Abstract

Researchers often tag and release animals to observe the animals' behavior within their habitats. Researchers must then roam the expected habitat with bulky equipment while manually tuning a radio to detect weak and intermittent signals from the animals' tags, a laborious and inefficient task. The Radio Telemetry Tracker team uses a drone outfitted with sensors to more efficiently locate animals based on the strength of detected signals. This quarter, I created a simulated development environment to reduce the need to time-share access to the drone and allow work to proceed remotely. I implemented portions of the communication pipeline and tested the functionality to collect and package data from the on-board sensors. To improve testing capabilities, I defined a new command packet that controls simulation of individual components on the drone.

1 Introduction

Tagging and tracking animals provides important information about biological diversity, the impact of individual species to the global ecosystem, and the benefits and harms that a species may cause for humans [10]. Thanks to advancement and miniaturization of sensors, animal tracking is a mature field with ongoing research, not just into animals' movements, but to explore and define best practice approaches [5].

An ideal application would allow researchers to search for an animal's location and receive real-time updates regarding the animal's movement. Unfortunately, researchers suffer from a manual process that remains far from this goal. Small tags and batteries are often a necessity when dealing with smaller-sized animals. To preserve battery life, the sensors may emit a weak, or even intermittent, signal. The researchers must then manually tune the radio to search for these signals while exploring difficult terrain, all while hoping the animal hasn't moved since the last signal reading. The overall process proves highly inefficient and leads to sub-optimal results. Combined, these issues significantly limit a researcher's ability to track multiple animals within a single expedition given time and budget constraints.

As commercial drones become more ubiquitous, drone-based animal tracking for terrestrial and marine applications has itself become an area of research to explore and formalize new techniques [2]. These approaches range from using drones to accompany long-term research on previously tagged animals to deploying camera-equipped drones to detect animals within raw video feeds [1, 8].

The **Radio Telemetry Tracker** (**RTT**) project is an ongoing Engineers for Exploration [6] effort at the University of California, San Diego. The RTT team uses drones to perform the tracking process in an effort to approximate the ideal application. By outfitting the drone with a software defined radio, the drone can detect the signals coming from the animal's tag. Since we now control the sensors via software, the system can check for signal readings much more frequently. Furthermore, we can dynamically scan for different frequencies in an automated fashion to track multiple animals within a single flight. Custom location prediction algorithms consume these more frequent readings, as well as data from sensors on board the drone that provide its current location and heading. The result allows the drone to further optimize its flight path. Since the drone can wirelessly communicate with computers on the ground, the researchers can monitor the drone's status and progress in real time. Furthermore, this wireless channel enables two-way communication, so the researchers can dynamically feed information and commands to the drone during its flight.

The RTT project is a mature effort to track tagged animals, and previous iterations of the project demonstrated the feasibility of a drone-based approach [3, 4, 7, 9]. While the flight time of the drone restricts when tracking occurs, the overall results become much closer to the ideal. As Figure 1 shows, the increased frequency with much we take measurements enables the location estimation algorithms to generate heatmaps indicating the likelihood of the animal's location. This approach significantly reduces the time and physical energy needed to track one, or possibly multiple, animals.

Given the release and discontinuation of various sensor and microprocessor lines, as well as the desire to optimize the location estimation algorithms, the RTT team is performing a refresh of the drone. To iterate on this drone-based approach, we require: a simulated development environment to reduce the need to time-share access to the drone and allow work to proceed remotely in a COVID-19 environment; a communication system among the components on board the drone; and the ability to successfully read data from the on-board sensors. Section 2 provides an overview of the various components of the system and highlights the components most relevant to my work. Section 3 discusses my goals and outcomes with the RTT team this quarter. I conclude in Section 4 with a discussion of the progress achieved and remaining work.

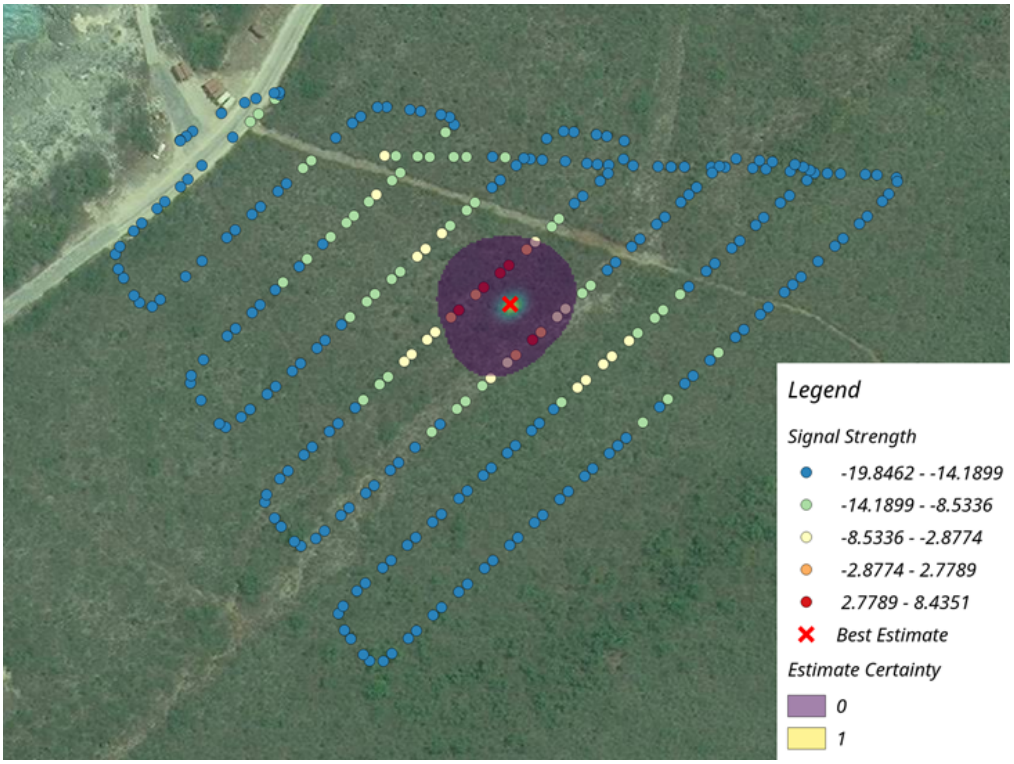

Figure 1: An example flight and associated sensor readings by an initial iteration of the RTT's project. The dots demonstrate the drone's flight path, and each individual dot represents a sensor reading. The different colors denote a reading within the listed range of signal strengths. The heatmap in the center represents the ping location algorithm's estimate of the location of the animal.

2 Technical Overview

2.1 Overview of Drone's Schematic

Figure 2 provides an overview of the components that comprise the system. Below, we discuss the relevant components for my work this quarter: for ease of discussion, I refer to them as the **User Interface Board** (**UI Board**), the **On Board Computer** (**OBC**), and the **Ground Control System** (**GCS**).

2.1.1 User Interface Board

Module 11 in Figure 2 represents the UI Board. Physically, the UI Board is an ATmega32 attached to the drone. The UI Board interfaces with the on-board sensors and packages the data for use by the OBC. In particular, the GPS pushes information to the UI Board every one second in National Marine Electronics Association GPS data format via a Serial channel. The UI Board processes this information byte-by-byte using a finite-state-machine-like algorithm until it receives a valid GPS reading. Upon receipt, the UI Board manually polls the remaining sensors (currently limited to a compass) and forms an internally-defined data packet. Each data packet contains a timestamp so that other modules can later associate the drone's position with any detected pings. The UI Board hands this packet off to the OBC via a Serial channel for further processing.

2.1.2 On Board Computer

Module 14 in Figure 2 represents the OBC. Physically, the OBC is an UP Core 0464 attached to the drone. The OBC acts as the coordinator among all of the various components on board the drone and interacts with the GCS. The OBC holds two main responsibilities that concern the work I performed: first, the OBC receives the data packets created and sent by the UI Board via a Serial channel connecting the two components. The OBC also receives each of the pings that the on-board software defined radio detects. Upon receipt of a ping, the OBC associates it with the most recent data from the UI Board. The OBC then sends this combined information to the ping estimation algorithm. Due to computational limitations, the ping estimation algorithm runs on the GCS.

The second role of the OBC involves sending heartbeats containing the drone's status to the UI Board and the GCS. The UI Board uses this information to set LEDs on the drone that correspond to the status contained in the heartbeat. We discuss the GCS's role in more detail next.

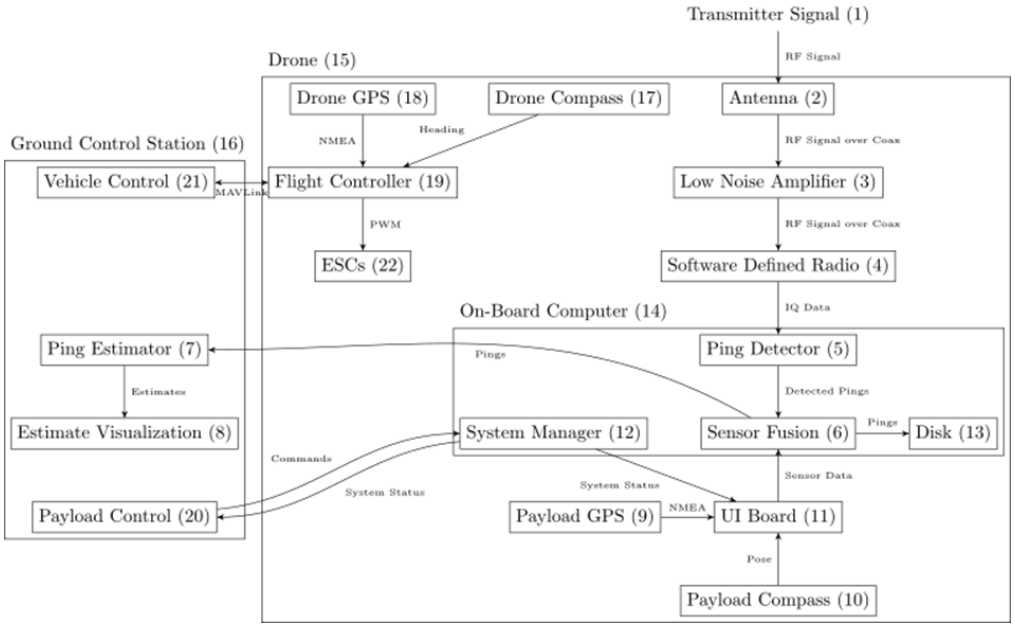

Figure 2: An overview of all components needed for the RTT project, including both on the drone and on the ground. Of important note for this paper are modules 11, 14, and 16 which I discuss in Sections 2.1.1, 2.1.2, and 2.1.3 respectively.

2.1.3 Ground Control System

Module 16 in Figure 2 represents the GCS. Physically, the GCS is a user interface that runs on a laptop located on the ground with the researchers. The drone (via the OBC) communicates to the GCS via Wi-Fi and enables the researchers to track the drone's progress in real time during the flight. The OBC relays the drone's status to the OBC via the heartbeats, and the GCS displays this information on the screen. The OBC also sends data packets containing the drone's sensor readings, as well as the ping information, to the GCS. The GCS feeds the data to the ping estimation algorithm and displays the result in a visual user interface (recall the heatmap in Figure 1).

The channel between the GCS and OBC allows for two-way communication, enabling the researchers to send configurations and commands to the drone during the flight. This allows for dynamic, real-time control of the drone and the individual modules.

2.2 Communication Protocol

The overall system maintains four main communication channels:

1. UI Board to OBC
2. OBC to UI Board
3. OBC to GCS
4. GCS to OBC

Channel 1 is a Serial communication line used by the UI board to send data packets to the OBC. Channel 2 is also a Serial communication line which the OBC uses to send heartbeats to the UI Board. The OBC also sends configuration or command packets to the UI Board via channel 2. These packets inform the UI Board to update its own configuration, or to update the configuration of the attached sensors.

Channels 3 and 4 are a Wi-Fi connection between the drone and the GCS on the ground. The OBC uses channel 3 to send data packets and heartbeats down to the GCS. The researchers can use channel 4 (via the GCS) to send command or configuration packets to the OBC (which, in turn, the OBC may forward to the UI Board).

The team previously designed and documented a custom protocol that runs on top of each of these communication channels. The core of this documentation specifies the format of each packet, as well as its intended use and expected side-effects. The system currently defines five classes of packets:

1. Status/Heartbeat
2. Configuration

3. Command
4. Upgrade
5. Data

The UI Board and OBC make extensive use of data packets to send sensor and ping readings. The OBC is the sole component that initiates status packets, but both the UI Board and GCS receive status packets in the form of heartbeats. The documentation defines many other different configuration, command, and upgrade packets that are pending implementation.

3 Milestones

Towards helping the RTT team develop the new iteration of the drone, I worked on several features during the quarter. I describe each below and discuss the status and impact of the work. As the only team member also enrolled in the class, I personally worked on every milestone below (unless explicitly mentioned that the work occurred with another team member).

3.1 Simulator

COVID-19 forced many individuals to work remotely, and the same situation holds true for the RTT team. To enable work on the project to continue, the team developed a simulator early on in the pandemic. As team members shifted to working on aspects of the project that did not require access to the drone, the simulator grew out of sync with the code base.

My first task involves updating the simulator to ensure that it meets the current drone's and APIs' requirements. When the team initially created the simulator, they also developed an end-to-end test case to verify the simulator's functionality. Success of my task is measured by the ability to pass that test case.

The main challenge involved needing to immediately dive into every piece of the code. The simulator should obscure the difference between the real and simulated sensors from the upper-layers of the code, as well as mimic the communication pipelines. This requires an understanding of how data flows through the actual setup to ensure that the simulation behaves correctly.

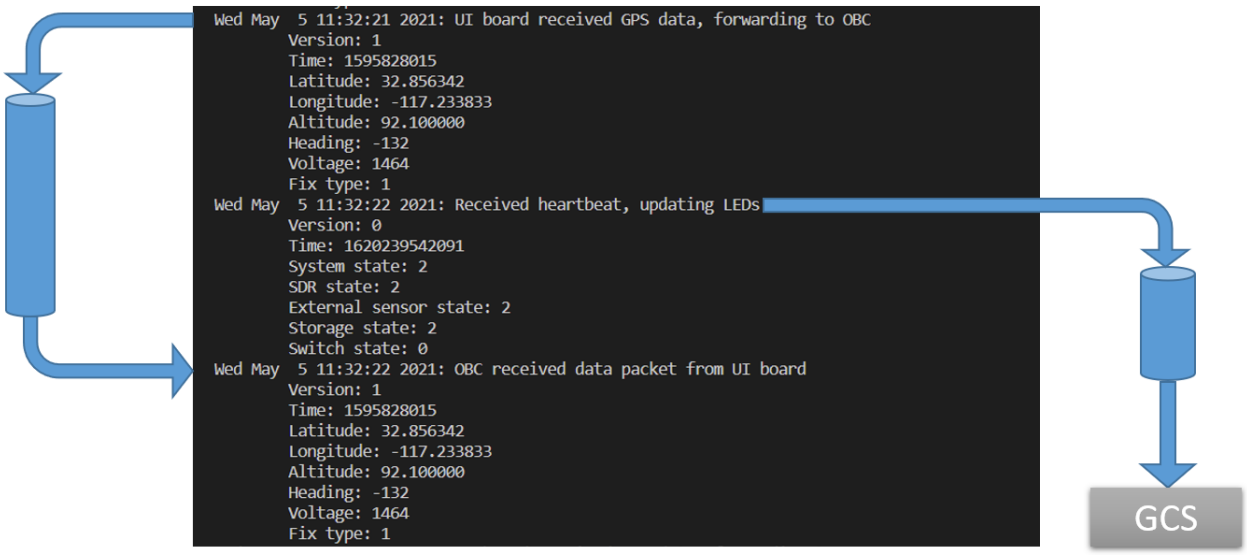

Figure 3: Example output from the simulator. The left pipe represents data sent from the simulated UI Board to the simulated OBC. The right pipe indicates a heartbeat sent from the simulated OBC to the simulated GCS (the OBC also sends the heartbeat to the UI Board, but I omit it for clarity). Notice that the data sent by the UI Board and received by the OBC are identical, as required.

Thankfully, the simulator required few changes. Instead, most of the work revolved around minor bug fixes, updating the names of the APIs, and relaxing assumptions about the order in which to initialize sensors. This last assumption proved most difficult as it involves the multi-threaded portion of the simulator. The simulator uses FIFO pipes in place of the Serial channels used in the actual communication. In C, FIFO pipes require that a reader exist for the pipe before creating any writers to the pipe. The original simulator assumed initialization occurred in this order; however, repeated executions of the simulator caused crashes when the reader and

writer threads were inadvertently scheduled such that the writer was created first. I fixed the assumption by forcing the creation to occur in the correct order, guaranteeing that the simulator starts correctly.

Figure 3 displays some terminal output from running the simulator. The existing testcase simply verifies that the simulator runs without crashing; hence, the ability to generate this output indicates success. As a further step, I added the ability to log the packets sent and received to the screen for additional verification. The left pipe in the diagram illustrates the UI Board sending sensor data to the OBC. Note that the data sent and received are identical, indicating that the communication channel within the simulation functions correctly. The right pipe represents a heartbeat (which contains the drone's status) sent from the simulated OBC to the simulated GCS. The OBC sends an identical heartbeat to the simulated UI Board which I verify as well, but I omit this heartbeat from the diagram for clarity.

With the simulator now up-to-date again, work on the remaining aspects of the drone can continue remotely. Even once the team is in person, the simulator reduces the need to time-share access to the single, physical drone and enables work to continue in parallel. As Section 3.4 elaborates on, the simulator also improves the ability to debug the software we wish to deploy on the physical drone. The presence of the simulator allows the team to compare the behavior of the software within the simulated environment to the behavior observed on the hardware.

3.2 UI Board Communication

This task involves implementing channel 1 mentioned in Section 2.2. As discussed, this channel enables the UI Board to send data packets containing information read from the sensors to the OBC. The original goal involved taking the packets and adding the functionality to send them via a Serial channel to the OBC. The simulator already implements the functionality to create the packets, and I tested that functionality when working with the simulator. The main task here involves implementing and testing the communication over an actual Serial channel instead.

As I began to inspect the existing code to determine how best to integrate this functionality, I discovered that this communication had already been implemented in a separate file. The existing code had no glaring issues, and its API was compatible with the rest of the drone. The last aspect of this goal involved testing the communication on the physical drone using the actual Serial channel; however, we encompassed all physical testing within a separate milestone (see Section 3.4), so we considered this task complete and I pivoted to other goals.

3.3 OBC Communication

After switching from the previous task, I began implementing portions of the OBC's side of the communication. As the middle-man, the OBC interacts with all of the existing communication channels. This implies that the OBC must understand how to parse data packets received from the UI Board, associate this information with received pings, and update the GCS and UI Board via heartbeats, data packets, and command packets.

This task focuses specifically on the OBC's ability to

1. Receive data packets from the UI Board via the Serial channel
2. Parse the data packet to extract the relevant information

I worked in parallel with another teammate to implement this functionality. The teammate focused on receiving the raw bytes via the Serial channel, and I focused on implementing the logic to combine the bytes and parse the sensor data from the packet.

One may notice that we can already receive and parse data packets in the simulated OBC. While true, the simulator is written entirely in C while the OBC runs entirely in Python. The team discussed simply wrapping the C code in a Python wrapper, but this was abandoned in favor of porting the code to Python. The main reason for this choice involves the existence of major portions of the other communication pipelines that carried over from the previous iteration of the drone. In order to maintain compatibility with the existing code, most importantly for ease of testing and continued development, duplicating the logic in Python proved the simpler and more effective long-term choice.

We faced two main challenges while implementing this logic. The first stems from the Windows-based machine that the teammate used. Working with Serial communication in Python on a Windows machine proved non-trivial, and it took a significant amount of work to establish a setup on her machine. With the setup functioning correctly, we worked independently on the two halves of the communication based on the existing design documentation. When it came time to combine the two halves, we discovered that we interpreted the documentation differently regarding the endianness of the packets. This required only minor changes to the code to fix, but identifying endianness as the issue required the team member to perform large amounts of debugging. Finally, we dedicated some additional time to read through the remaining documentation for this communication channel to ensure that it describes the desired endianness consistently.

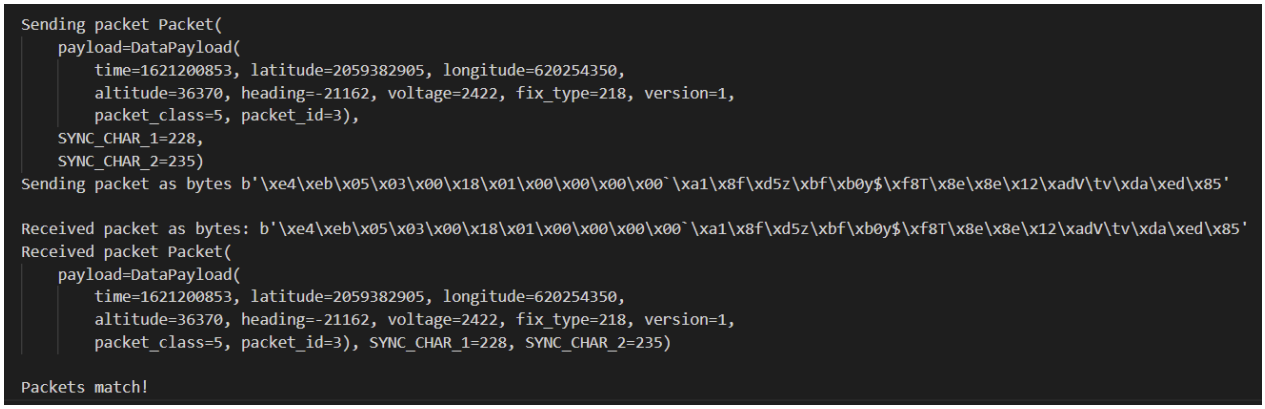

Figure 4: An example of the OBC's ability to parse packets received over a Serial communication channel. Notice that the data entering the Serial channel and the data received by the OBC are identical.

Figure 4 demonstrates the functionality of the code I developed, namely the ability to receive raw bytes from a Serial channel and create a `Packet` Python object. The `Packet` object displayed in the example contains a `DataPayload` which in turn holds the data read from the sensor. I developed the `Packet` object with extensibility in mind to allow one to easily update the payload to parse and store other information defined in the communication specifications.

With the two halves merged, the teammate moved forward with integrating the logic into the existing communication infrastructure. She performed some additional debugging, but the new logic passed all testing after integration with the existing code.

3.4 Testing on the Physical Drone

The team and I implemented and tested all of the previous work within a simulated environment. The next task required me to deploy the UI Board software onto the physical drone to expose any incompatibilities between the simulated environment and the hardware. Since I work remotely, my interactions with the drone are limited to viewing console output after compiling and deploying the software. Thus, verification involves inserting statements to log values to the console at critical portions of the main loop and verifying that the statements appear as output. In particular, the outputs we are most interested in are:

1. Reading data from the GPS
2. Creating the data packets
3. Writing the data packets to the Serial channel

While we also want the ability to read from a compass via I2C, the logic was unfortunately not integrated into the UI Board at this point. Regardless, I focused on the GPS since it operates via Serial and outputs raw bytes. This makes the process of decoding GPS data far trickier as we need to reconstruct the data one byte at a time. Since the GPS pushes data to the UI Board, rather than the UI Board polling the GPS, the UI Board has little control over when data arrives. This requires the UI Board to detect and handle situations in which portions of the GPS data are missing (for example, during initialization when flushing the Serial channel).

As expected, dealing with the physical hardware introduced several challenges. During the first two days of testing, remote access to the build server, and then the drone itself, was down. Once the team restored remote access, I successfully compiled and deployed the software onto the drone; however, running the UI Board immediately caused the software to crash. As mentioned above, the only way to interact with the software remotely is via the console output. Due to the structure of the code, console output is only available for the highest layer of the code, and libraries (even when written by us) are treated as a black box. Unfortunately, not a single line of console output appeared, and eventually I narrowed the bug down to a library we had written.

After recompiling the libraries with well-placed `return` statements to emulate breakpoints, I discovered that the issue originated with a `memset` statement in C. Since the statement caused no issues in the simulator, and we could manually iterate over that portion of memory with no issues, we further determined that the compilation process for the drone's hardware was introducing the bug. Unfortunately, as of now we could not deduce the core reason that this single statement worked within the simulator but not on the drone. Thankfully, that portion of the code was a remnant of a previous debugging effort and we could remove it with no effect to the overall functionality. With this bug removed, I faced no other issues when testing the UI Board software on the physical drone. While good news on its own, this further validates my work on the simulator and its ability to accurately emulate the environment in which the software runs.

3.5 SETTEST Command Packet

While I faced no additional software issues during the physical testing above, it did reveal an interesting remote debugging issue. Since the drone is located in a lab, the GPS module on board the drone only periodically produces valid readings (the remainder of the time it indicates that it cannot get a connection, which the UI Board software handles). Towards enabling better testability on the physical drone, we want to add a way to simulate individual hardware components, or even software modules. Unlike the previous simulator, which is required to simulate every piece of hardware, we want to be able to dynamically enable and disable the simulation of individual components at a more granular level when testing on the physical drone.

As mentioned in Section 2.2, channels 2 and 4 carry packets sent from the GCS to the OBC and from the OBC to the UI Board respectively. I defined a new SETTEST command packet that can originate from either the GCS or OBC. This new packet instructs the OBC and UI Board to enable or disable access to individual modules and instead simulate those modules' functionalities. This provides the desired dynamic reconfigurability but also improves testing capabilities by allowing one to individually examine components by dynamically enabling simulation for portions that are not the current focus.

SETTEST Command (ID 0x06)

Commands the payload to set or clear the provided test flags

Payload Offset	Type	Description	Scope
0x0000	U8	Packet Version (0x01)	All
0x0001	U32	LSB for GPS LSB+1 for UIB LSB+2 for SDR If bit=0 then real If bit=1 then simulated	All
0x0005			

This command shall result in an ACK/NACK packet and an Options packet (0x02:0x02) with the corresponding scope being transmitted.

Figure 5: The definition for a new SETTEST command packet. This instructs the OBC and UI Board to dynamically enable and disable simulation of various components while running on the physical drone.

Figure 5 displays the definition of the SETTEST command packet. The payload is a 32-bit bitmask in which each bit represents a different component. A value of 0 or 1 indicates using the real or simulated component respectively. To make the command extensible as well, I allocate a larger range than necessary so that the team can assign additional modules to the unused bits in the future. Note that, beyond simulating individual sensors (the GPS and software defined radio), the packet also enables simulation of the entire UI Board, enabling a way to focus testing even further on different modules running on the drone. Unfortunately, due to time limits within the quarter, I did not finish implementation of this new functionality and it remains as a pending item. Instead, I documented the new packet thoroughly in hopes of making the implementation a smooth process when it is required in the future.

3.6 Miscellaneous

The UI Board also interacts with a compass that operates over I2C. Another team member worked on writing a general I2C library, as well as a specific suite of functions to read from the compass itself. Since I worked closely with the UI Board this quarter, I worked with the team member to assist him with integrating the compass code into the existing UI Board infrastructure. While the team member performed extensive testing of the code separately, unfortunately he did not integrate the compass into the UI Board in time for me to include it in my testing as well.

4 Conclusion

Techniques and best practices for research that attempts to understand the behavior of animals in their natural habitats continue to evolve. As commercial drone technology progresses and becomes more ubiquitous, the RTT team hopes to merge the two fields to optimize the process of tracking tagged animals. Previous versions

of the project demonstrated the feasibility of such a drone-based approach, and the team is currently updating the sensors and software on board the drone.

This quarter, I assisted the team's efforts towards this goal by implementing a simulator that allows work to continue in a remote environment and reduces the need to timeshare access to the drone in the future. After discovering the UI Board's ability to send data to the OBC via a Serial channel already existed, I helped implement the OBC's ability to receive and parse the data packets. Since this work all proceeded within a simulated environment, I deployed the UI Board's software onto the physical drone to ensure that it works with the actual hardware. The ability to successfully run the UI Board's software on the hardware with only a single change provided further support as to the utility of the simulator as a realistic way to continue with development. To further improve testing capabilities moving forward, I defined a new `SETTEST` command packet. This packet allows one to dynamically enable the simulation of different hardware and software modules on board the drone.

The team continues to work towards the next iteration of the drone with a goal of in-flight testing at the end of the summer in a partnership with the San Diego Zoo. Many of the individual modules are implemented, and the majority of the work now revolves around merging the various pieces. The communication protocol also defines a wide variety of command, configuration, and update packets that are pending implementation as well. Other team members also continue to work on improving the ping estimation algorithms, both to improve the accuracy and latency of the implementation to improve the researcher's experience.

Acknowledgements
I would like to thank the entire RTT team for all of their help in on-boarding me and answering questions along the way. In particular, Nathan Hui, Mia Lucio, and David Salzmann provided significant support and tips for my specific tasks for the quarter.

References

[1] *Baboons on the Move*. http://e4e.ucsd.edu/baboons-on-the-move. [Online; accessed March June 6, 2021].

[2] Jan C. van Gemert et al. "Nature Conservation Drones for Automatic Localization and Counting of Animals". In: *Computer Vision - ECCV 2014 Workshops*. Ed. by Lourdes Agapito, Michael M. Bronstein, and Carsten Rother. Cham: Springer International Publishing, 2015, pp. 255–270.

[3] Nathan Hui. "Efficient Drone-based Radio Tracking of Wildlife". In: (). eprint: https://escholarship.org/uc/item/4574s85j. URL: https://escholarship.org/uc/item/4574s85j.

[4] Nathan T. Hui et al. "A more precise way to localize animals using drones". In: *Journal of Field Robotics* (). DOI: https://doi.org/10.1002/rob.22017. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.22017. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.22017.

[5] Robert E. Kenward. *A Manual for Wildlife Radio Tagging (Biological Techniques)*. English. Paperback. Academic Press, 2000, p. 311. ISBN: 978-0124042421. URL: https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=0124042422.

[6] *Radio Telemetry Tracker*. http://e4e.ucsd.edu/radio-collar-tracker. [Online; accessed March June 6, 2021]. 2013.

[7] Gilberto Antonio Marcon dos Santos et al. "Small Unmanned Aerial Vehicle System for Wildlife Radio Collar Tracking". In: *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*. 2014, pp. 761–766. DOI: 10.1109/MASS.2014.48.

[8] Jehan-Antoine Vayssade, Rémy Arquet, and Mathieu Bonneau. "Automatic activity tracking of goats using drone camera". In: *Computers and Electronics in Agriculture* 162 (2019), pp. 767–772. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2019.05.021. URL: https://www.sciencedirect.com/science/article/pii/S0168169918312894.

[9] Daniel Webber et al. "Radio receiver design for Unmanned Aerial wildlife tracking". In: Jan. 2017, pp. 942–946. DOI: 10.1109/ICCNC.2017.7876260.

[10] Martin Wikelski et al. "Going wild: what a global small-animal tracking system could do for experimental biologists". In: *Journal of Experimental Biology* 210.2 (Jan. 2007), pp. 181–186. DOI: 10.1242/jeb.02629. URL: https://doi.org/10.1242/jeb.02629.