AUTONOMOUS RF DATA COLLECTION WITH SOFTWARE DEFINED RADIO

An Undergraduate Research Scholars Thesis

by

RYAN CAMPBELL

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Gregory Huff

May 2018

Major: Electrical Engineering

TABLE OF CONTENTS

ABSTRACT

Autonomous RF Data Collection with Software Defined Radio

Ryan Campbell
Department of Electrical Engineering
Texas A&M University

Research Advisor: Dr. Gregory Huff
Department of Electrical Engineering
Texas A&M University

Characterizing the performance of an antenna or radio involves collecting a large amount of data through tedious processes. In order to combat this tedium, and to improve the data collected, this work explored the use of two burgeoning technologies: software defined radio (SDR) and autonomous drones. SDR provides the ability to perform repeatable measurements and to easily scan wide frequency bands, while autonomous drones add additional repeatablity to measurements as well as remote operation. To combine these two technologies into an experimental framework, an SDR was mounted to a rover and connected to an onboard Raspberry Pi single board computer. The SDR acted as a receiver, and the Raspberry Pi performed signal processing to determine the strength of the signal received. The Raspberry Pi then transmitted this data to the computer of the user running the experiment where it could be used to generate a heat map of signal strength or stored for later analysis. The experimental framework developed by this research greatly enhances the ability to perform experiments on reconfigurable antennas and to investigate multi-band and wide-band wireless networks.

NOMENCLATURE

RF	Radio Frequency
SDR	Software Defined Radio
TAMU	Texas A&M University

1. INTRODUCTION

Studying mobile RF data collection allows researchers and developers to examine how mobile users receive and transmit RF data and can help to identify parts of the process that can be improved. There are certain situations in which it is necessary to gather this data manually. This is unfortunate because the gathering of a significant amount of data can be very time consuming. For example, Texas A&M's Huff Research Group currently runs an experiment to generate a heat map of received signal strength (RSS) by mounting a set of radios on a rover and commanding it semi-autonomously to a number of locations. This process requires one or two rover operators, and someone handling the RF measurements. Automation of the rovers and the integration of software defined radios (SDRs) and reconfigurable antennas could greatly improve this process.

SDR is the implementation of radio components, like mixers, modulators, or filters in software rather than hardware. The technology was developed in the 1990s, but it has become poised to be a much more disruptive technology with the continuous improvement of supporting technologies [1]. As a test platform, SDR grants the ability to monitor a number of frequencies using a software front-end which can be automated. For instance, it would be possible to develop a means of writing tests quickly with a scripting language like Python to easily ensure repeatability of measurements. Using SDRs in the automation of data collection could also provide insight into the development of antenna arrays that are reconfigurable via autonomous movement of their host platform.

The field of robotics has also been booming in recent years with advances in autonomous vehicles and general purpose robots like Rethink Robotics' "Baxter". There has been research in the control of robotic swarms [2] which could be leveraged to manipulate reconfigurable antenna arrays. It is possible that a swarm of antennas deployed

on rovers or drones could be commanded to achieve a specific radiation pattern based on their position and SDR configuration.

Finally, data visualization brings data to life and allows much greater insight to what is gathered. Leveraging open-source libraries, like Python's matplotlib, for data visualization in a cross-platform application provides a tool that lowers the cost of studying spectral band coexistence. Understanding these issues could serve to improve wireless connectivity in autonomous vehicles and systems.

2. BACKGROUND

2.1 Software Defined Radio

As stated in the introduction, SDR is the implementation of radio components, like mixers, modulators, or filters in software rather than hardware. This allows signals to be passed through software algorthims for data processing or analysis as opposed to being forced to rely on more expensive, more complicated, and less flexible hardware solutions. This is especially useful as computer continue to get faster and can handle complex algorithms that take full advantage of the technology.

2.2 RF Measurement

The characterization of radio equipment or the frequency spectrum often relies on expensive equipment. More common measurements include voltage standing wave ratio (VSWR), S-parameters, antenna patterns, and the focus of this project: received signal strength.

2.2.1 Received Signal Strength (RSS)

Received signal strength is a measure of the power present in a signal at some frequency. This value is typically represented as dBm or decibels relative to a miliwatt. Given some signal strength, X in watts, the signal in dBm is found as:

$$P(X)[dBm] = 10\log_{10}\left(\frac{P(X)[W]}{0.001}\right) \tag{2.1}$$

So a signal with power of 1 nanowatt would be

$$P(X)[dBm] = 10\log_{10}\left(\frac{1 \times 10^{-9}}{0.001}\right) \tag{2.2}$$

$$= -60\ dBm \tag{2.3}$$

2.3 Spectrum Analysis

All electromagnetic frequencies are frequently represented on a line and are referred to as the "electromagnetic spectrum." This is often ordered by wavelength such that smaller wavelengths are on the left and larger on the right. Since wavelength is inversly related to wavelength, this means frequencies on the left are higher than those on the right of the frequency plot, shown in Figure 2.1.

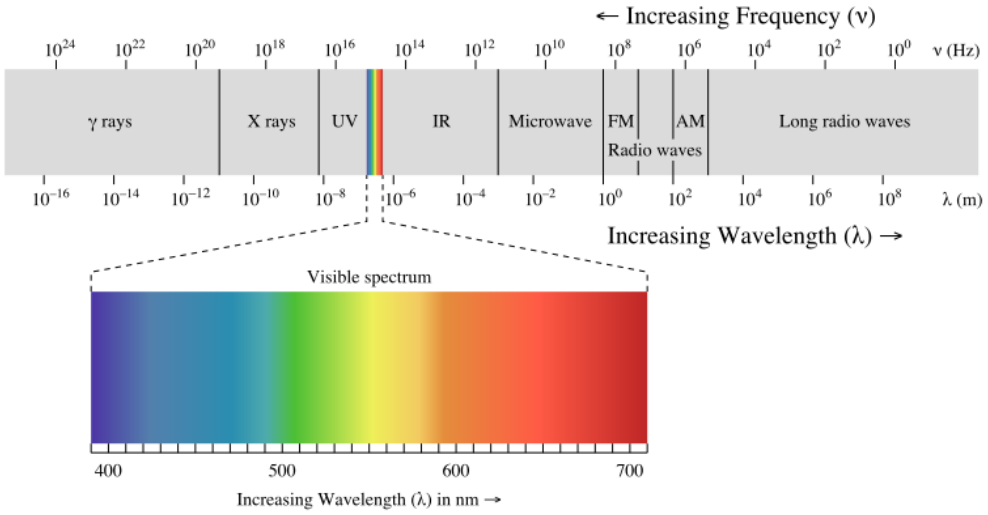

Figure 2.1: Electromagnetic Spectrum [3]

2.3.1 Fourier Transform

The Fourier Transform is a mathematical tool that allows a time-domain signal to be represented in the frequency domain. Given some time-domain signal, $f(x)$, the frequency-domain signal, $F(X)$ can be found by:

$$F(X) = \int_{-\infty}^{\infty} f(x) e^{-2\pi j x} dx \tag{2.4}$$

Note however, this is for a continuous signal. SDRs' data is represented is represented as discrete values which don't work with Equation 2.4. Instead, the Discrete Fourier Transform (DFT) must be used which calculates the Fourier Transform of a discrete set of data.

$$F_n = \sum_{k=0}^{N-1} f_k e^{\frac{-2\pi j n k}{N}} \tag{2.5}$$

This equation is computationally expensive. As a response to this, the Fast Fourier Transform (FFT) was developed. This is an algorithm which computes the DFT much faster than using the algorithm above. Note the FFT algorithm produces exactly the same output as computing the DFT directly. The graphical output of this operation is shown in Figure 2.2. This plot represents the power of the signal at each frequency. Note the three peaks corresponding to the three sine waves which make up the signal on the left.

2.3.2 Waterfall Plot

A waterfall plot shows the result of an FFT over time. This provides a sense of how waves are behaving in the spectrum. Figure 2.3 shows an example of a FFT(above) and a waterfall plot(below) of the frequency spectrum between 88 and 108 MHz. This is where FM radio is located and one can see the brighter solid lines at certain frequencies. These frequencies have high energy (compared to the background) because local radio stations are transmitting at those frequencies.

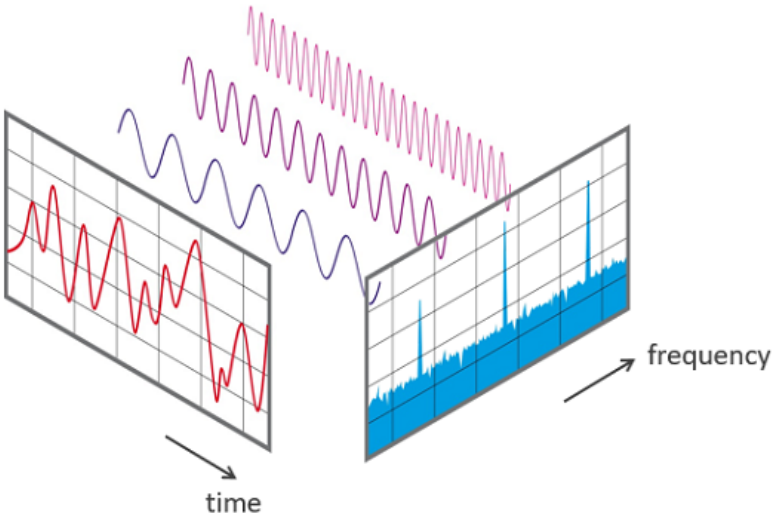

Figure 2.2: FFT of 3 Sine Waves [4]

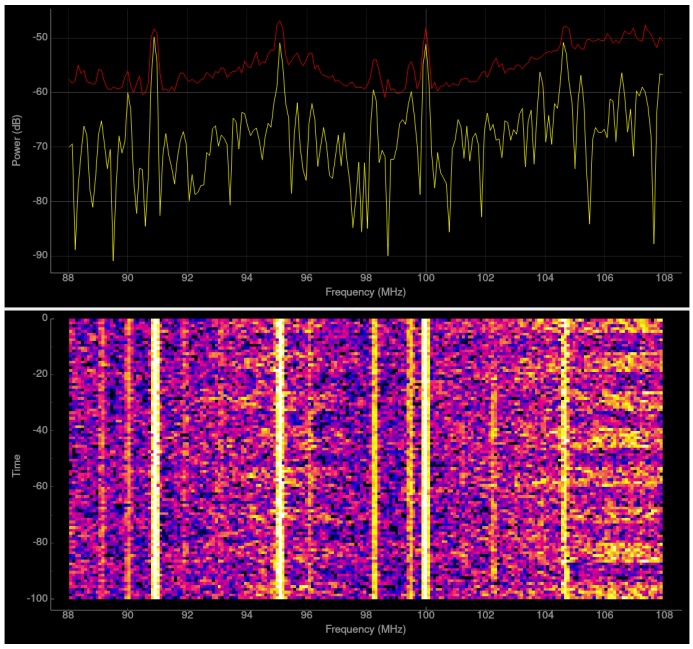

Figure 2.3: FFT and Waterfall Plot 88-108 MHz

3. DESIGN

3.1 System

The primary factor influencing the system design was the hardware available on hand: the rovers, the Pixhawk autopilot system, and the HackRF SDR. The first design considered, illustrated by Figure 3.1, would have relied on the Pixhawk to report releveant GPS information. This data would've then been compiled in to a data packet for transmission to a remote machine running software to catalog the information. Additionally, it was thought that using an Intel NUC (Next Unit of Computing) would ease development, eliminating the need for cross compilation. After some preliminary testing with the Pixhawk, it was determined to be too unreliable and difficult to work with to be feasible for this application. The power requirements for the NUC were also prohibitive. For these reasons, the following design changes were enacted:

- Migrate from an Intel NUC to a Raspberry Pi on board the rover to ease power requirements.

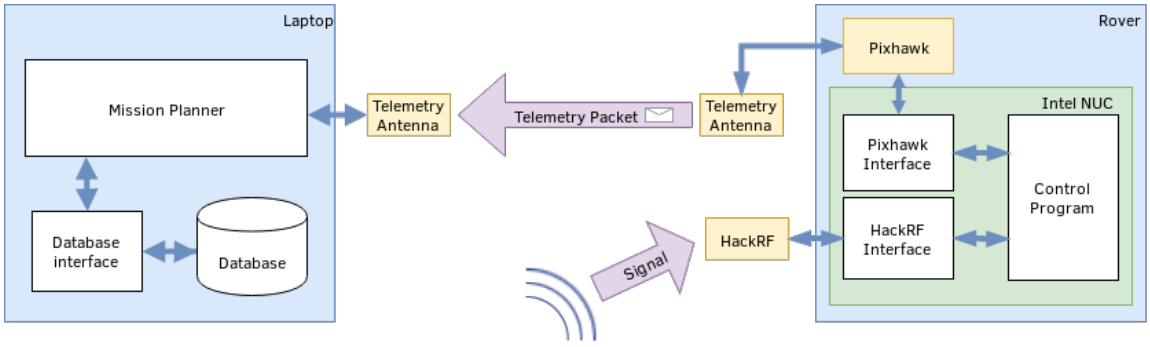

Figure 3.1: Initial System Design

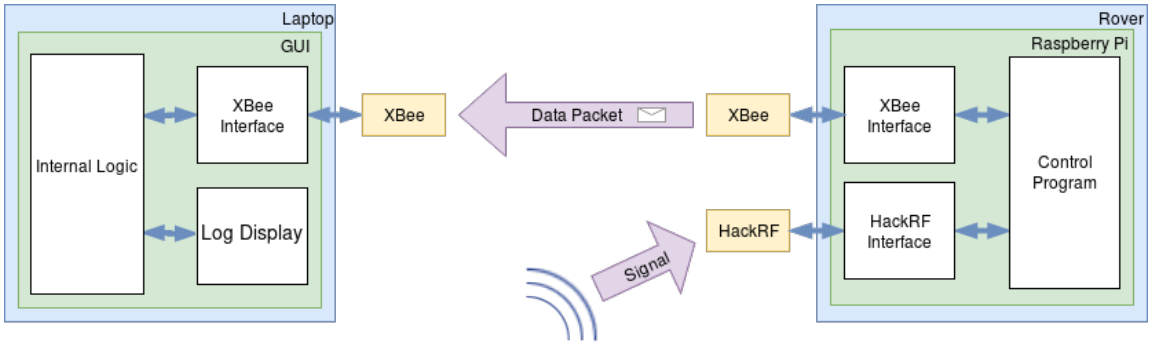

Figure 3.2: Final System Design

- Make the data collection system external to the autopilot to ease development and improve reliability
- Introduce XBee radios for communication (instead of relying on the Pixhawk's telemetry antennas) to provide greater flexibility.

These changes are illustrated by Figure 3.2.

3.2 Host Software

The host software is that which would be run on the machine conducting the experiment. The requirements of the Host software were as follows:

- Provide an intuitive cross-compatible graphical interface.
- Allow for the connection to and management of a SQL database for storing data in a systematic way.
- Allow for the visualization of collected data.
- Interface with the XBee attached to the host machine to send commands to the RaspberryPi on the rover.

- Have an awareness of the current state of the HackRF as reported by the rover.

3.2.1 Cross-platform Compatible Graphical Interface

A cross-platform program is one that is able to be compiled and run on multiple operating systems, e.g. Windows, Linux, MacOS, etc. Developing a cross-platform graphical interface introduces a rather significant challenge, though, thankfully, there exist a number of APIs that make this far more approachable. Some major cross-platform GUI libraries are: Qt, GTK+, and wxWidgets.

3.3 Remote Client Software

The remote client software is the software that would be run on the rover while it collects data to transmit back to the host. The requirements were as follows:

- Concatenate and packetize RSSI and GPS information.
- Send these data packets over the XBee.
- Receive, interpret, and act on commands sent from the Host.
- Control the HackRF and report its status via the XBee.

4. IMPLEMENTATION

Recall the design requirements as outlined by the previous chapter:

- Use hardware on hand:
  - Rovers
  - Pixhawk
  - HackRF SDR
  - Intel NUC or RaspberryPi
- Take limited availability of power on board rovers into consideration.
- Have the ability to change frequency "on-the-fly."
- Have the ability to visualize data in a meaningful way.

The hardware chosen is shown in Figure 4.1. The software was implemented using a host-client architecture i.e. a host (laptop in this case) sends commands to the client (the rover in this case) and the client keeps the host apprised of its status. This was accomplished by creating a GUI using the Qt library for the host program, and a command line program run on the RaspberryPi. Communication was handled through the use of XBee radios. The host sends commands to through an XBee to the client. The client's XBee receives this packet extracts the command and passes that along to the client program for interpretation. This architecture allowed for the fulfillment of all design requirements as will be detailed in the following sections.

4.1 Host Program

The host program was built using C++ and the Qt GUI library. C++ was chosen as the author is more capable in it than C. It also provides the speed required for this application. Qt was chosen for its legacy, its ease of development, its extensive library of helper classes

Figure 4.1: Rover Hardware

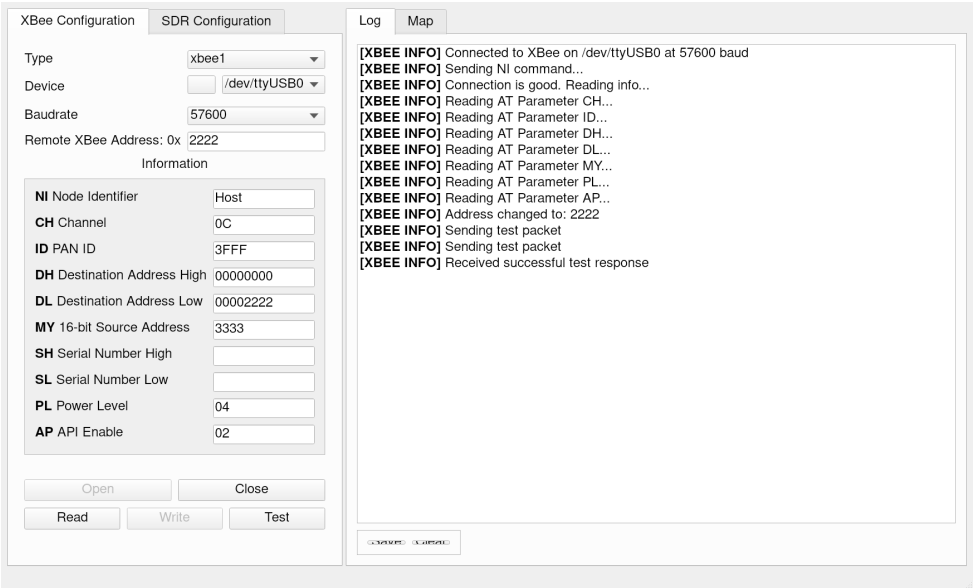

Figure 4.2: Graphical User Interface

which ease interaction with external components like databases and serial ports, and for its cross-platform capabilities. The final product is shown in Figure 4.2. The intention is for a user to establish that the XBees are functioning properly on both the host and the client before moving on to sending commands. This is done by reading the XBee attached to the host to ensure its proper functionality, then specifying the address of the remote XBee and sending a test command. When the client receives a test packet, it will respond with a test packet of its own, thus informing the host that communication has been established. With the connection established, the host commands the client to begin sweeping over some frequency range (as described in the later HackRF section) which returns FFT frames to the host. This data can then be plotted simply as raw spectral data, or can have some minimal post-processing done to generate a heat map.

4.2 Client Program

The client program was built using C++ for the same reasons described in the previous section. The development process for this program was however considerably more involved as it required cross-compilation. Cross-compilation is the act of compiling a program on one computer for another computer of a different processor architecture. The RaspberryPi's processor is an ARM architecture whereas the code was developed on a computer with an x86_64 processor. CMake was used extensively to ease this process.

4.3 RaspberryPi

The RaspberryPi is a single board computer built with an ARM processor. It is fairly cheap ($30) yet powerful enough to run Linux. The RaspberryPi was particularly compelling for this experiment for its size and minimal power requirements. Since it is powered from a standard 5V USB, it is able to receive power from a common power bank typically used for charging phones. The process of cross-compiling for the RaspberryPi is detailed in Appendix A.

4.4 HackRF Interface

The creator of the HackRF, Michael Ossmann, created a powerful though somewhat difficult to use API for controlling the HackRF [5]. The API provides functions for tuning the radio to a specific frequency, changing the sample rate, turning the antenna on or off, etc. In addition to these API functions, Ossmann has published a number of command line utilities for exploring the HackRF. Of particular note is the "hackrf_sweep" program. This program takes as inputs (among other things) a start and stop frequency and a number of samples per frequency then sweeps across that frequency range very quickly to provide an FFT of that bandwidth. This proved particularly useful in this experiment as this data is exactly the data in question: received power at specific frequencies. These frequency sweeps generate one frame of FFT data that is concatenated with a GPS location before being sent back to the host. The length of these frames vary depending on the number of samples and the FFT bin width chosen.

4.5 GPS Interface

The GPS is the simplest part of this architecure: a GPS shield for an Arduino. An Arduino is a simple microcontroller with a large number of libraries and pre-built "shields" or add on modules for additional functionality. With this platform, getting GPS information is acquired by using the Arduino to query the GPS, which then sends the coordinates as a string over it's serial port. The client program then reads the serial port whenever it needs to concatenate GPS information with FFT information from the HackRF.

4.6 XBees

XBees are a family amateur radios produced by Digi International Inc. These radios are frequently used in rapid prototyping, specifically for their ease of development. Different models offer different capabilities such as different operating frequencies, different

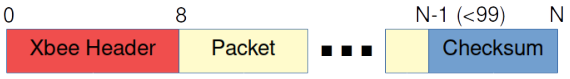

Figure 4.3: XBee Packet Structure [7]

Command Packets
Info
RSSData
StartHackRF
HackRFData
Test
Null

Figure 4.4: Packet Structure for Rover Communication

transmit power levels, and different communication protocols. There also exist a number of third party libraries for interfacing with XBees. For this project, the libxbee3 library from Attie Grande was chosen as it is well maintained and documented [6].

The packet structure of the XBees is shown in Figure 4.3. Notice the payload of a packet is limited to 100 bytes. In this 100 bytes sits another packet structure specific to this project that is detailed by Figure 4.4. Creating this additional packet structure simplified the code for receiving and interpreting packets. First the type of packet is determined, then the data is extracted and interpreted appropriately.

4.6.1 Splitting HackRF Data

Because a frame of HackRF data varies depending on the sweep parameters, it is necessary to split a frame into a number of chunks able to be fit into an XBee data packet. An arbitrary number of 75 bytes was chosen as the maximum packet size. HackRF Data is generated as a string determined by the hackrf_sweep program. Before being split and sent, GPS data is added to the beginning of the string. The exact frame structure and an

Table 4.1: HackRF Data Frame

Date	Time	Start Frequency	End Frequency	FFT Bin Width	Number of Samples	dB	dB	...	dB
2018-04-04	00:40:05.746468	2435000000	2440000000	98039.22	204	-85.02	-72.11	...	-74.48

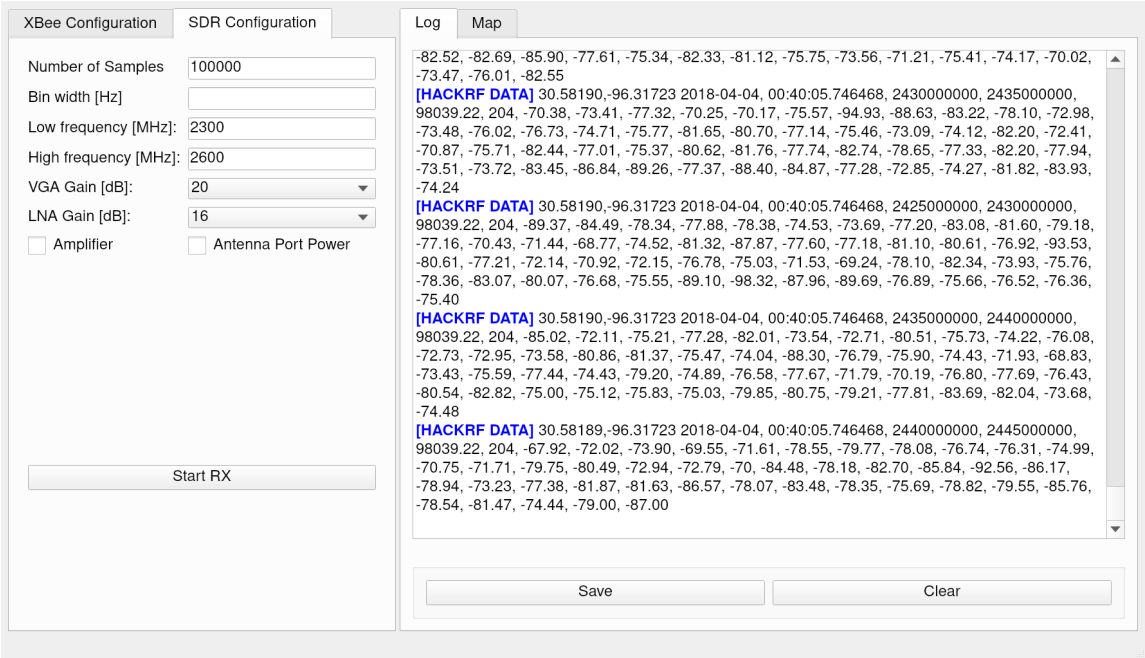

Figure 4.5: Receing HackRF Data Frames

example below it is shown in Table 4.1. The example from Table 4.1 is 475 bytes long. This plus 19 bytes for GPS data is a total of 494 bytes. This is evenly divided by 75 six times with a remainder of 44 bytes which will make up an non-full packet. A data frame packet also keeps track of the total number of packets in the frame as well as which number of the total it comprises. On the receiving end, the GUI recombines the packets individually until the received packet number matches the total number of packets, at which point a signal is fired containing the string which gets picked up by the log for display. This final step is shown in Figure 4.5. This data is similar to the data in Figure 2.3, just not shown graphically.

5. SUMMARY AND CONCLUSIONS

As an experiment, this platform is effective and accomplishes most of the established goals. There are however, a number of issues that should be rectified moving forward including: speed, code complexity, ease of use, and lack of data visualization.

5.1 Speed

XBees do not provide a large data throughput; the Series 1 has a data rate of 1200 bits per second or 150 bytes per second. Recall the example frame from Table 4.1 was 494 bytes. This means transferring a single frame of data takes almost 3.29 seconds. This example from is one of 200 in the whole sweep. This means a single sweep takes over 3 minutes, which is excessive for this application. In the future, radios with a higher data rate should be used.

5.2 Code Complexity

The speed of development precluded using good code practice. This means the code is fairly brittle; updating it to use another SDR or a different set of radios would take more work than if the code had a better architecure. Future work should also focus on rearchitecting the code to make it more modular and extensible.

5.3 Ease of Use

The code architecture and the hackrf_sweep program from the hackrf API make it impossible to stop or change the experiment without completely killing the client program and restarting it. This is a problem because the client program is intended to run on the rover with no display or input devices, which these series of steps requires. Future work should focus on developing this capability which could be achieved through rearchitecting the host and client programs.

5.4 Lack of Data Visualization

One of the goals of this research was to develop a means of visualizing the data collected. The difficulty involved in implementing the host and client programs and integrating the multiple pieces of hardware precluded the achievement of this goal. Future work in this area could utilized the Google Maps or Google Earth APIs which provide heatmap functionality that is fairly easy to interact with.

REFERENCES

[1] W. Tuttlebee, "Software-defined radio: Facets of a developing technology," *IEEE Personal Communications*, vol. 6, no. 2, pp. 38–44, 1999.

[2] S. Kantesaria and N. Olivarez, "Software defined radio," Master's thesis, Worcester Polytechnic Institute, 2011.

[3] P. Ronan, "Em spectrum," 2007.

[4] Phonical, "Fft time frequency view," 2017.

[5] M. Ossmann, "hackrf." `https://github.com/mossmann/hackrf`, 2018.

[6] A. Grande, "libxbee3." `https://github.com/attie/libxbee3`, 2014.

[7] MaxStream, Inc., *XBee/XBee-Pro OEM RF Modules Product Manual*, 1.x.ax ed., 2007.

APPENDIX A: CROSS COMPILATION

Cross Compilation is compiling a program on on machine with some processor architecure (e.g. x86_64) for another machine with a different processor architecure (e.g. ARM). A tool exists for creating cross compilers for a large number of processors called crosstool-ng (https://github.com/crosstool-ng/crosstool-ng). To use it, one sets up their environment according to the instructions then runs a configuration tool to finely tune the cross-compiler being built. After the configuration is complete, the cross-compiler is compiled.

A.1 CMake

CMake, which stands for "cross-platform make" is a tool for generating build files. These build files are then used to compile the desired program. In the case of this project, CMake was instructed to create build files that pointed to the cross-compiler created by crosstool-ng. The script used to configure CMake is shown below.

```
cmake_minimum_required(VERSION 3.5)
project(PiComm)

set (CMAKE_CXX_STANDARD 14)

# Initialize git submodules
if(EXISTS "${CMAKE_CURRENT_SOURCE_DIR}/lib/PiComm_Packets/packet.hpp")
    message("Checking for updates to PiComm_Packets")
    # Ensure PiComm_Packets is up-to-date
    execute_process(COMMAND git submodule update --recursive --remote
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})
else()
    message("PiComm_Packets not found...")
    execute_process(COMMAND git submodule update --init --recursive --remote
        WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR})
endif()

message("${CMAKE_LIBRARY_PATH}")

add_executable(PiComm
  lib/PiComm_Packets/packet.cpp
  lib/TimeoutSerial/TimeoutSerial.cpp
  src/picomm.cpp
  src/xbee.cpp
  src/hackrf.cpp
  src/sample.cpp
```

```
  )

target_include_directories(PiComm PRIVATE
  lib
  include)

target_link_libraries(PiComm
  xbee
  xbeep
  hackrf
  pthread
  boost_system)
```

The CMake command was invoked using the following shell script:

```
#!/bin/bash

if [ -e build ]; then
    rm -r build
fi
mkdir -p build
cd build

# Check where the cmake toolchain file is
if [ -e /home/rdcampbell/Documents/RaspberryPi/CMakeToolchain/Toolchain-RaspberryPi.cmake ];
then
    TOOLCHAIN_LOCATION=\
    /home/rdcampbell/Documents/RaspberryPi/CMakeToolchain/Toolchain-RaspberryPi.cmake
elif [ -e /home/rdcampbell/CMakeToolChain/Toolchain-RaspberryPi.cmake ]; then
    TOOLCHAIN_LOCATION=\
    /home/rdcampbell/CMakeToolChain/Toolchain-RaspberryPi.cmake
else
    echo "Could not find Toolchain-RaspberryPi.cmake. Aborting..."
    set +x
    exit 1
fi

cmake -DCMAKE_TOOLCHAIN_FILE=${TOOLCHAIN_LOCATION} -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ../
OUT=$?

if [ $OUT -eq 0 ]; then
    make
fi
```

APPENDIX B: CODE STRUCTURE

The code for this project is broken up into three different modules:

- PiComm (client program)
- DataLoggerGUI (host program)
- PiComm_Packets (shared by both programs)

B.1 PiComm_Packets

PiComm_Packets is a simple header and implementation file that defines the packet structure. Because this packet structure is shared between the two programs, it is made as a separate module that both reference. This prevents inconsistencies between the two programs. The header file is shown below to illustrate the structure of this module. A generic Packet class is declared that all packet types are derived from. Each packet class then implements its own serialization methods, and accessor functions.

```
#ifndef PICOMM_PACKET_HPP
#define PICOMM_PACKET_HPP

#include <string>
#include <vector>
#include <cstring>

namespace PiCommPacket {

enum class PacketType : std::uint8_t {
    Info,
    RSSData,
    StartHackRF,
    HackRFData,
    Test = 0xee,
    Null = 0xff
};

class Packet
{
  public:
    Packet();
    ~Packet();
    virtual std::vector<unsigned char> serialize() = 0;
```

```
  protected:
    PacketType type_;
};

class StartHackRFPacket : public Packet
{
    public:
        StartHackRFPacket(std::string command);
        StartHackRFPacket(std::vector<unsigned char> raw_packet);
        std::vector<unsigned char> serialize();
        std::string command();
    private:
        std::string command_;
};

class HackRFData : public Packet
{
    public:
        explicit HackRFData(std::string packet_string, uint8_t index, uint8_t total);
        explicit HackRFData(std::vector<unsigned char> raw_packet);
        std::vector<unsigned char> serialize();
        std::string data();
        uint8_t index();
        uint8_t total();
    private:
        // These two parameters indicate this is the Xth packet out of Y total packets
        uint8_t index_; // X
        uint8_t total_;  // Y
        std::string data_;
};

class InfoPacket : public Packet
{
  public:
    InfoPacket(std::string text);
    InfoPacket(std::vector<unsigned char> raw_packet);
    std::vector<unsigned char> serialize();
    std::string text();

  private:
    std::string text_;
};

class RSSDataPacket : public Packet
{
  public:
    RSSDataPacket(float rss_, float lat_, float lon_);
    RSSDataPacket(std::vector<unsigned char> raw_packet);
    std::vector<unsigned char> serialize();

  private:
    float rss_;
    float lat_;
    float lon_;
};

class TestPacket : public Packet
{
  public:
    TestPacket(bool success_);
    TestPacket(std::vector<unsigned char> raw_packet);
    std::vector<unsigned char> serialize();

  private:
```

```
    bool success_;
};
}

#endif
```

B.2 PiComm

PiComm is the name given to the program which runs on the RaspberryPi, referred to as the "client" program. This program coordinates the GPS, XBee, and HackRF. It is designed such that the main function creates references to the GPS serial port and the XBee. The HackRF is only interacted with when the "hackrf_sweep" program is invoked by the XBee packet handler routine. Some more interesting functions are shown in the code samples below.

B.2.1 perform_sweep

This is the function that gets run when PiComm receives a "StartHackRFPacket"

```
void XBeeConnection::perform_sweep(std::string command)
{
    // Setup pipe and process stuff
    FILE* fp;
    char *line = nullptr;
    size_t len = 0;
    ssize_t read;

    fp = popen2(command.c_str(), "r", hackrf_sweep_pid_);

    while ((read = getline(&line, &len, fp)) != -1) {
        send_line_of_fft_data(line);
    }

    delete line; // From documentation of C's getline

    pclose2(fp, hackrf_sweep_pid_);
    return;
}
```

popen2 and pclose2 are reimplementations of the Linux popen and pclose syscalls which return the PID of the created process. This was done so that in the future this PID could be used to kill the process if/when desired.

B.2.2 send_line_of_fft_data

This funtion is what is called above to send single frames of FFT data taken from the HackRF. It first reads the GPS serial port and concatenates this with the FFT data. It then determines the number of packets required to transmit the entire line before chunking the line into that many packets and sending them individually through the XBee.

```
void XBeeConnection::send_line_of_fft_data(char* fft_line)
{
    std::string gps_coordinates = gpsPort_->readStringUntil("\n");

    int line_len = strnlen(fft_line, 1024) + strnlen(gps_coordinates.c_str(), 128);
    int num_packets = line_len / 75 + (line_len % 75 != 0);

    std::string fft_string(fft_line);
    std::string whole_line = gps_coordinates + fft_string;

    std::vector<PiCommPacket::HackRFData> packets;
    for (int i = 0; i < num_packets; i++) {
        auto packet_length = 75; // default packet length
        // if it's the last packet and is partial, it will be shorter
        if ((i == num_packets - 1) && (line_len % 75 != 0)){
            packet_length = line_len % 75;
        }
        std::string packet_string(whole_line.substr(i * 75, packet_length));
        auto packet = PiCommPacket::HackRFData(packet_string, i+1, num_packets);
        packets.push_back(PiCommPacket::HackRFData(packet_string, i + 1, num_packets));
    }
    for (auto packet : packets) {
        send_packet(packet);
    }
}
```

B.2.3 handle_packet

This function is what interprets packets received from the host. It works by matching the packet header then deserializing the packet using the methods provided by PiComm_Packets.

```
void XBeeConnection::handle_packet(const std::vector<unsigned char>& packet)
{
    using PiCommPacket::PacketType;

    PacketType packet_type = static_cast<PacketType>(packet[0]);

    switch (packet_type) {
        case PacketType::Test: {
            send_packet(PiCommPacket::TestPacket(true));
        } break;
        case PacketType::Info: {
            std::string info_string(packet.begin() + 1, packet.end());
        } break;
```

```
        case PacketType::StartHackRF: {
            PiCommPacket::StartHackRFPacket start_packet(packet);
            perform_sweep(start_packet.command());
        } break;
        default: {
            send_packet(PiCommPacket::InfoPacket("ERROR: Unrecognized Packet"));
        } break;
    }
}
```

B.3 DataLoggerGUI

The DataLoggerGUI program is built using the Qt GUI toolkit. Each device has its own widget which controls it. For example, the XBee is interacted with through the XBeeWidget. Packets are handled using a function similar to PiComm's handle_packet method.

B.3.1 gotXBeePacket

The primary difference between this method and PiComm's handle_packet method is how it handles receiving packets that make up a partial HackRF Data frame. When the class is initialized, an empty vector is created which holds these packets as they are received. This method appends each packet to this vector until all packets have been received. At that point, a "gotFullDataPacket" signal is emitted which displays the complete packet in the LogBrowser widget. After emitting this signal, the vector containing the complete frame is cleared in preparation for the next frame.

```
void XBeeWidget::gotXBeePacket(const std::vector<unsigned char>& raw_packet)
{
    using PiCommPacket::PacketType;
    PacketType packet_type = static_cast<PacketType>(raw_packet[0]);
    switch(packet_type)
    {
    case PacketType::Test: {
        emit xBeeInfo("Received successful test response");
        break;
    }
    case PacketType::Info: {
        PiCommPacket::InfoPacket info_packet(raw_packet);
        emit xBeeInfo(info_packet.text().c_str());
        break;
    }
    case PacketType::HackRFData: {
        auto deserialized_packet = PiCommPacket::HackRFData(raw_packet);
        auto packet_fragment = deserialized_packet.data();
        std::copy(packet_fragment.begin(),
                  packet_fragment.end(),
                  std::back_inserter(full_packet));
```

```
        if (deserialized_packet.index() == deserialized_packet.total()) {
            QByteArray packet_array = QByteArray(reinterpret_cast<const char*>(full_packet.data()), full_pa
            emit gotFullDataPacket(packet_array);
            full_packet.clear();
        }
        break;
    }
    default: {
        emit xBeeError("Ignoring unrecognized packet");
    }
    }
}
```