ARL-TN-0847 ● Sep 2017

US Army Research Laboratory

# US Army Research Laboratory and University of Notre Dame Distributed Sensing: Software Overview

by Neal Tesny and Daniel T Galanos

ARL-TN-0847 ● Sep 2017

US Army Research Laboratory

# US Army Research Laboratory and University of Notre Dame Distributed Sensing: Software Overview

by Neal Tesny
*Sensors and Electron Devices Directorate, ARL*

Daniel T Galanos
*Alion Science and Technology, McLean, VA*

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved OMB No. 0704-0188* |

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* September 2017 | **2. REPORT TYPE** Technical Note | **3. DATES COVERED (From - To)** |
| **4. TITLE AND SUBTITLE** US Army Research Laboratory and University of Notre Dame Distributed Sensing: Software Overview | | **5a. CONTRACT NUMBER** |
| | | **5b. GRANT NUMBER** |
| | | **5c. PROGRAM ELEMENT NUMBER** |
| **6. AUTHOR(S)** Neal Tesny and Daniel T Galanos | | **5d. PROJECT NUMBER** |
| | | **5e. TASK NUMBER** |
| | | **5f. WORK UNIT NUMBER** |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** US Army Research Laboratory, ATTN: RDRL-SER-M, 2800 Powder Mill Road, Adelphi, MD 20783-1138 | | **8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TN-0847 |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** | | **10. SPONSOR/MONITOR'S ACRONYM(S)** |
| | | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution is unlimited.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Software was developed using Python and GNU Radio to control spectrum sensing modules that are part of a distributed sensing network. The modules consist of a low-cost Raspberry Pi single-board computer and an Ettus B205mini Universal Software Radio Peripheral software-defined radio and Message Queue Telemetry Transport network infrastructure. This technical note describes the software that was developed to control the individual sensors.

**15. SUBJECT TERMS**
software-defined radio, GNU Radio, Python, spectrum sensing, distributed sensing

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | Neal Tesny |
| Unclassified | Unclassified | Unclassified | SAR | 44 | **19b. TELEPHONE NUMBER (Include area code)** 301-394-5559 |

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

## Contents

## List of Figures

## 1. Background/Introduction

This project is part of a cooperative agreement between the US Army Research Laboratory (ARL) and the University of Notre Dame (ND). The project involved developing and testing a distributed sensing network comprising low-cost RF sensors connected remotely to a central control server. This type of distributed network would benefit the Army for sensing RF signals due to its very low cost and small size.

The hardware consists of a portable Raspberry Pi single-board computer (SBC) that controlled an Ettus B205mini software-defined radio (SDR) Universal Software Radio Peripheral (USRP).The B205mini can receive signals between 70 MHz and 6 GHz and has an instantaneous bandwidth of up to 56 MHz. However, the use of USB-2 (a restriction due to the Raspberry Pi) limited the usable bandwidth to 2 MHz. A commercial mobile ad-hoc network (MANET) was set up among the central host and the remote modules. A block diagram of the hardware in an individual module is shown in Fig. 1. The sensing network consists of several nodes connected to a central server. The individual nodes continuously collect in-phase and quadrature (IQ) data and send them to the central server for processing. A block diagram of the distributed sensing concept is shown in Fig. 2.

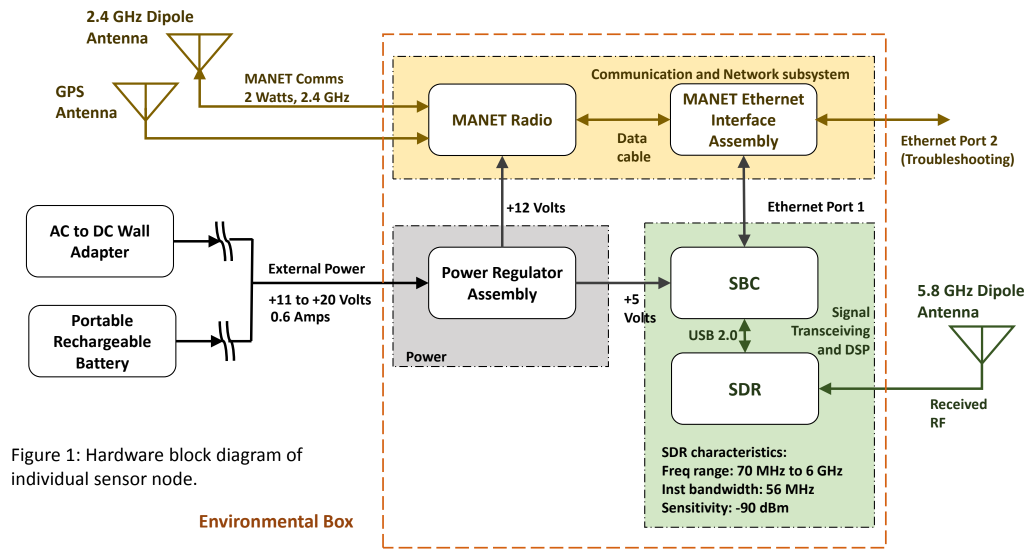
2.4 GHz Dipole Antenna
GPS Antenna
MANET Comms 2 Watts, 2.4 GHz
Communication and Network subsystem
MANET Radio
Data cable
MANET Ethernet Interface Assembly
Ethernet Port 2 (Troubleshooting)
+12 Volts
Ethernet Port 1
AC to DC Wall Adapter
Portable Rechargeable Battery
External Power
+11 to +20 Volts 0.6 Amps
Power Regulator Assembly
Power
+5 Volts
SBC
USB 2.0
Signal Transceiving and DSP
SDR
5.8 GHz Dipole Antenna
Received RF
SDR characteristics:
Freq range: 70 MHz to 6 GHz
Inst bandwidth: 56 MHz
Sensitivity: -90 dBm
Figure 1: Hardware block diagram of individual sensor node.
Environmental Box

**Fig. 1 Block diagram of the hardware in an individual module**

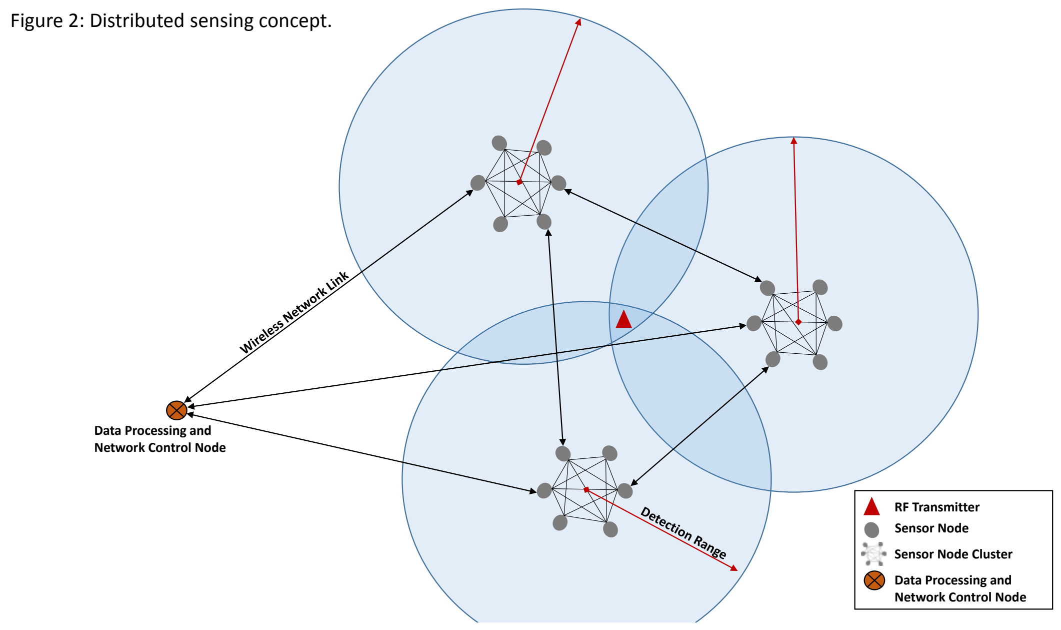
Figure 2: Distributed sensing concept.
Wireless Network Link
Data Processing and Network Control Node
Detection Range
RF Transmitter
Sensor Node
Sensor Node Cluster
Data Processing and Network Control Node

**Fig. 2 Distributed sensing concept**

The software was made up of a GNU Radio (GR) module that was interfaced with a Python module. This report focuses on the software that was developed by ARL for this effort.

GNU Radio is a free, open-source software development toolkit that provides signal processing blocks to allow users to implement SDRs and signal-processing systems.

## 2. Software Description

The software is written in Python and contains a GR module that was developed using GNU Radio Companion (GRC). GRC is the graphical interface that is used to design and generate GR modules. It translates a graphically designed flowchart to a Python script that can communicate with SDRs. The script it generates uses commands linked to GR blocks that are written in C++ for speed and efficiency. The software runs in the Ubuntu Linux operating system.

The following are the 4 main functions the software performs:

- Controls the USRP hardware and receives its data.
- Processes the data received from the USRP and performs frequency transforms on them.
- Communicates with the remote server.

- Obtains the local GPS coordinates from the MANET receiver.

## 2.1 GRC Blocks

The GR blocks that are used by the program include the following:

- USRP: This is the block that controls the B205 hardware and pulls out the data from it. The data are IQ data that were downconverted in frequency to baseband. The baseband is the instantaneous bandwidth of the USRP. This block has a long list of settable parameters, the more important ones being channel frequency, sample rate, and address (if Ethernet is used).
- FFT block: This block ended up not being used in GRC. Instead, we did fast Fourier transforms (FFTs) in Python using the Welch method.
- Stream to vector: This block converts a data stream into vectors of a fixed length.
- Message sink: This allows data to be sent to the Python module.
- Null sink: This works along with the Message Sink to provide a null output for the message sink.
- Vector decimator: This reduces the number of vectors being sent through the data stream to the Python module by allowing every *nth* vector to pass through. It is set by a through-rate setting.

## 2.2 GR/Python Data Interface

The data are transferred from the GR module to the Python portion of the program via a Message Sink block that is terminated in a null sink. The vector data stream from the USRP is fed into the Message Sink. The Message Sink is then accessed in program to obtain the data using the command **tb.msgq_out.delete_head() .to_string()**. This reads in a structured block of data, which is then unpacked and formatted into a numerical array.

## 2.3 Functions

The individual methods that the program uses to perform its functions are described:

- On_connect: This callback is run when the client receives a CONNACK response from the server. It subscribes to the Message Queue Telemetry Transport (MQTT) channels that are needed to send and receive messages.

- On_message: This is the callback that is run when a PUBLISH message is received from the server.
- parseJson: This method parses JavaScript Object Notation (JSON) data into standard format data that are usable by Python. It then calls the proper method to run based on the message payload.
- send_heartbeat: This method sends a message to the server at 10-s intervals. The message consists of the mac_address, latitude, longitude, and altitude.
- send_status: This method sends the status of the program to the server when it is requested. The parameters that are sent in the message are the following:

  "job_id"

  "disk_used"

  "disk_free"

  "mac_address"

  "IP_addr"

  "latitude"

  "longitude"

  "altitude"
- set_location: This method sets the location coordinates to that which is specified in a message directed to this remote node from the server. It allows the server to tell each remote node its GPS coordinates.
- coordinates.set_time: This method sets the system time of the remote node to that specified in an incoming message from the server with this command.
- acquire_GPS: This method is run when directed by the server. It acquires the GPS coordinates from the MANET receiver.
- change_config: This method allows the server to change the node's program configuration remotely. Changeable parameters include the following:

  ("check_in_server")

  ("master_controller")

("hostname")

"short_name")

("group_name")

("rh_software_version")

("system_version")

("hardware_version")

("rh_hardware_version")

("IP_addr")

- send_config: This method sends the current configuration parameters to the remote server when it is requested. The parameters sent are the same ones as those listed under the "change_config" method.
- Scan: This method performs a data acquisition scan when a "scan" request comes in from the remote server.
- parse_sdr_data: This method extracts an array of IQ data from a structured block of data, which is output from the GR Message Sink.
- format_data_for_tx: This method puts the acquired signal data into the proper format required by the remote server. This format is essentially one large Python dictionary with the IQ data formatted into a single string sequence. The output format is described in Appendix A. The Welch method that is used to perform the FFTs is described in Appendix B.
- send_data: This method puts the formatted data into JSON format and transmits them to the remote server.
- Main: This is the method that is run when the program starts, which calls the top block module with the GR components. A listing of the entire program is given in Appendix C.

## 2.4 Settable Parameters

The parameters that are settable in this program include the following:

- Window type: This is a string input specifying the windowing function to use before FFT operations are performed. These are described in Appendix D. An example is "hamming".

- Nfft: This is the FFT size that is used in the frequency transform. Acceptable values are integers greater than 16.
- Sample rate: This is the rate at which the SDRs sample the data. The default value is 2 MHz because of the bandwidth being limited by the USB-2 cable connection. This is of double type.
- Gain: This is the gain of the SDR. This is of double type between 0 and 76.
- Freq0: This is the center frequency of the band being scanned. This is of double type. The range is 70E6 to 6E9.

There are additional values that can be set internally but cannot be changed remotely:

- Vector size: This is the size of the data blocks that are output from the GR module to the Python module. The default value for this is 1024. Integer values are acceptable.
- Vector rate: The GR module uses a vector decimator, which reduces the rate at which data blocks are transferred to the Python module. It was found that the overall data rate is maximized by setting this to 100 vectors per second, which is the default value.

The following are changeable/nonchangeable items for scan requests:

- "fmin": This is settable via the scan command.
- "fmax": This is settable via the scan command.
- "fs": This is settable via the scan command.
- "overlap": This is settable via the scan command.
- "nsamples": This is settable via the scan command.
- "gain": This is settable via the scan command. The B205 receive frontends have 76 dB (or 73?) of available gain.
- "nfft": The FFT size cannot be changed after the program is started. It can be set manually in code before the program is run o/a line 101. Default is 1024. This set by changing the value of the variable “vector_size”.
- "noverlap": This is not directly changeable, but is modified when the “overlap” value is changed. It is set to nfft*overlap.
- "window": This value cannot be changed after the program is started. This can be set manually in code before running the program. This is a window

that is recognized by GNU Radio, such as “window.blackmanharris”. To change it, go to Line 135 and change the name of the window argument in the line where it calls the “fft.fft_vcc” function. One must also change the variable “window_type”, which stores the name of the window type that is saved in the metadata of the output. Two commented-out lines of how to change the window to “Hanning” should be given in the code. Selectable windows that GNU Radio can use are listed in Appendix D.

- "zero_pad_to": not implemented.
- "detrend": not implemented.

## 3. Network

The network of devices comprises nodes and one central server. The server does the following:

- Stores all data collected by the nodes.
- Disseminates instructions to the nodes.
- Interfaces the user to the network of nodes

Nodes are able to exchange information by leveraging a MANET architecture. Each node is equipped with a MANET radio that provides access to the MANET. Information can be sent between any 2 nodes within signal range and, additionally, information can be relayed along a set of nodes such that if nodes form a connected graph, information can be passed between any pair of nodes on the network. The MANET is self-forming; as nodes come online or go out of range the MANET continuously adapts. All of this is taken care of on the MANET card and is transparent to the devices connected to it.

The central server did not include a Dynamic Host Configuration Protocol capability, so nodes were configured with static IP addresses. To better take advantage of the flexibility of the MANET, future efforts should leverage dynamically allocated IP addresses and, potentially, Domain Name System.

### 3.1 MQTT

MQTT is a lightweight publish–subscribe messaging protocol. MQTT requires a designated message broker to distribute messages and track subscriptions. Clients connect to a broker, subscribe to various topics, and publish messages to topics. The broker then distributes the published messages to clients that are subscribed to the corresponding topics.

The MQTT protocol was implemented for this effort to send and receive data/instructions between all devices on the MANET. Each node connects to the central server (broker) and subscribes to the following topics, where mac_address is the unique media access control (MAC) address of the node:

'radiohound/clients/command_to_all'

'radiohound/clients/command/' + mac_address

'radiohound/clients/data/' + mac_address

'radiohound/clients/status/' + mac_address

An instruction may now be published to

'radiohound/clients/command_to_all'

Because all connected nodes are subscribed to this topic, all connected nodes receive this instruction. For a full list and description of topics, see *RadioHound API Documentation* by Nik Kleber and Gonzalo Martinez.[1]

### 3.2 GPS

Knowing the physical location of each module is key for activities such as emitter geolocation. The MANET card includes a GPS receiver, and each module is equipped with a GPS antenna. Each MANET card also hosts a configuration website that can control the MANET (individual node or entire network) and provide application programming interface (API) functionality. Nodes queried their GPS information from this API over Transmission Control Protocol/IP.

## 4. Conclusions

Software programming has been written to support distributed sensing using low-cost, highly portable computers and SDRs. The software was demonstrated successfully at a joint field test between ARL and ND in June 2017.

## 5. References

Kleber N, Martinez G. University of Notre Dame, South Bend, IN. RadioHound API documentation. Personal communication, 2017.

INTENTIONALLY LEFT BLANK.

## Appendix A. Output Format, In-Phase and Quadrature (IQ) Data in the Header

The data are output to the remote host in a Python dictionary format that contains all the data and parameters. The frequency transformed data are formatted as a single string variable. The metadata contain the in-phase and quadrature (IQ) data and settable parameters and are formatted as a separate Python dictionary within the overall data dictionary. The format of the data is as follows:

```
"software_version"
"timestamp"
"gain"
"data"
"short_name"
"uncertainty"
"longitude"
"sample_rate"
"mac_address"
"latitude"
"center_frequency"
"metadata": {"nfft"
        "fmax"
        "scan_type"
        "detrend"
        "report_type"
        "nsamples"
        "antenna"
        "actualGain"
        "overlap"
        "window"
        "fmin"
        "zero_pad_to"
        "noverlap"
        "iq_data"
```

# Appendix B. Welch's Method

Welch’s method is form of spectral energy estimation that uses a form of time averaging of overlapping power measurements. It first applies a filtering window such as a Hann window to each time segment. It then performs a fast Fourier transform (FFT) on each segment. Time averaging is then performed on the overlapping FFT segments reducing the variance of the individual power measurements.

Input parameters for Welch’s method in this program include the following:

- Data: the time-domain vector of data.
- Fs: the sample rate at which the data were collected.
- Window: the windowing, or tapering, function used, for example, Hanning.
- Nperseg: the number of points to use for the FFT.

## Appendix C. Program Listing

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Top Block
# Generated: Mon May 22 10:17:11 2017
##################################################

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt4 import Qt
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import fft
from gnuradio import gr
from gnuradio import uhd
from gnuradio.eng_option import eng_option
from gnuradio.fft import window
from gnuradio.filter import firdes
from optparse import OptionParser
from distutils.version import StrictVersion
#import osmosdr
import sys
import time
import struct
import numpy
import datetime
import numpy as np
import base64
from scipy import signal


import paho.mqtt.client as mqtt
import json
import urllib2
import os

import requests
from requests.packages.urllib3 import exceptions
requests.packages.urllib3.disable_warnings(exceptions.Insecure
RequestWarning)

#import matplotlib.pyplot as pyplot
#broker_ip = '192.168.3.3'
broker_ip = '127.0.0.1'
mac_address = 'b8:27:eb:09:99:3f'
```

```
tb = None
client = mqtt.Client()
last_heartbeat_time = time.time()
last_trace_time = time.time()
send_data_flag = 0
msg_queue = []
latitude = 0.0
longitude = 0.0
altitude=0.0
check_in_server = "check_in_server0"
master_controller = "master_controller0"
hostname = "hostname0"
short_name = "ARL_xxx"
group_name = "ARL_group0"
rh_software_version = "1.0"
system_version = "1.0"
hardware_version = "1.0"
rh_hardware_version = "1.0"
IP_addr = "192.168.3.131"
# uhd_serial_number = "30F5565" # set this for each node's B205
uhd_serial_number = "30AEBF0" # set this for each node's B205
# window_type = "window.blackmanharris"
window_type = "scipy.hanning"
nfft=1024
samp_rate_current=2e6
window_welch="hann"

class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")
        Qt.QWidget.__init__(self)
        # self.setWindowTitle("Top Block")
        # try:
        #      self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-
grc'))
        # except:
        #     pass
        # self.top_scroll_layout = Qt.QVBoxLayout()
        # self.setLayout(self.top_scroll_layout)
        # self.top_scroll = Qt.QScrollArea()
        # self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        # self.top_scroll_layout.addWidget(self.top_scroll)
        # self.top_scroll.setWidgetResizable(True)
        # self.top_widget = Qt.QWidget()
        # self.top_scroll.setWidget(self.top_widget)
        # self.top_layout = Qt.QVBoxLayout(self.top_widget)
        # self.top_grid_layout = Qt.QGridLayout()
        # self.top_layout.addLayout(self.top_grid_layout)

        #self.settings = Qt.QSettings("GNU Radio", "top_block")

#self.restoreGeometry(self.settings.value("geometry").toByteAr
ray())
```

```
        ##################################################
        # GNU Radio Variables
        ##################################################
        self.vector_size = vector_size = 1024 # block size GR
uses to send data to Python
        self.vector_rate = vector_rate = 100 # rate at which GR
sends vectors to Python
        self.samp_rate = samp_rate = 2e6
        self.gain0 = gain0 = 30
        self.freq0 = freq0 = 100e6


        ##################################################
        # Processing Variables
        ##################################################
        self.nsamples = 16384

        ##################################################
        # Message queues (added by grcconvert)
        ##################################################
        self.msgq_out   =   blocks_message_sink_0_msgq_out   =
gr.msg_queue(2)

        ##################################################
        # Blocks
        ##################################################
        self.uhd_usrp_source_0 = uhd.usrp_source(
           ",".join(("", "")),
           uhd.stream_args(
                 cpu_format="fc32",
                 channels=range(1),
           ),
        )
        self.uhd_usrp_source_0.set_samp_rate(samp_rate)
        self.uhd_usrp_source_0.set_center_freq(freq0, 0)
        self.uhd_usrp_source_0.set_gain(gain0, 0)




        #  self.fft_vxx_0  =  fft.fft_vcc(vector_size,  True,
(window.blackmanharris(vector_size)), True, 1)
        #self.fft_vxx_0   =   fft.fft_vcc(vector_size,   True,
(window.hanning(vector_size)), True, 1)
        #window_type = "window.hanning"
        self.blocks_stream_to_vector_decimator_0             =
blocks.stream_to_vector_decimator(
           item_size=gr.sizeof_gr_complex,
           sample_rate=samp_rate,
           vec_rate=vector_rate,
           vec_len=vector_size,
        )
        self.blocks_message_sink_0                          =
blocks.message_sink(gr.sizeof_gr_complex*vector_size,
blocks_message_sink_0_msgq_out, False)
        #self.blocks_complex_to_mag_0                       =
blocks.complex_to_mag(vector_size)
        #            self.blocks_complex_to_mag_0           =
blocks.complex_to_mag_squared(vector_size)

        ##################################################
        # Connections
        ##################################################
        #    self.connect((self.blocks_complex_to_mag_0,    0),
(self.blocks_message_sink_0, 0))

self.connect((self.blocks_stream_to_vector_decimator_0,     0),
(self.blocks_message_sink_0, 0))
        #        removed        by        grcconvert:        #
self.connect((self.blocks_message_sink_0, 'msg'), (self, 0))
        #
self.connect((self.blocks_stream_to_vector_decimator_0,     0),
(self.fft_vxx_0, 0))
        #          self.connect((self.fft_vxx_0,            0),
(self.blocks_complex_to_mag_0, 0))
        self.connect((self.uhd_usrp_source_0,               0),
(self.blocks_stream_to_vector_decimator_0, 0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "top_block")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()


    def get_vector_size(self):
        return self.vector_size

    def set_vector_size(self, vector_size):
        self.vector_size = vector_size

    def get_vector_rate(self):
        return self.vector_rate

    def set_vector_rate(self, vector_rate):
        self.vector_rate = vector_rate

self.blocks_stream_to_vector_decimator_0.set_vec_rate(self.vec
tor_rate)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
self.blocks_stream_to_vector_decimator_0.set_sample_rate(self.
samp_rate)
        self.uhd_usrp_source_0.set_samp_rate(self.samp_rate)

    def get_gain0(self):
        return self.gain0

    def set_gain0(self, gain0):
        self.gain0 = gain0
        self.uhd_usrp_source_0.set_gain(self.gain0, 0)

    def get_freq0(self):
        return self.freq0

    def set_freq0(self, freq0):
        self.freq0 = freq0
        self.uhd_usrp_source_0.set_center_freq(self.freq0, 0)

##############################################################
##################
# MQTT Config
##############################################################
##################
# The callback for when the client receives a CONNACK response
from the server.
def on_connect(client, userdata, flags, rc):
    print('MQTT Connected with result code '+str(rc))

    # Subscribing in on_connect() means that if we lose the
connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe('radiohound/clients/command_to_all')

client.subscribe('radiohound/clients/command/'+mac_address)
    client.subscribe('radiohound/clients/data/'+mac_address)
    client.subscribe('radiohound/clients/status/'+mac_address)

#client.subscribe('radiohound/clients/announce/'+mac_address)

# The callback for when a PUBLISH message is received from the
server.

def on_message(client, userdata, msg):
    global msg_queue
    if len(str(msg.payload))<=300:
        print(msg.topic+' '+str(msg.payload))
    else:
        print(msg.topic+' : length is:',len(str(msg.payload)))
    msg_queue.append(msg.payload)

def parseJson(msg_payload):
    global send_data_flag
    #print "msg_payload is:",msg_payload
    #print "Line 208, msg_payload is:",msg_payload
    j = json.loads(msg_payload)

    if type(j) == dict:
        message_value = ""
        payload_value = ""
        for key, value in j.iteritems():
            if key == 'message':
                message_value = value
            elif key == 'payload':
                payload_value = value

        if message_value == 'run':
            send_data_flag = 2
        elif message_value == 'get_single_trace':
            send_data_flag = 1
        elif message_value == 'stop':
            send_data_flag = 0
        elif message_value == 'scan':
            scan(payload_value)
        elif message_value == 'set_location':
            set_location(payload_value)
        elif message_value == 'change_config':
            change_config(payload_value)
        elif message_value == 'send_config':
            send_config()
        elif message_value == 'release_location':
            acquire_GPS()
        elif message_value == 'set_time':
            set_time(payload_value)
    else:
        print "Not a dict, val is: %s" % j


##############################################################
##################
# send_heartbeat
##############################################################
##################

def send_heartbeat():
    global last_heartbeat_time
    current_time = time.time()
    if current_time - 10 > last_heartbeat_time:
        #latitude = 0.0
        #longitude = 0.0
        #altitude = 0.0

        out = """{
        "message": "HEARTBEAT",
        "payload":
         {
            "mac_address": %s
            "latitude": %f
            "longitude" %f
            "altitude" %f
         }
        }""" % (mac_address, latitude, longitude, altitude)

client.publish('radiohound/clients/announce/'+mac_address,payl
oad=out)
        last_heartbeat_time = current_time
        #print mac_address, latitude,longitude,altitude


##############################################################
##################
# send_status
##############################################################
##################

def send_status(reset_flag, pct_done):
    global latitude, longitude, altitude
    global last_status_time
    if reset_flag:
        last_status_time = time.time()

    current_time = time.time()
    #print  "In  send_status--",f_min_current,  f_max_current,
overlap_current
    a = str(f_min_current/1e6)
    a = a + "MHz-"
    a = a + str(f_max_current/1e6)
    a = a + "MHz"
    a = '"' + a + '"'

    statvfs = os.statvfs('/')
    b                                                         =
str(np.round(statvfs.f_bfree*statvfs.f_frsize/2.0**30,2))
    c                                                         =
str(np.round(statvfs.f_blocks*statvfs.f_frsize/2.0**30,2))
    d = str(np.round(pct_done*100,1))+'%'
    d = '"' + d + '"'
    #ipad='"131.68"'
    ipad='"' + IP_addr + '"'
    #print   "In   send_status",a,",   ",b,",   ",c,",   ",d,",
",mac_address,", ",ipad
    #print latitude, longitude, altitude

    if current_time - 10 > last_status_time:
        out = '{"message": "STATUS","payload":{'
        out=out+'"job_id" : ' + a + ',"%_complete":' + d
        out=out+',"disk_used": ' + c
        out=out+',"disk_free": ' + b
        out=out+',"mac_address": ' + '"' + mac_address + '"'
        out=out+',"IP_addr": ' + ipad
        out=out+',"latitude": ' + str(latitude)
        out=out+',"longitude": ' + str(longitude)
        out=out+',"altitude": ' + str(altitude)+' }}'
        #print "Status out:", out


client.publish('radiohound/clients/status/'+mac_address,payloa
d=out)
        last_status_time = current_time
        #print statvfs('/').f_bfree*statvfs('/').f_frsize
        #print statvfs('/').f_blocks*statvfs('/').f_frsize
        #print
statvfs('/').f_bfree*statvfs('/').f_frsize/2.0**30



################################################################
##################
# Set_location
################################################################
##################

def set_location(payload_value):
    global latitude, longitude, altitude

    if type(payload_value) == dict:
        if payload_value.has_key("latitude"):
            latitude = payload_value.get("latitude")
        if payload_value.has_key("longitude"):
            longitude = payload_value.get("longitude")
        if payload_value.has_key("altitude"):
            altitude = payload_value.get("altitude")


################################################################
##################
# Set_time
################################################################
##################

def set_time(payload_value):
    print "set_time: payload_value is:",payload_value
    if type(payload_value) == dict:
        if payload_value.has_key("time"):
            time_string = payload_value.get("time")
            time_string = '"' + time_string + '"'
            print "time_string is:",time_string
            os.system(   'sshpass   -p   "aaaaaaaaa"   sudo   -p
"password:" date -s ' + time_string)



################################################################
##################
# Acquire_GPS
```

```
##############################################################
##################

def acquire_GPS():
    global latitude, longitude, altitude
    pass

    locationFile                                              =
'/home/pi/radiohound/client_gr/tempLocation.txt'
    key_timestamp = 'timestamp'
    key_longitude = 'longitude'
    key_latitude = 'latitude'
    key_altitude = 'altitude'

    def record_location_to_file(lat,lon,alt,timestamp):
       # convert information to json
       newinfo = {}
       newinfo[key_latitude] = lat
       newinfo[key_longitude] = lon
       newinfo[key_altitude] = alt
       newinfo[key_timestamp] = timestamp
       with open(locationFile, 'w') as fp:
           fp.write(json.dumps(newinfo))
       print("     Saved new location to file...")

    #=================             START            EXECUTING
====================================

    # setup alias IP so that we can talk to the Persistent
Systems management screen
    #os.system("sudo  ifconfig  enxd8eb97b69c7e:1  10.3.1.105
netmask 255.255.255.0 up")
    a = 'sshpass -p "aaaaaaaaa" sudo -p "password:" '
    #os.system(a+"ifconfig enxd8eb97b69c7e:1 10.3.1.105 netmask
255.255.255.0 up")
    os.system(a+"ip addr add 10.3.1.131/24 dev enxb827eb09993f")

    # check if the JSON file already exists and delete it
    if os.path.isfile(locationFile):
       print locationFile + " exists, deleting it"
       os.remove(locationFile)

    # url to the Persistent Systems management screen associated
with the GPS
    url                                                       =
"https://10.3.1.254/management.cgi?command=gps_status.json&pas
sword-input=aaaaaaaa"
    #print url


    response = requests.get(url, verify=False, stream=True)
    for line in response.iter_lines():
        #
        # Screen scrape the response line by line until we find
the location data
        #
        # expected format:"Latitude:  ###.###### deg"
        #fiprint line
        if "Latitude:" in line:
            if "unknown" in line: # unknown means we do not yet
have GPS data so set a default
                #line = "Latitude:  41.703059 deg"
                line = "Latitude:  1.111 deg"
            latStr = line[10:len(line)-3]
        # expected format:"Longitude: ###.###### deg"
        if "Longitude:" in line:
            if "unknown" in line:
                #line = "Longitude: -86.238987 deg"
                line = "Longitude: 2.222 deg"
            lonStr = line[11:len(line)-3]
        # expected format:"Altitude:  208 m (682 ft)"
        if "Altitude:" in line:
            if "unknown" in line:
                #line = "Altitude:  208 m (682 ft)"
                line = "Altitude:  3.333 m (0 ft)"
            altStr = line[line.find(":")+1:line.find("m")]
            break

    print  latStr.strip()  +  ",  "  +  lonStr.strip()  +  ",  "  +
altStr.strip()
    latitude = float(latStr.strip())
    longitude = float(lonStr.strip())
    altitude = float(altStr.strip())
    # Update the JSON file with the new data

#record_location_to_file(latStr.strip(),lonStr.strip(),altStr.
strip(),time.time())
    #time.sleep(10)




##############################################################
##################
# Change_config
##############################################################
##################

def change_config(payload_value):
    #global latitude, longitude, altitude

    if type(payload_value) == dict:
        if payload_value.has_key("check_in_server"):
            check_in_server                                  =
payload_value.get("check_in_server")
        if payload_value.has_key("master_controller"):
            master_controller                               =
payload_value.get("master_controller")
        if payload_value.has_key("hostname"):
            hostname = payload_value.get("hostname")
        if payload_value.has_key("short_name"):
            short_name = payload_value.get("short_name")
        if payload_value.has_key("group_name"):
            group_name = payload_value.get("group_name")
        if payload_value.has_key("rh_software_version"):
            rh_software_version                             =
payload_value.get("rh_software_version")
        if payload_value.has_key("system_version"):
            system_version                                  =
payload_value.get("system_version")
        if payload_value.has_key("hardware_version"):
            hardware_version                                =
payload_value.get("hardware_version")
        if payload_value.has_key("rh_hardware_version"):
            rh_hardware_version                             =
payload_value.get("rh_hardware_version")
        if payload_value.has_key("IP_addr"):
            IP_addr = payload_value.get("IP_addr")


##############################################################
##################
# Send_config
##############################################################
##################

def send_config():
    out = '{"message": "CONFIGURATION","payload":{'
    out=out+'"IP_addr" : "' + IP_addr +'"'
    out=out+',"short_name" : "' + short_name +'"'
    out=out+',"system_version" : "' + system_version +'"'
    out=out+',"hostname" : "' + hostname +'"'
    out=out+',"master_controller" : "' + master_controller +'"'
    out=out+',"hardware_version" : "' + hardware_version +'"'
    out=out+',"group_name" : "' + group_name +'"'
    out=out+',"rh_hardware_version" : "' + rh_hardware_version
+'"'
    out=out+',"rh_software_version" : "' + rh_software_version
+'"'
    out=out+',"check_in_server" : "' + check_in_server +'"'
    out=out+'}}'

client.publish('radiohound/clients/status/'+mac_address,payloa
d=out)



##############################################################
##################
# Scan
```

```
################################################################
##################

def scan(payload_value):
    global tb, send_data_flag
    global f_min_current, f_max_current, overlap_current
    global nsamples_current
    global latitude, longitude, altitude, window_welch, nfft

    change_settings = False
    # nfft = tb.get_vector_size()
    samp_rate_current=tb.get_samp_rate()
    fc_current = tb.get_freq0()
    send_status(True,0.0)

    if type(payload_value) == dict:
        if payload_value.has_key("gain"):
            a=payload_value.get("gain")
            tb.set_gain0(a)
            change_settings = True

        if payload_value.has_key("fs"):
            samp_rate_new = payload_value.get("fs")
            if samp_rate_new!=samp_rate_current:
                tb.set_samp_rate(samp_rate_new)
                change_settings = True
                samp_rate_current = samp_rate_new

        if payload_value.has_key("fmin"):
            f_min_new = payload_value.get("fmin")
            fc_new = f_min_new + samp_rate_current/2
            if fc_new!=fc_current:
                tb.set_freq0(fc_new)
                change_settings = True
                fc_current = tb.get_freq0()
            f_min_current = f_min_new

        if payload_value.has_key("fmax"):
            f_max_new = payload_value.get("fmax")
            f_max_current = f_max_new

        if payload_value.has_key("overlap"):
            overlap_new = payload_value.get("overlap")
            overlap_current = overlap_new

        if payload_value.has_key("nsamples"):
            nsamples_current = payload_value.get("nsamples")

        if payload_value.has_key("nfft"):
            nfft = payload_value.get("nfft")

        if payload_value.has_key("window"):
            windowstr = payload_value.get("window")
            if "boxcar" in windowstr:
                window_welch="boxcar"
                window_type="scipy."+window_welch
            elif "blackmanharris" in windowstr:
                window_welch="blackmanharris"
                window_type="scipy."+window_welch
            elif "blackman" in windowstr:
                window_welch="blackman"
                window_type="scipy."+window_welch
            elif "triang" in windowstr:
                window_welch="triang"
                window_type="scipy."+window_welch
            elif "barthann" in windowstr:
                window_welch="barthann"
                window_type="scipy."+window_welch
            elif "hamming" in windowstr:
                window_welch="hamming"
                window_type="scipy."+window_welch
            elif "hann" in windowstr:
                window_welch="hann"
                window_type="scipy."+window_welch
            elif "bartlett" in windowstr:
                window_welch="bartlett"
                window_type="scipy."+window_welch
            elif "flattop" in windowstr:
                window_welch="flattop"
                window_type="scipy."+window_welch
            elif "parzen" in windowstr:
                window_welch="parzen"
                window_type="scipy."+window_welch
            elif "bohman" in windowstr:
                window_welch="bohman"
                window_type="scipy."+window_welch
            elif "nuttall" in windowstr:
                window_welch="nuttall"
                window_type="scipy."+window_welch

    # figure out how many scan windows there are
    fc0 = f_min_current + samp_rate_current/2
    fc_last = f_max_current - samp_rate_current/2
    fc_interval = samp_rate_current * (1 - overlap_current)
    # num_iterations is the number of scans to do for to get the
full freq range
    # N_windows  is  the  number  of  averages  to  do  for  each
iteration
    num_iterations = (fc_last - fc0) / fc_interval + 1
    N_windows                                                  =
int(np.floor(nsamples_current/tb.get_vector_size()))
    print
"fc0:",fc0/1e6,"fc_last",fc_last/1e6,"samp_rate_current:",samp
_rate_current/1e6
    print                  "num_iterations:",num_iterations,",
N_windows:",N_windows

    data_sum = 0;
    # capture data and throw away because might be old settings
    #figure out how many to throw away
    print "change_settings is:",change_settings
    if change_settings:
        for j in range(0,10):
            vec_list                                          =
parse_sdr_data(tb.msgq_out.delete_head().\
            to_string(),tb.get_vector_size())  #  this  indeed
blocks

    # loop over scan windows
    # for i in range(0,N_windows):
    #     # configure SDR - Fc should alread be set above
    #     #todo

    #     # capture data with good settings
    #     vec_list = parse_sdr_data(tb.msgq_out.delete_head().\
    #          to_string(),tb.get_vector_size()) # this indeed
blocks
    #     data = vec_list[0]
    #     data_sum = np.add(data_sum,data)

    vec_list=[]
    data=[]
    while (len(data)<nsamples_current):
        iq_block=tb.msgq_out.delete_head().to_string()  #  this
indeed blocks
        for i in range(0,len(iq_block),8):
            re = (struct.unpack_from('f',iq_block[i:i+4]))
            im = (struct.unpack_from('f',iq_block[i+4:i+8]))
            if (len(data)<nsamples_current):
                data.append(re[0] + 1.0j*im[0])

    # Do fft:
    f,Pxx_den                                                  =
signal.welch(data,fs=samp_rate_current,window=window_welch,npe
rseg=nfft)
    Pxx_den=np.fft.fftshift(Pxx_den)
    f=np.fft.fftshift(f)

    # format data and send
    formatted_data = format_data_for_tx(Pxx_den,data,tb)
    send_data_flag = 1
    send_data(formatted_data)
    send_data_flag = 0
    print "fc:",fc0/1e6, data[254:259]




    # If >1 iterations in the scan, do them:
    for k in range(2, int(np.ceil(num_iterations+1))):
        # loop through the iterations to do the full scan:
```

```
        send_status(False,float(k/num_iterations))
        # configure SDR - change Fc
        fc_new   =   tb.get_freq0()   +   (1-overlap_current)   *
samp_rate_current
        tb.set_freq0(fc_new)

        # capture data and throw away because might be old
settings
        #figure out how many to throw away
        for j in range(0,10):
            vec_list                                            =
parse_sdr_data(tb.msgq_out.delete_head().\
            to_string(),tb.get_vector_size())  #  this  indeed
blocks

        # Collect good data:
        data=[]
        while (len(data)<nsamples_current):
            iq_block=tb.msgq_out.delete_head().to_string()    #
this indeed blocks
            for i in range(0,len(iq_block),8):
                re = (struct.unpack_from('f',iq_block[i:i+4]))
                im                                              =
(struct.unpack_from('f',iq_block[i+4:i+8]))
                if (len(data)<nsamples_current):
                    data.append(re[0] + 1.0j*im[0])

        # Do fft:
        f,Pxx_den                                               =
signal.welch(data,fs=samp_rate_current,window=window_welch,npe
rseg=nfft)
        Pxx_den=np.fft.fftshift(Pxx_den)
        f=np.fft.fftshift(f)


        # format data and send
        formatted_data = format_data_for_tx(Pxx_den,data,tb)
        send_data_flag = 1
        send_data(formatted_data)
        send_data_flag = 0
        print "fc:",fc0/1e6, data[254:259]




##############################################################
##################
# Parse_sdr_data
##############################################################
##################

def parse_sdr_data(data_str,vector_size):
    block_size = 4 * vector_size
    N_blocks = len(data_str)/block_size
    vec_list = []
    for i in range(0,N_blocks):
        floats = []
        for j in range(0,block_size,4):

floats.append(struct.unpack_from('f',data_str[(i*block_size)+j
:(i*block_size)+j+4])[0])
        vec = np.array(floats)
        vec_list.append(vec)
    return vec_list


###############################################################
##################
# Format_data_for_tx
###############################################################
##################

def format_data_for_tx(Pxx_den,data,tb):

    # metadata
    #software_version = "0.1";
    timestamp                                                 =
datetime.datetime.utcfromtimestamp(time.time()).strftime('%Y-
%m-%dT%H:%M:%S.%f-%z') + "0000"
    gain = tb.get_gain0()
    #short_name = "ARL_xxx"
    uncertainty = 35.2;
    fcen = tb.get_freq0()
    sample_rate = tb.get_samp_rate()
    fhi = fcen + sample_rate/2.0;
    flo = fcen - sample_rate/2.0;
    iq_data=base64.b64encode(str(data))

    # data
    data_str = ""
    for x in xrange(0,len(Pxx_den)):
        data_str = data_str + str(Pxx_den[x]) + ","

    test_data2 = {
        "software_version": rh_software_version,
        "timestamp": timestamp,
        "gain": tb.get_gain0(),
        "data": data_str,
        "short_name": short_name,
        "uncertainty": uncertainty,
        "longitude": longitude,
        "sample_rate": tb.get_samp_rate(),
        "mac_address": mac_address, # e.g. b827eb228478
        "latitude": latitude,
        "center_frequency": fcen,
        "metadata": {"nfft": nfft,
                     "fmax": fhi,
                     "scan_type": "linear",
                     "detrend": "True",
                     "report_type": "periodogram",
                     "nsamples": nsamples_current,
                     "antenna": "default",
                     "actualGain": tb.get_gain0(),
                     "overlap": overlap_current,
                     #"window": "numpy.hanning",
                     "window": window_type,
                     "fmin": flo,
                     "zero_pad_to": 0,
                     "noverlap":
int(overlap_current*tb.get_vector_size()),
                     "iq_data": iq_data
                     }
    }
    #print test_data2["metadata"]
    return json.dumps(test_data2)


##############################################################
##################
# Send_data
##############################################################
##################

def send_data(formatted_data):
    global last_trace_time
    global send_data_flag

    if send_data_flag == 2:
        current_time = time.time()
        if current_time - 1 > last_trace_time:

client.publish('radiohound/clients/data/'+mac_address,\
                payload=formatted_data)
            last_trace_time = time.time()
    elif send_data_flag == 1:
        pass

client.publish('radiohound/clients/data/'+mac_address,\
            payload=formatted_data)
        send_data_flag = 0
    else:
        pass

    POST_LOCATION = 'http://radiohound1.crc.nd.edu/storedata/'
    TIMEOUT = 5 # We have found that the timeout needs to be
greater than 1 for consistent ingestion of data
    # req = urllib2.Request(POST_LOCATION)
    # req.add_header('Content-Type', 'application/json')
    #                        response                        =
urllib2.urlopen(req,json.dumps(formatted_data))
    # print response.getcode()
```

```
###############################################################
##################
# Main
###############################################################
##################

def main(top_block_cls=top_block, options=None):
    global tb, msg_queue
    global   f_min_current,   f_max_current,   overlap_current,
nsamples_current
    ##################################################
    # MQTT Init
    ##################################################
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect(broker_ip)
    client.loop_start()
    ###################################################


    if StrictVersion(Qt.qVersion()) >= StrictVersion("4.5.0"):
        style    =    gr.prefs().get_string('qtgui',    'style',
'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)


#client.publish('radiohound/clients/command_to_all',payload=st
r(i))

#client.publish('radiohound/clients/command/'+mac_address,payl
oad=str(i))

    # heartbeat


#client.publish('radiohound/clients/announce/'+mac_address,pay
load=str(i))

#client.publish('radiohound/clients/status/'+mac_address,paylo
ad=str(i))


    tb = top_block_cls()
    tb.start()
    tb.vector_size
    #tb.show()
    # time2 = time.time()
    # print "Time:",time2-time1
    # time1 = time.time()
    ave_length =16
    vector_size = tb.get_vector_size()
    block_size = 4 * vector_size
    nsamples_default = 16384
    num_aves = nsamples_default/vector_size
    samp_rate = tb.get_samp_rate()

    f_min_current = tb.get_freq0() - samp_rate/2;
    f_max_current = tb.get_freq0() + samp_rate/2;
    overlap_current = 0.5;
    nsamples_current = 16384;

    #print vector_size, block_size, num_aves

    # main loop
    while True:
        if len(msg_queue) > 0:
            first_msg = msg_queue.pop(0)
            parseJson(first_msg)

        send_heartbeat()
        vec_list = parse_sdr_data(tb.msgq_out.delete_head().\
            to_string(),tb.get_vector_size())  #   this   indeed
blocks

        data = vec_list[0]
        f,Pxx_den                                               =
signal.welch(data,samp_rate_current,nperseg=nfft)

        formatted_data = format_data_for_tx(Pxx_den,data,tb)

        send_data(formatted_data)

        #break
        #print data#[255:258]



if __name__ == '__main__':
    main()
```

## Appendix D. Window Types

There are several window types that can be selected. These windows applied to the time-domain data vectors before performing frequency transforms on them. The list of window types is as follows:

- "boxcar"
- "blackmanharris"
- "blackman"
- "triang"
- "barthann"
- "hamming"
- "hann"
- "bartlett"
- "flattop"
- "parzen"
- "bohman"
- "nuttall"

## List of Symbols, Abbreviations, and Acronyms

| | |
|---|---|
| API | application programming interface |
| ARL | US Army Research Laboratory |
| FFT | fast Fourier transform |
| GPS | global positioning system |
| GR | GNU Radio |
| GRC | GNU Radio Companion |
| IP | Internet Protocol |
| IQ | in-phase and quadrature |
| JSON | JavaScript Object Notation |
| MAC | media access control |
| MANET | mobile ad-hoc network |
| MQTT | Message Queue Telemetry Transport |
| ND | University of Notre Dame |
| RF | radio frequency |
| SBC | single-board computer |
| SDR | software-defined radio |
| USB | universal serial bus |
| USRP | Universal Software Radio Peripheral |

|  |  |
|---|---|
| 1<br>(PDF) | DEFENSE TECHNICAL<br>INFORMATION CTR<br>DTIC OCA |
| 2<br>(PDF) | DIR ARL<br>RDRL DCM<br>IMAL HRA MAIL & RECORDS<br>MGMT<br>RDRL IRB<br>TECH LIB |
| 1<br>(PDF) | GOVT PRINTG OFC<br>A MALHOTRA |
| 2<br>(PDF) | ARL<br>RDRL SER W<br>N TESNY<br>C DIETLEIN |
| 1<br>(PDF) | ALION SCIENCE AND<br>TECHNOLOGY<br>D GALANOS |