSocketConnector`)

capture_power_spectrum(*fstart*, *fstop*, *rbw*, *device_settings=None*, *mode='SH'*, *continuous=False*)
Initiate a data capture from the *real_device* by setting up a sweep list and starting a single sweep, and then return power spectral density data along with the **actual** sweep start and stop frequencies set (which might not be exactly the same as the requested *fstart* and *fstop*).

Note: This function does not pipeline, and if the last sweep isn't received before starting a new one, it will generate a failure.

Parameters

- **fstart** (*int*) – sweep starting frequency in Hz
- **fstop** (*int*) – sweep ending frequency in Hz
- **rbw** (*float*) – the resolution bandwidth (RBW) in Hz of the data to be captured (output RBW may be smaller than requested)
- **device_settings** (*dict*) – attenuation and other device settings
- **mode** (*str*) – sweep mode, 'ZIF', 'SH', or 'SHN'
- **continuous** (*bool*) – set sweep to be continuously or not (once only)

Returns fstart, fstop, power_data

The fstart & fstop are the actual frequencies associated to the power_data

set_geolocation_callback(*func*, *data=None*)
Set a callback that will get called whenever the geolocation information of the device is updated. The callback function should accept two parameters. The first parameter will be the callback data that was passed in this function set_geolocation_callback(func, data, geolocation_dictionary). The geolocation_dictionary will have the following properties: - oui - seconds - altitude - longitude - speedoverground - secondsfractional - track - latitude - magneticvariation - heading See the programmer's guide for usage on each of these properties.

Parameters

- **func** – the function to be called
- **data** – the data to be passed to the function

Returns None

exception pyrf.sweep_device.**SweepDeviceError**
Exception for the sweep device to state an error() has occured

class pyrf.sweep_device.**SweepPlanner**(*dev_prop*)
An object that plans a sweep based on given paramaters.

Parameters **dev_prop** (*dict*) – the sweep device properties

class pyrf.sweep_device.**SweepSettings**
An object used to keep track of the sweep settings

2.3.5 pyrf.config

This module provides configuration classes for trigger and sweep entry setup.

class pyrf.config.**SweepEntry**(*fcstart=2400000000*, *fcstop=2400000000*, *fstep=100000000*, *fshift=0*, *decimation=1*, *hdr_gain=0*, *spp=1024*, *ppb=1*, *trig-type='none'*, *dwell_s=0*, *dwell_us=0*, *level_fstart=50000000*, *level_fstop=1000000000*, *level_amplitude=-100*, *attenuation=30*, *rfe_mode='SH'*)
Sweep entry setup for *sweep_add()*

Parameters

- **fcstart** (*int*) – starting center frequency in Hz
- **fcstop** (*int*) – ending center frequency in Hz
- **fstep** (*int*) – frequency step in Hz
- **fshift** (*int*) – the frequency shift in Hz
- **decimation** (*int*) – the decimation value (0 or 4 - 1023)
- **gain** (*str*) – the RF gain value ('high', 'medium', 'low' or 'vlow')
- **ifgain** (*int*) – the IF gain in dB (-10 - 34)

Note: ifgain parameter is deprecated, kept for a legacy device

- **hdr_gain** (*int*) – the HDR gain in dB (-10 - 30)
- **spp** (*int*) – samples per packet (256 - max, a multiple of 32) that fit in one VRT packet

- **ppb** (*int*) – data packets per block
- **dwell_s** (*int*) – dwell time seconds
- **dwell_us** (*int*) – dwell time microseconds
- **trigtype** (*str*) – trigger type ('none', 'pulse' or 'level')
- **level_fstart** (*int*) – level trigger starting frequency in Hz
- **level_fstop** (*int*) – level trigger ending frequency in Hz
- **level_amplitude** (*float*) – level trigger minimum in dBm
- **attenuator** – vary depending on the product
- **rfe_mode** (*str*) – RFE mode to be used, such as 'SH', 'SHN', 'DD', etc.

Returns a string list of the sweep entry's settings

class pyrf.config.**TriggerSettings**(*trigtype='NONE'*, *fstart=None*, *fstop=None*, *amplitude=None*)

Trigger settings for *trigger()*.

Parameters

- **trigtype** (*str*) – "LEVEL", "PULSE", or "NONE" to disable
- **fstart** (*int*) – trigger starting frequency in Hz
- **fstop** (*int*) – trigger ending frequency in Hz
- **amplitude** (*float*) – minimum level for trigger in dBm

Returns a string in the format: TriggerSettings(trigger type, fstart, fstop, amplitude)

exception pyrf.config.**TriggerSettingsError**

Exception for the trigger settings to state an error() has occured

2.3.6 pyrf.numpy_util

This file contains DSP as well as many computational functions needed for data analysis.

pyrf.numpy_util.**calculate_channel_power**(*power_spectrum*)

Return a dBm value representing the channel power of the input power spectrum. The algorithm is: Pchan = 10 * log10(sum(10^(Pdbm[i]/10))) where i = start_bint to stop_bin

Reference: http://uniteng.com/index.php/2013/07/26/channel-power-measurements/ However, instead of calculating over the whole bandwidth as in the ref link, this fn only needs to calculate between the given power_spectrum range.

Parameters **power_spectrum** (*list*) – an array containing the power spectrum to be used for the channel power calculation

Returns the channel power result

pyrf.numpy_util.**calculate_occupied_bw**(*pow_data*, *span*, *occupied_perc*)

Return the occupied bandwidth of a given percetage of the spectrum, in Hz. The algorithm uses the idea presented in a Keysight's method

Parameters

- **pow_data** (*list*) – spectral data to be analyzed, in dBm

- **span** (*int*) – span of the given spectrum, in Hz
- **occupied_perc** (*float*) – Percentage of the power to be measured

Returns float values of the occupied bandwidth (in Hz), the percentage of the left bin span, and the percentage of the right bin span

pyrf.numpy_util.**calibrate_time_domain**(*power_spectrum*, *data_pkt*)

Return a list of the calibrated time domain data

Parameters

- **power_spectrum** (*list*) – spectral data of the time domain data
- **data_pkt** (pyrf.vrt.DataPacket) – a RTSA VRT data packet

Returns a list containing the calibrated time domain data

pyrf.numpy_util.**compute_fft**(*dut*, *data_pkt*, *context*, *correct_phase=True*, *iq_correction_wideband=True*, *hide_differential_dc_offset=True*, *convert_to_dbm=True*, *apply_window=True*, *apply_spec_inv=True*, *apply_reference=True*, *ref=None*, *decimation=1*)

Return an array of power values in dBm by computing the FFT of the input data and adjusted with the input reference level.

Parameters

- **dut** (pyrf.devices.thinkrf.WSA) – WSA device
- **data_pkt** (pyrf.vrt.DataPacket) – packet containing samples
- **context** (*dict*) – context values, such as 'bandwidth', 'reflevel', etc.
- **correct_phase** (*bool*) – apply phase correction for captures with IQ data or not
- **iq_correction_wideband** (*bool*) – apply wideband IQ correction or not
- **hide_differential_dc_offset** (*bool*) – mask the differential DC offset present in captures with IQ data or not
- **convert_to_dbm** (*bool*) – convert the output values to dBm or not
- **apply_window** (*bool*) – apply windowing to FFT function or not
- **apply_spec_inv** (*bool*) – apply spectral inversion to the FFT bin or not. *Recommend to leave as default*
- **apply_reference** (*bool*) – apply reference level correction or not
- **ref** (*float*) – a reference value to apply to the noise level
- **decimation** (*int*) – the decimation value (1, 4 - 1024)

Returns numpy array of spectral data in dBm, as floats

2.3.7 pyrf.util

pyrf.util.**capture_spectrum**(*dut*, *rbw=None*, *average=1*, *dec=1*, *fshift=0*)

Captures a BLOCK of data, computes and returns its spectral data, and the usable start and stop frequencies corresponding to the RTSA's current configuration

Parameters

- **rbw** (*int*) – rbw of spectral capture (Hz) (will round to nearest native RBW)
- **average** (*int*) – number of capture iterations

- **dec** (*int*) – decimation factor applied
- **fshift** (*int*) – the fshift applied, in Hz

Returns (fstart, fstop, pow_data) where pow_data is a list

pyrf.util.**read_data_and_context**(*dut*, *points=1024*)

Initiate capture of one VRT data packet, wait for and return data packet and collect preceeding context packets.

Parameters **points** (*int*) – Number of data points to capture

Returns (data_pkt, context_values)

Where *context_values* is a dict of {field_name: value} items from all the context packets received.

2.3.8 pyrf.vrt

Classes and functions for handling and parsing VRT packets captured from thinkRF's RTSA.

class pyrf.vrt.**ContextPacket**(*packet_type*, *count*, *size*, *tmpstr*, *has_timestamp*)

A Context Packet received from *read()*. See VRT section of the product's Programmer's Guide for more information.

Parameters

- **packet_type** – VRT packet type
- **count** – VRT packet counter (see VRT protocol)
- **size** (*int*) – The VRT packet size, less headers and trailer words
- **tmpstr** – hold the raw data for parsing
- **has_timestamp** (*bool*) – to indicate timestamp is available with the packet

fields

a dict containing field names and values from the packet

is_context_packet(*ptype=None*)

Parameters **ptype** (*str*) – "Receiver", "Digitizer" or None for any packet type

Returns True if this packet matches the *ptype* passed

is_data_packet()

To indicate this VRT packet is not of data type as it's a ContextPacket

Returns False

class pyrf.vrt.**DataPacket**(*count*, *size*, *stream_id*, *tsi*, *tsf*, *payload*, *trailer*)

A Data Packet received from *read()*

data

a *pyrf.vrt.IQData* object containing the packet data

is_context_packet(*ptype=None*)

Returns False

is_data_packet()

Returns True

class pyrf.vrt.**IQData**(*binary_data*)
Data Packet values as a lazy collection of (I, Q) tuples read from *binary_data*.

This object behaves as an immutable python sequence, e.g. you may do any of the following:

```
points = len(iq_data)

i_and_q = iq_data[5]

for i, q in iq_data:
    print i, q
```

numpy_array()
Return a numpy array of I, Q values for this data

exception pyrf.vrt.**InvalidDataReceived**

pyrf.vrt.**vrt_packet_reader**(*raw_read*, *with_buffer=False*)
Read a VRT packet, parse it and return an object with its data.

Implemented as a generator that yields the result of the passed *raw_read* function and accepts the value sent as its data.

Parameters **raw_read** (*list*) – VRT packet of raw data (bytes)

2.4 Examples

This section explains **some** of the examples included with the PyRF3 source code (see the **examples** folder for other examples).

Typical Usage:

```
Linux: python3.x <example_file>.py [device_IP_when_needed]
Windows: py-3.x <example_file>.py [device_IP_when_needed]
```

where 3.x is the python version you have (eg: 3.6, 3.7 or 3.8, etc.).

2.4.1 discovery.py / twisted_discovery.py

These examples detect RTSA devices on the local network.

Example output:

```
R5700-427 180601-661 1.5.0 10.126.110.133
R5500-408 171212-007 1.5.0 10.126.110.123
R5500-418 180522-659 1.4.8 10.126.110.104
```

2.4.2 show_i_q.py / twisted_show_i_q.py

These simple examples connect to a device of IP specified on the command line, tunes it to a center frequency of 2.450 MHz, then reads and displays one capture of 1024 i, q values.

Example output (truncated):

```
0,-20
-8,-16
0,-24
-8,-12
0,-32
24,-24
32,-16
-12,-24
-20,0
12,-32
32,-4
0,12
-20,-16
-48,16
-12,12
0,-36
4,-12
```

2.4.3 pyqtgraph_plot_single_capture.py / pyqtgraph_plot_block.py

These examples connect to a device of IP specified on the command line, tunes it to a center frequency, then continually capture and display the computed spectral data using pyqtgraph.

2.4.4 pyqtgraph_plot_sweep.py

This example connects to a device of IP specified on the command line, makes use of sweep_device.py to perform a single sweep entry monitoring and plots computed spectral results using pyqtgraph.

2.4.5 matplotlib_plot_sweep.py

This example connects to a device specified on the command line, and plots a large sweep of the spectrum using NumPy and matplotlib.

2.4.6 pyqtgraph_plot_flattening_async.py / pyqtgraph_plot_flattening_sync.py

These examples make use of sweep_device.py and asynchronous or synchronous, respectively, Twisted callback to perform sweeping and plot with spectral flattening turned on and off to show the differences.

Usage:

```
Linux: python3.x run_gui.py <device_ip>
Windows: py-3.x run_gui.py <device_ip>
```

where 3.x is the python version you have (eg: 3.6, 3.7 or 3.8, etc.).

2.5 Change Logs

2.5.1 PyRF3 3.0.6

Changes in this version

- Fixed decimation default to 1 in config.py
- **Changes in thinkrf.py:**
  - fixed gain handling for when not applicable to a device
  - the reset SCPI command is corrected for reset() function
  - added a catch of when system error query returns nothing
  - removed stream_status() function as this is not applicable for R5xx0
- Added 'seconds' and 'secondsfractional' to geolocation printout in sweep_device.py
- Added to some sweep examples Geolocation data when applicable
- The release package now also includes a wheel package, besides the egg package
- Added PyRF3 installation instruction with wheel package using pip
- Added header information to each file

2.5.2 PyRF3 3.0.5

- Added 2 new examples manual_sweep.py and twisted_manual_sweep.py
- Corrected fstart and fstop in SweepEntry to fcstart and fcstop, respectively, to correctly reflect these sweep entry's frequency values as the 'center' frequencies being programmed in the device, not the starting or stoping frequencies of the sweep span in a display for instance.

2.5.3 PyRF3 3.0.4

New feature:

- Added to sweep-device the ability to handle more than one packet per block (hence allows for smaller RBWs down to 1 Hz).
- **Warning**: usage of small RBWs (in the Hz range) could cause very slow sweep time due to the large data points for processing.

Changes in this version:

- Adjusted sweep-device's frequencies to the tuning resolution.
- Changed MIN_FREQ to be 100kHz as data below this range is not usable.
- Corrected DD's PASS_BAND_CENTER to be 0 (not at half the operating bandwidth).
- When used sweep-device in DD range (0-50MHz), capture_power_spectrum() now returned only the data of the requested range (i.e. not the full 62.5MHz).
- Fixed capture_time_domain() and compute_spp_ppb() to handle decimation properly.
- Fixed adjust_usable_fstart_fstop() and compute_usable_bins() to handle DD's range properly.
- Fixed trim_to_usable_fstart_fstop() to handle DD's data range.
- Fixed capture_spectrum() to handle decimation and frequency shift properly when used and to stitch packets in a block properly.
- Added checking throughout the API for improper decimation used with HDR.
- Improved SPP & PPB input value handling.
- Corrected some functions' comment.

- Fixed peak calculation in some examples.

2.5.4 PyRF3 3.0.2 & 3.0.3

- Improved SCPI set/get handling.
- Improved trace stream start/stop handling.
- Corrected some information in the Reference Guide.
- Added usage information into examples and improved some examples.

2.5.5 PyRF3 3.0.1

2019-09-26

- Added support for R55x0-408P products.
- Added 2 new examples to show how to use sync setup or Twisted async, with flattening.
- New algorithm for *calculate_occupied_bw()* method.
- Updated the documentation for PyRF3.
- Updated existing examples to work with PyRF3 changes.
- Fixed *SweepEntry* to handle the parameters properly.
- Fixed handling of IQ_OUTPUT_CONNECTOR for R57x0.
- Fixed compute_usable_bins() handling of SHN mode and providing undesired 'usable bins' during decimation.
- Corrected the center frequency value for DD mode to 0, not 31.25 MHz.

2.5.6 PyRF3 3.0.0

2019-08-31

New Addition:

- Migrated and upgraded former pyRF v2.x.0 source to work with Python3.
- Added spectral flattening methods to support RTSA devices with flattening vectors.
- `scpiget()` now has a timeout option.
- Increased SPP max to 65504.
- RBW could go down to Hz range.

Changes:

- Updated source code and examples to work with pyqtgraph and PySide2 in Python3.
- Adjusted twisted_async.py to work Twisted in Python3.
- Restructured thinkrf_properties.py.
- Improved handling of data capture in capture_device.py, as well as RBW, SPP and PPB parameters.
- Improved sweep-device handling, including allow for smaller RBWs due to larger SPPs used.

- Removed support for legacy thinkRF products (WSA4000, WSA5000 and their equivalent) and their associated features.

Fixes:

- Fixed issues found in pyrf version and added more error handling.

2.5.7 PyRF 2.10.0 and earlier

Refers to pyRF source for the python 2.7 version of the PyRF.

CHAPTER

THREE

CONTACT US

thinkRF Support website provides online Documentation and Resources for resolving technical issues with thinkRF products. For all customers who hold a valid end-user license, thinkRF provides technical assistance 9 AM to 5 PM Eastern Time, Monday to Friday. Contact through our online support or by calling +1.613.369.5104.

PYTHON MODULE INDEX

p

INDEX

L

M

N

P

R

S

T

V

W