Airborne Radio Collar Tracker

CSE 145/237D

Daniel Knapp
Thomas An
Naveen Ketagoda
Sam Vineyard

Spring 2017

1.0 Abstract

Researchers often want to track wildlife to gather animal movement patterns, population size, and other valuable statistics. The main inconvenience with traditional methods is traversing difficult terrain, some of which can take an hour to walk a mile. The Radio Collar Tracker Project is an ongoing project that addresses this issue using an aerial drone to track animals tagged with radio collars. Some of the project's weaknesses were improved upon this quarter. This project update consists of creation of a radio collar emulator, GPS parsing scripts, as well as a new user-interface. This paper will cover the improvements made to previous iterations of the project as well as guidance on how the product operates.

2.0 Introduction

Researchers often want to tag and release animals into the wild and track them as time goes on to gather animal movement patterns, population size, and other valuable statistics. To track the animals, small radio collars are affixed to their bodies which emit short tones periodically at a given frequency. Using traditional methodology, researchers will attempt to find the animals using a large hand-held antenna several times throughout the course of a few weeks or less. In some places, such as the Cayman Islands and the Dominican Republic, the wilderness is tough terrain to traverse, potentially taking more than an hour to walk just a mile. The Radio Collar Tracker project is designed to address this terrestrial problem by taking to the skies. We use a quadcopter that is autonomously flown over the area suspected of containing the animals of interest. The system of electronics on the copter listens to and aggregates pings transmitted by the animals' radio collars including the copter's current GPS location when a ping is heard. The animals' positions are triangulated from the signal strength of pings heard at recorded GPS locations.

Previous iterations of the project have had issues with radio collar detection range and the copter platform handling gusty winds. One of the improvements made this quarter is focused on utilizing a new copter platform (3DR Solo) that is has more robust flight even in windy conditions. Further references to the new 3DR Solo platform will be condensed to "the Solo". Attempts have been made to access the internal GPS unit on the Solo, but this has resulted in a couple "bricked" copters. Using this problem as inspiration, we decided to go with a more loosely coupled approach using an external GPS module which has the added benefit that it can be easily moved to any copter with minimal effort. A new user interface was built for the system on the copter to display important status information and allow user input to quickly change between standby actively logging states. The last improvement to the project focuses on accurately emulating a radio collar at low power and using the exact same waveform. Due to lack of time and the likelihood that it would not prove to be effective in increasing detection range while maintaining the same energy consumption, we did not test different transmission techniques. This research should be easy to extend from the work done this quarter if it is deemed valuable in future work.

Work done relating to the external GPS module is in section 3.1. Contributions to the user-interface including a web interface are in section 3.2. The collar emulator is covered in section 3.3. The emulator is broken up into an signal analysis part and signal generation part.

3.0 Technical Material

Due to the divided project focuses under the RCT project, the technical portion of this write up will be divided into three subsections: (1) Ublox M8N GPS Module (2) Drone User Interface and (3) Radio Collar Emulator.

3.1 Ublox M8N GPS Module

This subsection of the RCT project consists of Python scripts parsing the binary stream of data sent via serial from the Ublox M8N GPS module. Previous iterations of the RCT utilized an integrated GPS chip built into the drone platform, the Solo, itself. The blocking issue with these internal GPS access attempts was consistent loss of autopilot functionality. As a result, an external GPS module is now incorporated onto the RCT platform to provide a modular GPS implementation with the Intel Joule, separate from any drone hardware. Future iterations of the RCT may now replace the drone without GPS considerations. The system architecture including the new GPS module can be seen below in Figure 3-1.

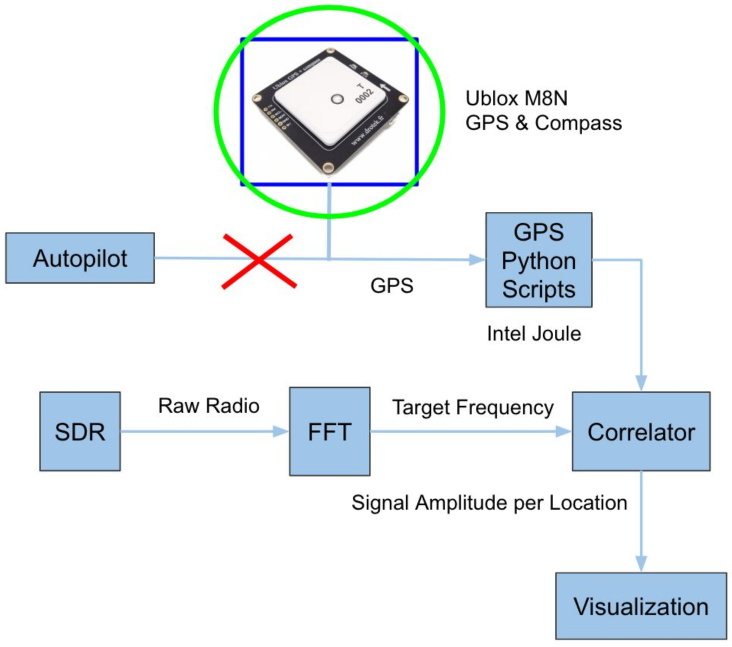


Figure 3-1. System Architecture with GPS Module

Two python scripts were written in order to handle two distinct requirements: (1) data logging for RF ping correlation and (2) 3D GPS fix check. Both of these scripts utilize the open source python library pyUblox (see Section 6.0 References for a github link) to handle the raw serial stream and to group the binary values into message fields derived from the ublox M8 protocol. The first script, ubx_gps_logger.py, parses the ublox message, NAV-PVT (Position Velocity Time) for the following fields: timestamp, latitude, longitude, altitude, velocity, and heading of vehicle. These field values are recorded in an log within an output directory specified as a command line argument. In a field implementation, GPS timestamped position from these logs will be correlated with radio collar pings to locate the animals under investigation. The second script, gpsFix.py, is called from the bash startup script that performs status checks to indicate operational readiness and that data logging can begin. The process started by gpsFix.py will only be killed if 3D GPS fix is acquired, thus the bash startup script is dependent on its success before executing the next line. A high-level flowchart of the ublox parsing process is depicted below in Figure 3-2.

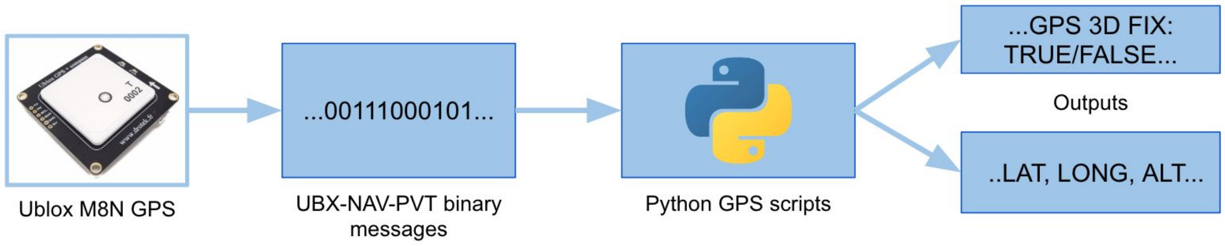


Figure 3-2. Parsing Flowchart

The GPS data logging script was demoed with the Ublox M8N during a drive around the UCSD campus. Utilizing the python library gmplot (see section 6.0 References for the github link), an HTML file was generated utilizing the position data parsed and recorded during the drive. This HTML file included plotted lines from the position data on top of a google map. A screenshot from this HTML file opened in a web browser is depicted below in Figure 3-3.

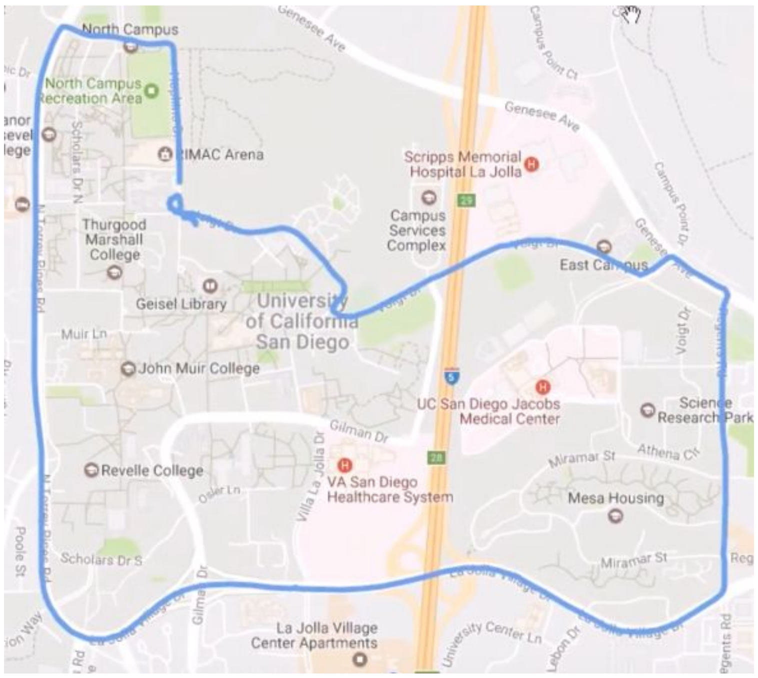

North Campus
North Campus Recreation Area
RIMAC Arena
Thurgood Marshall College
Geisel Library
University of California San Diego
John Muir College
Revelle College
Scripps Memorial Hospital La Jolla
Campus Services Complex
East Campus
UC San Diego Jacobs Medical Center
Science Research Park
VA San Diego Healthcare System
Mesa Housing
La Jolla Village Center Apartments

Figure 3-3. Screenshot of HTML-Generated Google Map from GPS Parsing Demo

3.2 Drone User Interface

This section of the RCT project mainly consisted of giving the drone better overall functionality and user-friendliness. In previous iterations of the RCT project, there were instructions to detail every single procedure required for a simple step such as how to start the drone, how to start data logging, how to stop the drone, etc. Although these previous iterations were useable, the design of it and the process necessary was rather unintuitive. The device is originally meant for researchers and biologists out in the wilderness where internet isn't necessary readily available; so, error handling is rather difficult should they occur. Aside from the complicated start up procedures, if the system were to fail the start up procedure debugging the specific blocking issue was a hassle as well. The previous iteration relied only on a single LED light that lit up if start-up was successful or not.

The first step towards simplifying this process was designing a new status check script which would condense all the checks and requirements necessary to start the drone. The state machine of the process can be modeled by the following diagram, Figure 3-3.

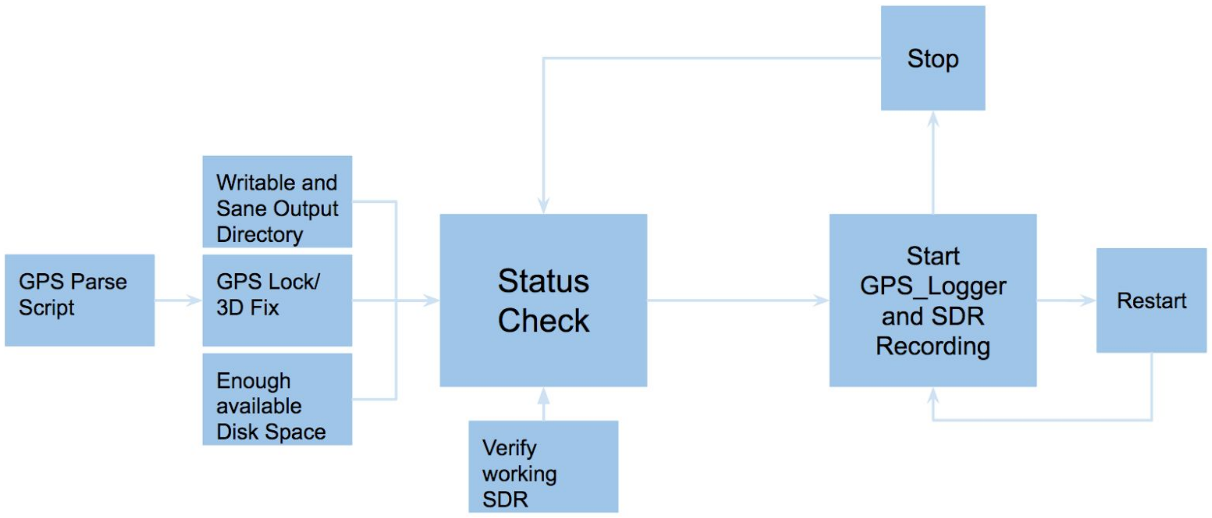

Stop
Writable and Sane Output Directory
GPS Parse Script
GPS Lock/ 3D Fix
Enough available Disk Space
Status Check
Verify working SDR
Start GPS_Logger and SDR Recording
Restart

Figure 3-4. State machine of Status Checks

In the above figure, the final script of *rct_status.sh*, the following five requirements of a writable output directory, a sane output directory, enough available disk space, and a verified successfully flashed SDR (Software Defined Radio), and 3D GPS fix (comes from GPS parsing script).. All of these checks run sequentially in the script and should any of them fail and error is thrown and the status script exits. Only when all of these checks are successfully completed, then the user is notified that he or she may begin data logging and the quadcopter awaits for the start command. When data logging begins, the status script terminates while data logging happens and will resume once data logging is ended by the user.

On top of this status script, two different user interfaces were implemented to a hardware interface and a cyber interface. The hardware interface serves as the barebones for the script since it was the simplest and most robust design. On the PCB (Printed Circuit Board), there are 5 different LED's implemented along with a single switch. Each LED used were KINGBRIGHT LED's and their mappings are: HYPERRED (orange) for a sane output directory, BLUE for a verified functional SDR, GREEN for ensuring there's available disk space, YELLOW for a writable output directory, and RED for a GPS lock found. The PCB was printed with the specifications of a 10mm by 85mm due to the Solo being such a light and compact quadcopter.

The LED circuit is implemented as a barebones physical indication of status checks. In order to extend upon this interface with the field scientist in mind, a graphical user interface (GUI) is now accessible from any web browser wirelessly connected to the Intel Joule. By simply typing in the Intel Joule's statically assigned IP address for its wireless network in the address bar, a simple and intuitive web interface appears. All five status checks from bash UI script are included, as well as the toggle switch to start and stop data logging. The web interface is depicted below in Figure 3-4.

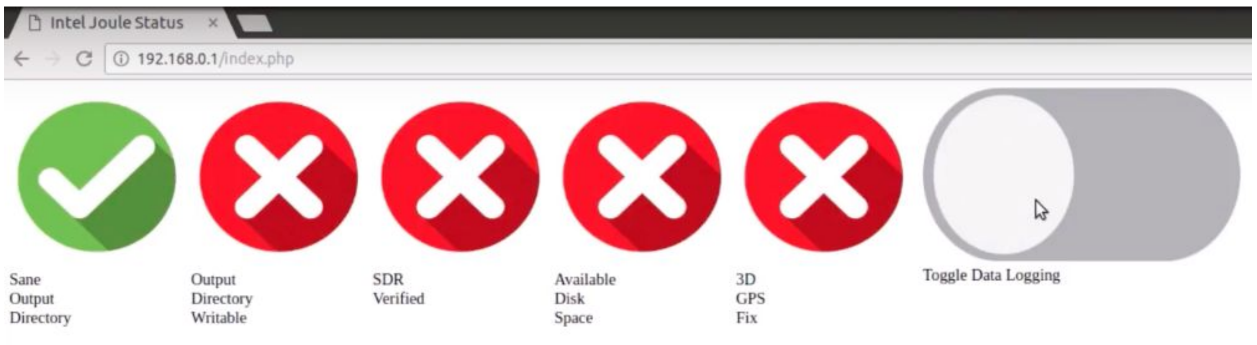


Figure 3-5. Intel Joule Web Interface

A significant portion of the efforts for this interface were configuration based. The Intel Joule and its linux distribution, Ostro, have relatively limited documentation and a lack of a package manager, causing blocking issues and delays when installing multiple components from source. After extensive configuration and overcoming the web development learning curve, the web interface V1 was completed. It consists of an Apache HTTP server, PHP extension scripts, and javascript for dynamic interactions. This system architecture for the web interface is depicted below in Figure 3-5.

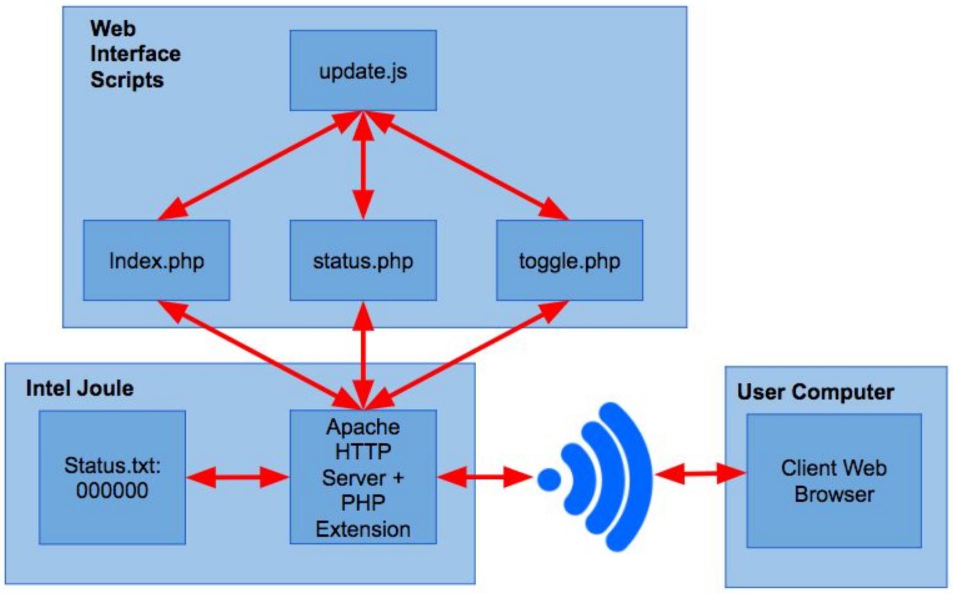


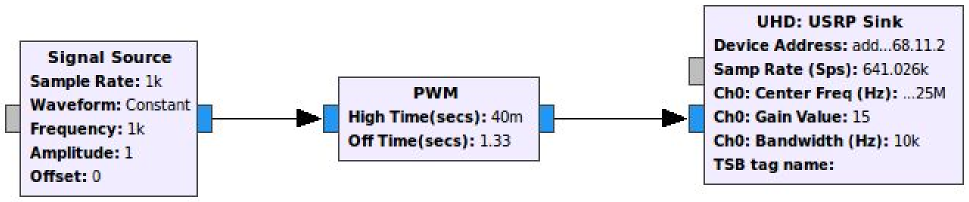
Figure 3-6. Web Interface System Architecture

A simple text file on the Intel Joule, Status.txt, contains a six digit number. The first 5 represent the LOW/HIGH state of the five status checks and the last digit is for the toggle switch. This text file is checked by the index.php to initially generate the web page, and then update.js utilizes status.php to continue status checks on the Intel Joule. To begin data logging, update.js waits for an “onclick” action on the web page which will then overwrite the last digit of Status.txt to its appropriate OFF/ON value via toggle.php. In an operational scenario, status.txt will also be checked and overwritten by the bash UI script to maintain consistency with the LED interface. Ultimately, this web interface is a foundation for future iterations to improve the front end and extend the backend capabilities e.g. real time processing of GPS and radio collar ping data.

3.3 Radio Collar Emulator

Building the radio collar emulator included creation of a block path in GNURadio companion that would allow emulation and analysis of the radio collar's transmitting signal. This includes characteristics such as amplitude, frequency, and pulse width. Some of the tools we used were the USRP-2920, a copper dipole antenna, and GNURadio Companion.

In order to recreate the signal in GNURadio, we had to initially read its existing waveform. The radio collar transmits a short pulse around every 1.5 seconds. Based on our sampling rate, we wrote binary 16-bit "1" and "0" values to a file. The number of "1"s written depended on how many samples were in 50 ms while the rest of the values were "0"s. Since one of the goals of the radio collar emulator was to have the ability to transmit different waveforms in order to test reduce the power consumption, this method was ineffective. Thus, we had to create a custom block in order to extend the emulator to be able to transmit different waveforms. Furthermore, after analyzing the collar's signal more in depth, we found out that it transmits a 40 millisecond short pulse every 1.33 seconds, which was hard to mimic with the previous method. Since there are no blocks in GNURadio that could perform the Pulse Width Modulation technique, we had to create our own in GNURadio. The block keeps a count of time to see when every 1.33 seconds occur and allows the incoming signal to pass through for about 40 milliseconds. The parameters of the block are the time length of the pulse (High Time) and the time length between each pulse (Off Time). Shown in Figure 3-3 is the block path from GNURadio Companion. Signal Source block allows to choose the waveform to transmit, along with frequency, amplitude and the sample rate. The middle block is the custom block that we created that mimics the Pulse Width Modulation technique, and it allowing the user to choose the duty cycle of the signal. The block on the right sinks the incoming signal to the transmitter (USRP).



Signal Source
Sample Rate: 1k
Waveform: Constant
Frequency: 1k
Amplitude: 1
Offset: 0

PWM
High Time(secs): 40m
Off Time(secs): 1.33

UHD: USRP Sink
Device Address: add...68.11.2
Samp Rate (Sps): 641.026k
Ch0: Center Freq (Hz): ...25M
Ch0: Gain Value: 15
Ch0: Bandwidth (Hz): 10k
TSB tag name:

Figure 3-7. Transmitting Block Path from GNURadio Companion

Before we can even transmit the desired signal, we needed to first understand the signal behavior itself. We decided upon four signal characteristics that would accurately emulate a radio collar. These characteristics are pulse width, pulse period, frequency content, and amplitude profile. They are consecutively depicted in Figure 3-8, 3-9, 3-10, and 3-11 below. The radio collar we were testing with this quarter is a falcon collar that operates at a frequency of 216.025 MHz. The National Instruments USRP-2029 was used as the receiver with a gain of 0 and center frequency of 216MHz, so the recorded collar signal should appear to be a 25kHz wave. Signal recording was implemented using GNURadio at a sampling frequency of 640kHz. The figures came from several seconds of recorded data from the collar that was as close as possible to the receiver (almost touching). All figures were created using the matplotlib library in python 3.6.1.

UC San Diego ENGINEERS FOR EXPLORATION

SAN DIEGO ZOO INSTITUTE FOR CONSERVATION RESEARCH

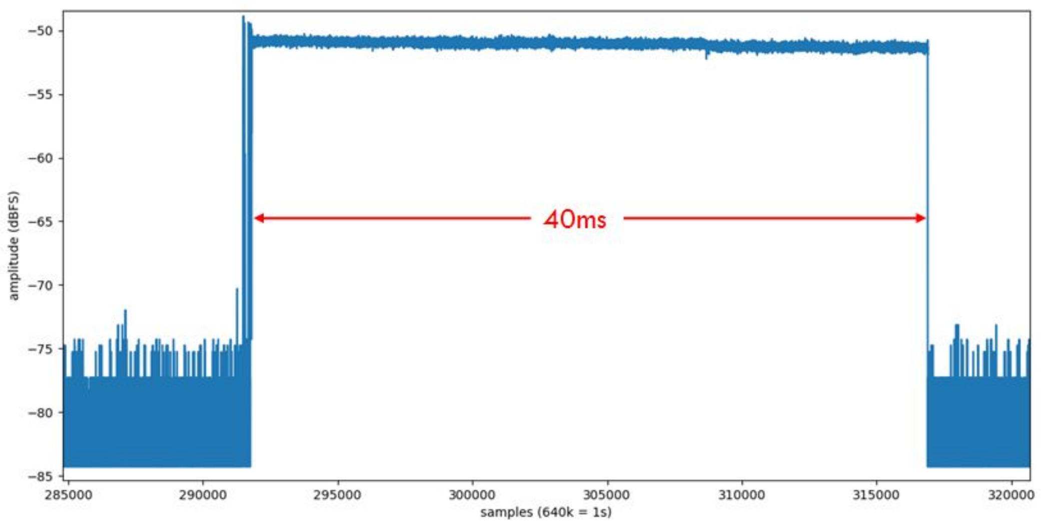


Figure 3-8. Measuring Pulse Width

The radio collars want to be as low power as possible, so they transmit a very short pulse periodically at a known frequency. In the case of the falcon collar, the transmitted frequency is 216.025MHz with a pulse width of 40ms.

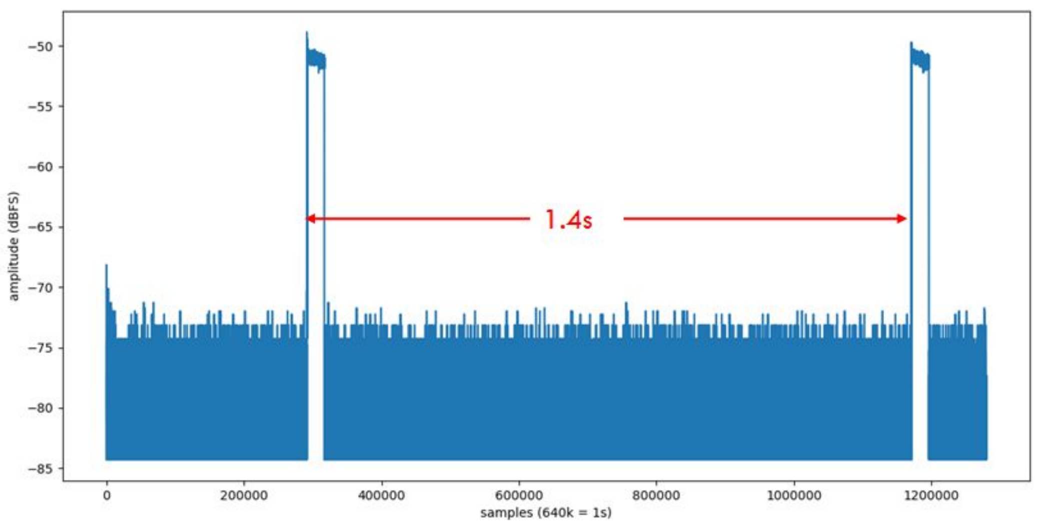


Figure 3-9. Measuring Pulse Period

The pulse period is simply the time between the start of one pulse to the start of the next pulse. Ideally this is exactly the same for every pair of pulses. For the falcon collar, the time between any pair of our measured pulses was within 1/1000 seconds which is good.

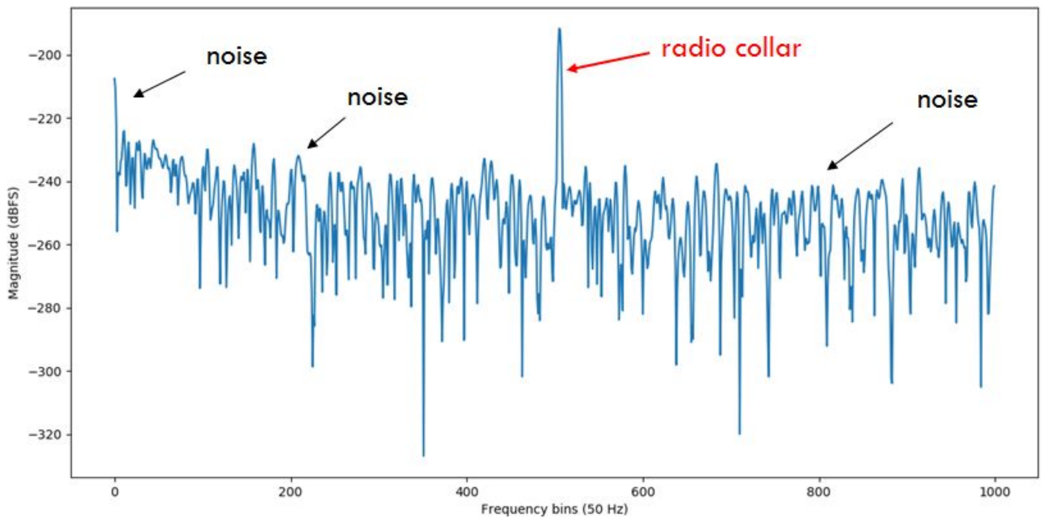


Figure 3-10. Measuring Frequency Content

The frequency content metric is focused on what frequencies are actually transmitted from the radio collar. An ideal transmitter in this case would have all of its power focused at exactly one frequency, but this is not the case in practice. The transmitted signal will generally power transmitted over some amount bandwidth. In the case above, the signal is present at a 25kHz offset as expected, but it has a width of around +/- 100Hz.

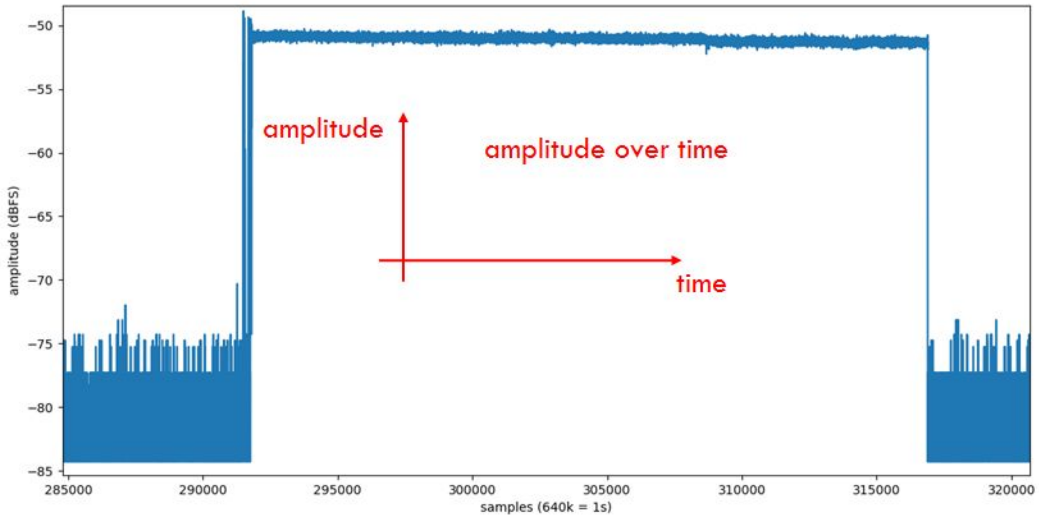


Figure 3-11. Measuring Amplitude Profile

The amplitude profile is essentially just the magnitude of the recorded complex (IQ) data plotted over time. This will be most useful when looking at the sampling range within the start and end of a single pulse. From this we can try to accurately mimic even the small fluctuations in power at every point in time. This seems highly susceptible to the noise in the environment though, so it's possible some filtering may be useful here before transmitting. The downside of filtering is that certain small characteristics of the original waveform may be lost. Determining its impact would likely require some more extensive field testing. If these small characteristics make little difference to our detection, then a much simpler approach would be to average the amplitude of the pulse and transmit the result as our emulated pulse.

There are python scripts to help with this analysis as well as do the plotting. They require a good amount of technical skill and understanding to utilize as there are a number of parts the the analysis that seemed difficult to completely generalize. As such, they would only be used by our engineers to analyze radio collars. In our repo is the folder "Collar_Emulator/analysis" that contains the two scripts useful for analysis. The analyzeSignal.py script plots the recorded collar data in amplitude of the signal vs time in samples. fftSignal.py is the other useful script that does a bunch of short time fourier transformations and plots the amplitude vs frequency in 50 Hz bins.

4.0 Milestones

The GPS portion of this project was divided into two sub-milestones: (1) completion of two functional parsing scripts for logging and GPS fix utilizing an open source ublox python library and (2) integration and testing of the scripts with the Intel Joule. Initially, a significant amount of time was spent researching a python library for the proprietary ublox protocol, significantly more uncommon than the NMEA standard. The library included numerous scripts and tools, however, only the ublox.py script was needed to convert raw binary values into parsable string formats. The ublox.py script did not include the NAV-PVT message that was desired so a new message had to be defined, configured, and tested. This first milestone was completed on time halfway through the quarter, which enabled early testing with ping correlation scripts utilized in earlier RCT iterations. The results were satisfactory.

At the initial start of this quarter, the UI script had originally been paced as completing the tasks of: having a fully functional UI-script (status script) by week 6, having a fully functional PCB by week 7, and allocating the rest of the time to the web interface portion. Overall, everything stayed on track. There were some difficulties at the very start of the project with familiarizing the members with the project and the overall architecture of it as well as different scripting languages. Aside from those obstacles everything progressed smoothly. The initial time allocation for creating the status script was 3 weeks (until week 6) and that was completed on time. After that came the PCB design. The PCB design was a little bit rushed and tasks had to be delegated wisely but in the end the PCB came out on time. As for the web interface, it actually progressed further than we had expected. Initially we had thought that we would have at least a framework set up but instead we were able to create a (simply) fully functional web interface to interact with the quadcopter and display all sorts of notifications and updates.

For the radio collar emulator's first milestone, we started by getting familiar with GNURadio in order for us to recreate the radio collar's signal. We were able to analyze and regenerate the radio collar's signal by week 3. Based on our sampling rate we obtained, we wrote binary 16-bit "1" and "0" values to a file that GNURadio read to transmit. This works, but is not very configurable. So for our second milestone, we created a custom block in GNURadio that will allow us to tweak our signal as needed. The custom block mimics the Pulse Width Modulation technique. After implementing more precise parameters to the block, we notice that the block would skip some of the pulses that should be outputted. We think this may be due to the performance of the computer performance since it would work for lower sampling rates. The other constraint we had was the limited knowledge in digital signal processing. But with the help of Nathan, Google and Curt Schurgers, we were able to complete some of the milestones we set out for ourselves.

Initially, we expected to test the emulator towards reducing power consumption and increasing the collar's detection range by testing different combinations of waveform, pulse width, and gain. But due to time constraints, we weren't able to get to a point to try this part out. With the help of the documentation we created for this project, this can be achieved in the future.

5.0 Conclusion

In this day and age, as people grow into a global mindset in terms of taking care of our planet, understanding how humans affect the environment has become more important than ever. Learning about wildlife is one of these important topics that is ever urgent especially as more and more animals are added to the endangered list. Being able to track these animals plays a large part in beginning to comprehend how our decisions and other factors affect wildlife populations. Researchers from places like the San Diego Zoo take a significant interest in this area and anything we can do to make the process of tracking animals easier is a big help to the cause. Enabling researchers to use a drone to cover much of the hazardous terrain that would otherwise have to be covered on foot should save time and be safer. The Radio Collar Tracker has had some success in previous field deployments and will likely continue for years to come. As such, the improvements made this quarter were focused on ensuring the project will be successful in future iterations.

The GPS pipeline we have developed uses an external GPS module that can be easily moved to another platform as needed. This was a great benefit that come from deciding not to use the internal GPS of our current copter. Future iterations of the project should be able to treat this functionality almost as a given.

The user interface is a significant improvement from the previous iteration that used a single status LED. The backend of the web interface includes various error checks and provides status for each to the front end to clearly see any issues with the logging portion of the system. It is also easy to understand for non-technical people with large icons and clear messages. Adding in the extensibility of our user interface and we have a lot of added value for now as well as future work.

Lastly, the collar emulator was a useful improvement for a few reasons. One, we often have limited access to fully functioning collars to test. A few months before the field deployment we may receive one or two collars to test with. Many collars have batteries designed to last three weeks and have a shelf life of around six months. As such, after the deployment we may not even have collars to test with. Having a consistent baseline to test with using the USRP helps to avoid all these issues because we need the collar to be functioning just long enough for us to record it after which we'll be able to emulate it for all of our further testing. That makes this a really nice tool to have going forward with this project as well.

6.0 References

Reference	Link/Document Title
Python library: gmplot	https://github.com/vgm64/gmplot
Python library: pyUblox	https://github.com/tridge/pyUblox