API for RF receivers including ThinkRF WSA4000

Release 0.4.0

ThinkRF Corporation

Oct 11, 2018

Contents

Contents:

CHAPTER 1

Manual

1.1 Installation

Install from PyPI:

```
pip install pyrf
```

Or Obtain the Development Version:

```
git clone git://github.com/pyrf/pyrf.git
```

(Or Download a Stable Release)

Then Install from Source or Extracted Tarball/Zip file:

```
python setup.py install
```

1.1.1 Installing GUI Requirements

On Debian/Ubuntu:

```
apt-get install python-pyside python-twisted python-numpy \
     python-zope.interface python-pip
pip install -e git://github.com/pyrf/qtreactor.git#egg=qtreactor
```

On Windows:

Download and install:

- 32-bit version of Python 2.7

Find the latest version of each of the following and install:

- NumPy for 32-bit Python 2.7 e.g. "numpy-1.6.2-win32-superpack-python2.7.exe"
- PySide for 32-bit Python 2.7 e.g. "PySide-1.1.2.win32-py2.7.exe"
- zope.interface for 32-bit Python 2.7 e.g. "zope.interface-4.0.3-py2.7-win32.egg"
- Twisted for 32-bit Python 2.7 e.g. "Twisted-12.3.0.win32-py2.7.msi"
- pywin32 for 32-bit Python 2.7 e.g. "pywin32-218.win32-py2.7.exe"

Download the latest version of qtreactor, extract it then switch to the qtreactor directory and run:

```
python setup.py install
```

1.1.2 Installing GUI Requirements from Source

On Debian/Ubuntu:

```
apt-get install qt-sdk python-dev cmake
pip install -r gui-requirements.txt
```

1.1.3 Building EXE Version of GUI

On Windows:

Install the GUI requirements above and make sure you can launch the GUI.

Find and install the latest version of py2exe for 32-bit Python2.7 e.g. "py2exe-0.6.9.win32-py2.7.exe".

Then switch to your pyrf directory and run:

```
python setup.py py2exe
```

1.2 API for WSA4000 RF Receiver

pyrf.devices.thinkrf.WSA4000 is the class that provides access to WSA4000 devices. Its methods closely match the SCPI Command Set described in the Programmers Reference available in ThinkRF Resources.

There are simple examples that use this API under the "examples" directory included with the source code.

This API may be used in a blocking mode (the default) or in an asynchronous mode with using the Twisted python library. Asynchronous modes using other libraries may be added in the future.

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

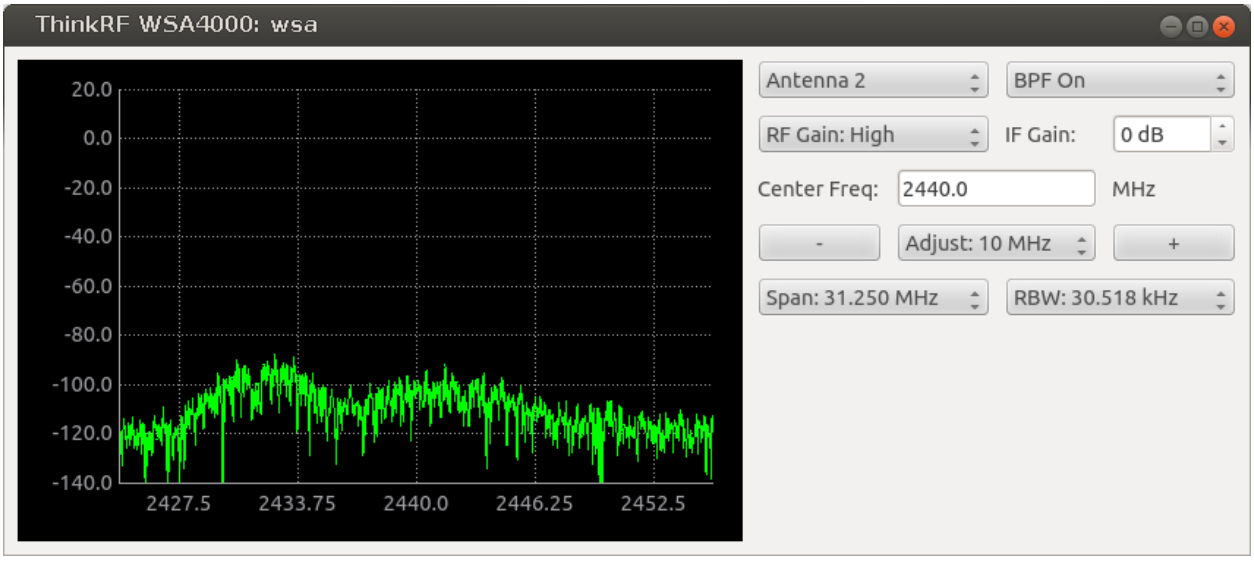
In asynchronous mode all methods will send their commands to the device and then immediately return a Twisted Deferred object. If you need to wait for the response or completion of this command you can attach a callback to the Deferred object and the Twisted reactor will call it when ready. You may choose 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 WSA4000 instance is created you must pass a *pyrf.connectors.twisted_async.TwistedConnector* instance as the connector parameter, as in *twisted_show_i_q.py*

1.3 Processing Tools

Additional PyRF tools are under active development, but will soon support processing blocks, multiprocess use and distributed processing as described in *Planned Development*.

1.4 GUI



`wsa4000gui` is a cross-platform GUI application built with the Qt toolkit and PySide bindings for Python.

The GUI may be launched with the command:

```
wsa4000gui <hostname> [--reset]
```

If *hostname* is not specified a dialog will appear asking you to enter one. If `--reset` is used the WSA will be reset to defaults before the GUI appears.

CHAPTER 2

Reference

2.1 pyrf.devices

2.1.1 .thinkrf

class pyrf.devices.thinkrf.**WSA4000**(*connector=None*)

Interface for WSA4000

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.

---

abort()

This command will cause the WSA4000 to stop the data capturing, whether in the manual trace block capture, triggering or sweeping mode. The WSA4000 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.

antenna(*number=None*)

This command selects and queries the active antenna port.

Parameters **number** – 1 or 2 to set; None to query

Returns active antenna port

capture(*spp*, *ppb*)

This command will start the single block capture and the return of *ppb* packets of *spp* samples each. 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.

Parameters

- **spp** – the number of samples in a packet
- **ppb** – the number of packets in a capture

connect(*host*)
connect to a wsa

Parameters **host** – the hostname or IP to connect to

decimation(*value=None*)
This command sets or queries the rate of decimation of samples in a trace capture. This decimation method consists of cascaded integrator-comb (CIC) filters and at every *value* number of samples, one sample is captured. The supported rate is 4 - 1023. When the rate is set to 1, no decimation is performed on the trace capture.

Parameters **value** (*int*) – new decimation value (1 or 4 - 1023); None to query

Returns the decimation value

disconnect()
close a connection to a wsa

eof()
Check if the VRT stream has closed.

Returns True if no more data, False if more data

flush()
This command clears the WSA4000'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 packet.

flush_captures()
Flush capture memory of sweep captures.

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

Parameters **freq** (*int*) – the new center frequency in Hz (0 - 10 GHz); None to query

Returns the frequency in Hz

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

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

Returns the amount of frequency shift

gain(*gain=None*)
This command sets or queries RFE quantized gain configuration. The RF front end (RFE) of the WSA4000 consists of multiple quantized gain stages. The gain corresponding to each user-selectable setting has been pre-calculated for either optimal sensitivity or linearity. The parameter defines the total quantized gain of the RFE.

Parameters **gain** – 'high', 'medium', 'low' or 'vlow' to set; None to query

Returns the RF gain value

has_data()
Check if there is VRT data to read.

Returns True if there is a packet to read, False if not

have_read_perm()
Check if we have permission to read data.

Returns True if allowed to read, False if not

id()
Returns the WSA4000's identification information string.

Returns "<Manufacturer>,<Model>,<Serial number>,<Firmware version>"

ifgain(*gain=None*)
This command sets or queries variable IF gain stages of the RFE. The gain has a range of -10 to 34 dB. This stage of the gain is additive with the primary gain stages of the LNA that are described in *gain()*.

Parameters **gain** – float between -10 and 34 to set; None to query

Returns the ifgain in dB

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 digital card.

Parameters **modulestr** – 'vco' for rf lock status, 'clkref' for mobo lock status

Returns True if locked

ppb(*packets=None*)
This command sets the number of IQ packets in a capture block

Parameters **packets** – the number of samples in a packet

Returns the current ppb value if the packets parameter is None

preselect_filter(*enable=None*)
This command sets or queries the RFE preselect filter selection.

Parameters **enable** – True or False to set; None to query

Returns the RFE preselect filter selection state

raw_read(*num*)
Raw read of VRT socket data from the WSA.

Parameters **num** – the number of bytes to read

Returns bytes

read()
Read a single VRT packet from the WSA.

request_read_perm()
Aquire exclusive permission to read data from the WSA.

Returns True if allowed to read, False if not

reset()
Resets the WSA4000 to its default settings. It does not affect the registers or queues associated with the IEEE mandated commands.

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

This is the lowest-level interface provided. Please see the Programmer's Guide for information about the commands available.

Parameters **cmd** (*str*) – the command to send

Returns the response back from the box if any

scpiset(*cmd*)
Send a SCPI command.

This is the lowest-level interface provided. Please see the Programmer's Guide for information about the commands available.

Parameters **cmd** (*str*) – the command to send

spp(*samples=None*)
This command sets or queries the number of Samples Per Packet (SPPacket).

The upper bound of the samples is limited by the VRT's 16-bit packet size field less the VRT header and any optional fields (i.e. Stream ID, Class ID, Timestamps, and trailer) of 32-bit wide words. However since the SPP must be a multiple of 16, the maximum is thus limited by 2**16 - 16.

Parameters **samples** – the number of samples in a packet or None

Returns the current spp value if the samples parameter is None

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 WSA4000 whenever data is available.

Parameters **stream_id** – optional unsigned 32-bit stream identifier

stream_status()
This query returns the current running status of the stream capture mode.

Returns 'RUNNING' or 'STOPPED'

stream_stop()
This command stops the stream capture. After receiving the command, the WSA system will stop when the current capturing VRT packet is completed.

sweep_add(*entry*)
Add an entry to the sweep list

Parameters **entry** (pyrf.config.SweepEntry) – the sweep entry to add

sweep_clear()
Remove all entries from the sweep list.

sweep_read(*index*)
Read an entry from the sweep list.

Parameters **index** – the index of the entry to read

Returns sweep entry

Return type *pyrf.config.SweepEntry*

sweep_start(*start_id=None*)
Start the sweep engine.

sweep_stop()
Stop the sweep engine.

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** (pyrf.config.TriggerSettings) – the new trigger settings; None to query

Returns the trigger settings

2.2 pyrf.connectors

2.2.1 .blocking

class pyrf.connectors.blocking.**PlainSocketConnector**
This connector makes SCPI/VRT socket connections using plain sockets.

connect(*host*)

disconnect()

eof()

has_data()

raw_read(*num*)

scpiget(*cmd*)

scpiset(*cmd*)

sync_async(*gen*)
Handler for the @sync_async decorator. We convert the generator to a single return value for simple synchronous use.

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

2.2.2 .twisted_async

class pyrf.connectors.twisted_async.**SCPIClient**

connectionMade()

dataReceived(*data*)

scpiget(*cmd*)

scpiset(*cmd*)

class pyrf.connectors.twisted_async.**SCPIClientFactory**

buildProtocol(*addr*)

clientConnectionFailed(*connector*, *reason*)

clientConnectionLost(*connector*, *reason*)

startedConnecting(*connector*)

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

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.

connect(*host*)

disconnect()

eof()

raw_read(*num_bytes*)

scpiget(*cmd*)

scpiset(*cmd*)

sync_async(*gen*)

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*)

dataReceived(*data*)

eof = False

makeConnection(*transport*)

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

buildProtocol(*addr*)

clientConnectionFailed(*connector*, *reason*)

clientConnectionLost(*connector*, *reason*)

startedConnecting(*connector*)

2.3 pyrf.config

class pyrf.config.**SweepEntry**(*fstart=2400000000*, *fstop=2400000000*, *fstep=100000000*, *fshift=0*, *decimation=0*, *antenna=1*, *gain='vlow'*, *ifgain=0*, *spp=1024*, *ppb=1*, *trigtype='none'*, *level_fstart=50000000*, *level_fstop=10000000000*, *level_amplitude=-100*)
Sweep entry for *pyrf.devices.thinkrf.WSA4000.sweep_add()*

Parameters

- **fstart** – starting frequency in Hz
- **fstop** – ending frequency in Hz
- **shift** – the frequency shift in Hz
- **decimation** – the decimation value (0 or 4 - 1023)
- **antenna** – the antenna (1 or 2)
- **gain** – the RF gain value ('high', 'medium', 'low' or 'vlow')
- **ifgain** – the IF gain in dB (-10 - 34)
- **spp** – samples per packet
- **ppb** – packets per block

- **trigtype** – trigger type ('none' or 'level')
- **level_fstart** – level trigger starting frequency in Hz
- **level_fstop** – level trigger ending frequency in Hz
- **level_amplitude** – level trigger minimum in dBm

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

Trigger settings for *pyrf.devices.thinkrf.WSA4000.trigger()*.

Parameters

- **trigtype** – "LEVEL" or "NONE" to disable
- **fstart** – starting frequency in Hz
- **fstop** – ending frequency in Hz
- **amplitude** – minumum level for trigger in dBm

exception pyrf.config.**TriggerSettingsError**

2.4 pyrf.numpy_util

pyrf.numpy_util.**compute_fft**(*dut*, *data_pkt*, *context*)

Return an array of dBm values by computing the FFT of the passed data and reference level.

Parameters

- **dut** (pyrf.devices.thinkrf.WSA4000) – WSA device
- **data_pkt** (pyrf.vrt.DataPacket) – packet containing samples
- **context** – dict containing context values

This function uses only *dut.ADC_DYNAMIC_RANGE*, *data_pkt.data* and *context['reflevel']*.

Returns numpy array of dBm values as floats

2.5 pyrf.util

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

Wait for and capture a data packet and collect preceeding context packets.

Returns (data_pkt, context_values)

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

2.6 pyrf.vrt

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

A Context Packet received from *pyrf.devices.thinkrf.WSA4000.read()*

fields

a dict containing field names and values from the packet

is_context_packet(*ptype=None*)

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

Returns True if this packet matches the type passed

is_data_packet()

Returns False

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

A Data Packet received from *pyrf.devices.thinkrf.WSA4000.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 similar to:

```
array([[ -44,    8],
       [ -40,   60],
       [ -12,   92],
       ...,
       [-132,   -8],
       [-124,   56],
       [ -44,   80]], dtype=int16)
```

exception pyrf.vrt.**InvalidDataReceived**

pyrf.vrt.**vrt_packet_reader**(*raw_read*)

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.

CHAPTER 3

Examples

These examples may be found in the "examples" directory included with the PyRF source code.

3.1 show_i_q.py

This example connects to a device 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.

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

import sys
from pyrf.devices.thinkrf import WSA4000

# connect to wsa
dut = WSA4000()
dut.connect(sys.argv[1])

# setup test conditions
dut.reset()
dut.request_read_perm()
dut.ifgain(0)
dut.freq(2450e6)
dut.gain('low')
dut.fshift(0)
dut.decimation(0)

# capture 1 packet
dut.capture(1024, 1)

# read until I get 1 data packet
while not dut.eof():
    pkt = dut.read()
```

(continues on next page)

(continued from previous page)

```
    if pkt.is_data_packet():
        break

# print I/Q data into i and q
for i, q in pkt.data:
    print "%d,%d" % (i, q)
```

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
```

3.2 plot_fft.py

This example connects to a device specified on the command line, tunes it to a center frequency of 2.450 MHz and sets a trigger for a signal with an amplitude of -70 dBm or greater between 2.400 MHz and 2.480 MHz.

When the trigger is satisfied the data is captured and rendered as a spectrum display using NumPy and matplotlib.

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

from pyrf.devices.thinkrf import WSA4000
from pyrf.config import TriggerSettings
from pyrf.util import read_data_and_context
from pyrf.numpy_util import compute_fft

import sys
import time
import math

from matplotlib.pyplot import plot, figure, axis, xlabel, ylabel, show

# connect to wsa
dut = WSA4000()
dut.connect(sys.argv[1])

# setup test conditions
dut.reset()
dut.request_read_perm()
```

(continues on next page)

(continued from previous page)

```
dut.ifgain(0)
dut.freq(2450e6)
dut.gain('high')
dut.fshift(0)
dut.decimation(0)
trigger = TriggerSettings(
    trigtype="LEVEL",
    fstart=2400e6,
    fstop=2480e6,
    amplitude=-70)
dut.trigger(trigger)

# capture 1 packet
data, context = read_data_and_context(dut, 1024)

# compute the fft of the complex data
powdata = compute_fft(dut, data, context)

# setup my graph
fig = figure(1)
axis([0, 1024, -120, 20])

xlabel("Sample Index")
ylabel("Amplitude")

# plot something
plot(powdata, color='blue')

# show graph
show()
```

3.3 twisted_show_i_q.py

This is a Twisted version of the show_i_q.py example above.

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

import sys
from pyrf.devices.thinkrf import WSA4000
from pyrf.connectors.twisted_async import TwistedConnector

from twisted.internet import reactor, defer
import twisted.python.log

# connect to wsa
dut = WSA4000(connector=TwistedConnector(reactor))

@defer.inlineCallbacks
def show_i_q():
    yield dut.connect(sys.argv[1])

    # setup test conditions
    yield dut.reset()
    yield dut.request_read_perm()
```

(continues on next page)

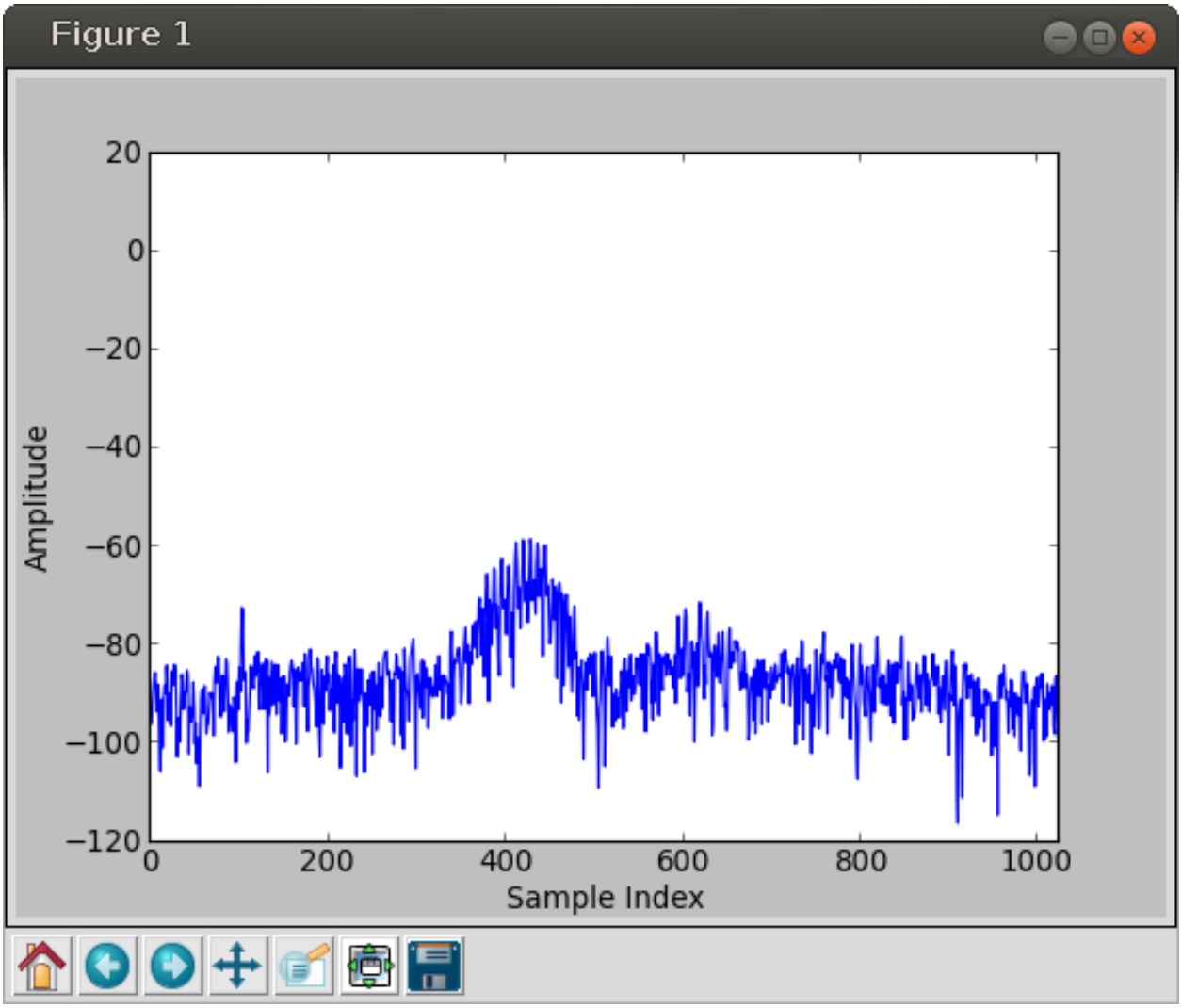


Fig. 1: Example output of `plot_fft.py`

(continued from previous page)

```
    yield dut.ifgain(0)
    yield dut.freq(2450e6)
    yield dut.gain('low')
    yield dut.fshift(0)
    yield dut.decimation(0)

    dut.connector.vrt_callback = receive_vrt
    # capture 1 packet
    yield dut.capture(1024, 1)

def receive_vrt(packet):
    # read until I get 1 data packet
    if not packet.is_data_packet():
        return

    # print I/Q data into i and q
    for i, q in packet.data:
        print "%d,%d" % (i, q)
    # exit
    reactor.stop()

d = show_i_q()
d.addErrback(twisted.python.log.err)
reactor.run()
```

CHAPTER 4

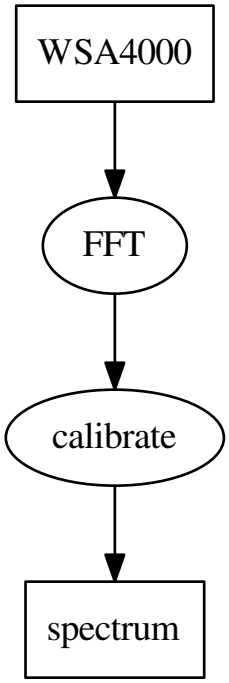
Planned Development

4.1 Processing Blocks

See also:

issue on github.

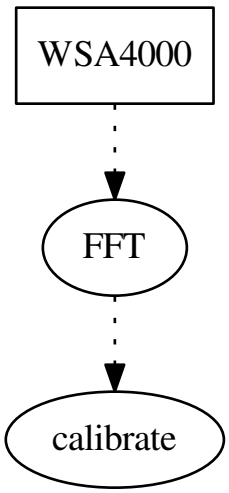
The simplest Twisted use will have all processing blocks in the same process, much like the current GUI example but without the problem of the UI freezing when no data is arriving from the device. This mode is the simplest for the programmer and incurrs no cost for passing data from one processing block to the next.




```
wsa = WSA4000(host)
fft = fft_block(wsa)
calibrate = calibrate_block(fft)
spectrum = spectrum_display(calibrate)
```

Processing blocks will use Python interfaces based on zope.interface to describe connections that may be made from consumer to producer.

Consumers will connect to their configured producers only if they are not producers (e.g. a graph renderer) or if all their required producer interfaces have consumers connected.




```
wsa = WSA4000(host)
fft = fft_block(wsa)
calibrate = calibrate_block(fft)
```

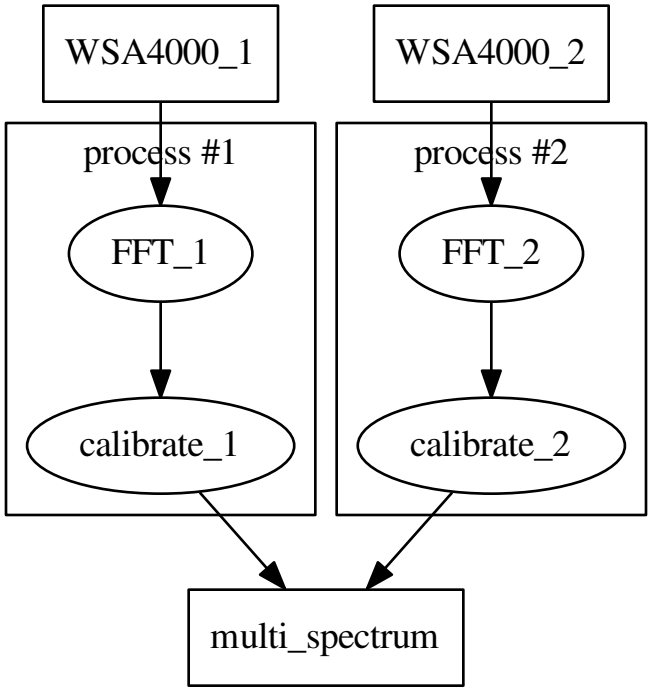
4.2 Multiprocess and HTTP

See also:

issue on github.

Using multiple cores for data processing will be accomplished by grouping some or all processing blocks into separate processes. These processes will pass data with long-polling HTTP requests at the boundaries.

HTTP Headers will be used to indicate the type of data/packet being sent. The body will contain the raw packet bytes.




```
process1 = process()
process2 = process()
wsa1 = WSA4000(host1)
fft1 = fft_block(wsa1, proc=process1)
calibrate1 = calibrate_block(fft1, proc=process1)
wsa2 = WSA4000(host2)
fft2 = fft_block(wsa2, proc=process2)
calibrate2 = calibrate_block(fft2, proc=process2)
multi_spectrum = multi_spectrum_display(calibrate1, calibrate2)
```

4.3 Distributed

See also:

issue on github.

HTTP servers work across different machines without modification. Setting up a distributed processing chain across separate machines will be possible to set up, but will require some more manual configuration than multiprocess configuration.

Authentication between machines is outside the scope of this library.

Extending the process block deployment across machines in an easier way (with ssh, for example) is a possible future enhancement.

CHAPTER 5

Hardware Support

This library currently supports development for the WSA4000 Platform, but may support additional hardware in the future.

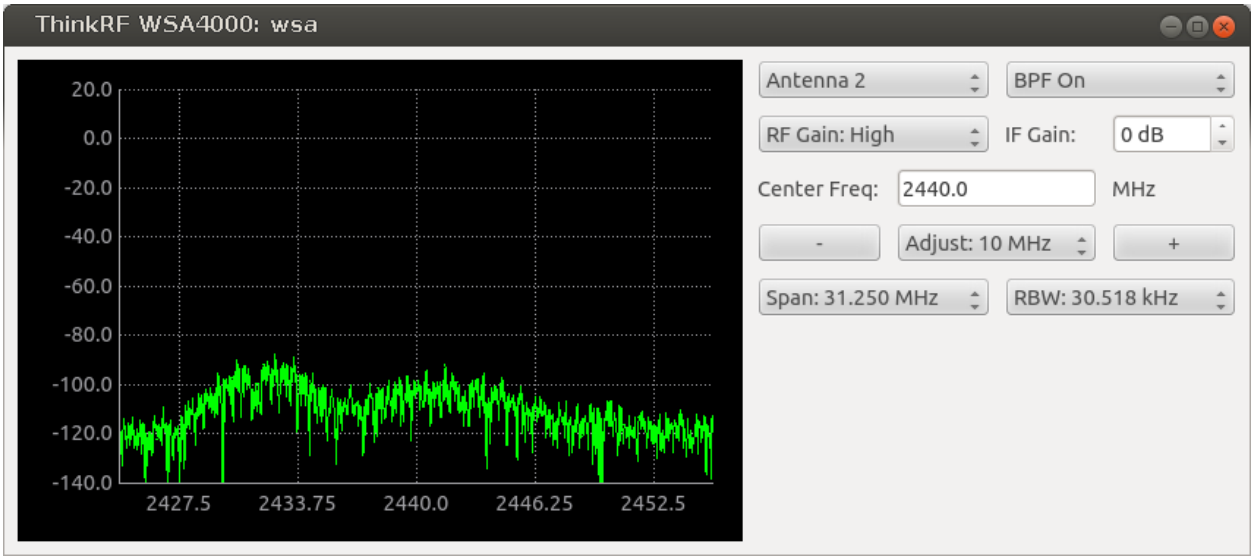
CHAPTER 6

Links

- Official github page
- Documentation for this API
- WSA4000 Documentation

CHAPTER 7

GUI Example Included



CHAPTER 8

Indices and tables

- genindex
- search

Python Module Index

p

Index