CS 4999 Alpha Cubesat Spring 2024 Semester Report

Jonathan Ma

Space Systems Design Studio, Cornell University

I. Introduction

This report details my contributions to the Alpha Cubesat project for the Spring 2024 semester. For this semester, I mainly pair-programmed with Cameron Goddard as we continued development of the ChipSat flight software, which included finalizing the uplink and downlink logic, refactoring the GPS parser, and various bug fixes. I also worked with Sean Zhang on the ground station, redeploying it on a lab computer and improving the Kibana dashboards.

II. Mission Overview

The Alpha mission aims to validate the performance of a retroreflective solar sail, which will be deployed by a 1U CubeSat in LEO. After the release of the sail, data will be collected from four PCB-sized ChipSats positioned at the corners of the sail. The CubeSat's flight software will manage the sail deployment, transmit sensor data using the Iridium satellite network, process received commands, and oversee attitude control. The ChipSat flight software will handle sensor data transmission through a LoRa radio and execute basic commands. Meanwhile, ground station software will aggregate the received sensor data into graphs while providing an interface for mission operators to send commands. Alpha is scheduled to launch aboard a Falcon 9 rocket, with CubeSat deployment from the ISS through NanoRacks. As of the current schedule, the fully completed CubeSat must be delivered to NanoRacks by the end of September with a launch around mid-December.

III. ChipSat Flight Software

Last semester, Cameron and I mainly worked on slowly building up the object oriented version of the ChipSat FSW based on Josh-Umansky Castro’s Arduino script he was using to validate the hardware design. We had successfully integrated the IMU, temperature, and radio downlinking/uplinking, but ran into a memory issue with the radio once we integrated the GPS.

Radio Listen Memory Bug

The first issue we tackled was a memory bug we discovered near the end of last semester. We noticed that after adding the GPS parsing code from Josh's script, listening for commands using the radio would repeatedly result in error code -3, `ERR_MEMORY_ALLOCATION_FAILED`. After diving into the source code of RadioLib, the LoRa radio library we were using, we narrowed down the cause of this issue to our use of Arduino strings to store the received command. Examining the implementation to the `PhysicalLayer::receive()` function, we saw that the function was dynamically allocating a `MAX_PACKET_LENGTH = 255` byte array since we were not supplying it with the expected size of the packet. We reasoned that since the microcontroller could not find a contiguous 255 byte block of memory, it returned a null pointer, leading to the RadioLib error described above. Realizing the inefficiency of using strings to transmit data, instead passed in a fixed length `uint8_t` array to the receive function, which fixed the issue and allowed the ChipSat to listen for commands.

GPS Parsing Refactoring

As part of our debugging for the previous issue, we realized that the current GPS sentence parser which we took from Josh's Arduino script contained a lot of array operations. Specifically, it created a lot of static arrays in order to parse the raw UTC time string into hours, minutes, and seconds as well as the raw latitude and longitude strings into minutes and degrees. Hypothesizing that the unnecessary complexity could be behind the error we were seeing, we looked into ways that we could replace the script with something simpler and more robust.

First, we looked into using a popular GPS sentence parsing library called TinyGPS++. Unfortunately, we quickly discovered that it was not possible to use it since our code was already using ~90% of the microcontroller's 2KB of flash memory and loading the library pushed us over the max. Therefore, we decided to rewrite the GPS parser ourselves, trying to keep the code as simple as possible. Every GPS sentence starts off with a $ sign followed by its type (such as GPGGA). As a side note, GPGGA is the most common GPS sentence type and contains the three data points we want: latitude, longitude, and altitude. The type is then followed by a comma separated list of values specified by the NMEA message standard until it is terminated by a newline. An example is shown below:

`$GPGGA,052315.000,2447.09094,N,12100.52369,E,2,12,0.6,97.9,M,19.6,M,,0000*6B<CR><LF>`

So every time our parser encountered the start of a sentence, it would keep track of the number of commas that it had seen. Additionally, it would copy each non-comma character into a buffer. Each time a comma is encountered, the parser would check the number of commas to determine if the buffer contained either the UTC time, latitude/longitude, latitude/longitude hemisphere, or altitude. If it did, the buffer would be copied to the corresponding SFR field and cleared.

However, we quickly ran into another problem. Now, each time the software tried to downlink a packet, a silent crash would occur followed by a restart of the code. We attempted numerous debugging strategies, but what confused us initially was that the bug was difficult to reproduce consistently. We tried commenting out various portions of the code such as the parsing function or reading from serial, but sometimes it would work and sometimes it would not. We ruled out an out of memory issue since the parser used static allocation and we found that we consistently had 600-700 bytes of free RAM. Testing with an oscilloscope ruled out a power issue. We also tried using two debuggers, PlatformIO's flash debugger as well as ATMega's proprietary debugger, without much luck. The first one required a custom interrupt handler that conflicted with RadioLib's while the second one used a custom compiler which could not compile the library dependencies. A few times, we thought we got a breakthrough when the transmit function returned the error code -100, only to find out the error was not real since we were storing the 16-bit error code as an 8-bit integer and the 8 LSBs did not match any of the existing codes. Finally, after adding serial prints and delays after every line in the RadioLib functions we were calling, we realized that `SX127x::transmit()` was calling `SX127x::startTransmit()`, but it appeared like the program was not stepping into the function and crashing instead.

After taking a closer look at our parsing code and printing out the GPS sentences that were being read from serial, I realized that we had made a crucial oversight. We had set the length of the input buffer to 11 chars since according to the NMEA spec, the max length of a field was 11. However, as shown below, we soon realized that roughly half of the messages in the serial were corrupted and invalid GPS sentences, some with a malformed message ID, some with missing commas/fields, or some with just completely random characters.

```
$GPGGA6215118.000,0000/00000,N.00000.00000.E,0,00,0.0,0.0,M,0.0
$GPGGA,215414.000,0800.00000,N,00000.00000,Q bÇÇbÇrÇbÇrÇbj±ÇrÇb
$GPGG@,2LMì&rÇÇÇbÇÇÇÇrÇÇÇÇÇbs±ÇÇÇÇÇrÇÇÇÇÇb*±ÇbÇÇbÇrÉbÇrÅbj±ÇrÇb
$GP‘T±íä™ä™rÇÇÇbÉÇÇÇrÇÇÇÇÇbr±ÇÇÇÇÇrÇÇÇÇÇb*±ÇbÇÇbÇrÇbÇrÇbj±ÇrÇb
```

As a result of the missing commas, the fields were much larger than expected, causing the buffer to overflow since we did not have a length check when writing to the buffer. We hypothesized that this would eventually overwrite the function call to `startTransmit()`. After adding an `isValid` boolean which would stop parsing the current message if the message ID was invalid or if a field had greater than 11 characters, the ChipSat was able to successfully downlink.

This worked great, but during our code review we made a realization. The main reason we had chosen to write our own parser instead of using a pre-existing library was because we did not have enough flash space. By now, Cameron had cut out a lot of dead code, specifically the non-LoRa functionality in RadioLib and CubeSat-specific code that we did not need, putting us at around 80% usage. Other benefits of using a library include being able to parse other GPS sentences besides GPGGA that contain latitude and longitude, better fault handling, and using proven code (due to its popularity within the embedded software community). In the end, after adding the TinyGPS++ library to the codebase, the flash usage only increased by around 4%.

One difference between TinyGPS++ and our parser was the way invalid messages were detected. We mainly used the field length, but the library used the checksum bits at the end of each sentence. While more robust, we ran into an issue where it seemed like the GPS module was not adding checksums to the end of its messages. However, we realized this was occurring since GPGGA messages are 79 bytes long but maximum buffer length of the SoftwareSerial was 64 bytes. Luckily, as shown below, the fix was as simple as changing the max length constant in `SoftwareSerial.h` from 64 to 128 bytes (since this has to be a power of 2).

```
$GPGGA,215606.000,4226.68369,N,07629.00884,W,1,13,1.0,249.4,M,-
$GPGGA,205312.000,4226.78006,N,07729.00900,W,0,11,1.3,233.7,M,-3
3.6,M,,0000*6C
```

In terms of testing, we did a mix of testing using both real and simulated data. Real data testing was done by acquiring a GPS lock outside and verifying that the parsed values were correct. Simulated data testing was done by feeding a hardcoded set of GPS sentences, some of which were malformed in various ways, into the code and verifying that it was parsed correctly. We also tried using a HackRF GPS spoofer but unfortunately the ChipSat's GPS module was not able to pick up the signals.

Downlink Packet

Another crucial task we completed this semester was the finalization of the ChipSat's downlink packet. After consulting with Josh, he told us that the maximum transmit time he would be comfortable with was 1.5 seconds, which limited the size of our downlink packet to around 15 bytes. With these limitations in mind, we started looking at the mission critical values that we wanted to downlink, the sensor readings. After looking at the min and max ranges, we saw that gyroscope and accelerometer readings for the X, Y, and Z axes as well as temperature easily fit within 1 byte for each. And just like in the CubeSat downlink report, the values are mapped from their original ranges to 0 to 255. The readings are also saturated so if for some reason a sensor gives a value above/below the min/max range, it will report its min/max instead of rolling over. Next came the GPS latitude and longitude, with the hemisphere encoded as the sign, and altitude. After running some calculations, we realized that downlinking latitude and longitude as 2 bytes little endian would better suit the mission objectives since it resulted in ~0.25 mile accuracy compared to ~50 mile for 1 byte.

With sensor data taking up 12 bytes, we decided that it would be helpful to downlink 2 bytes of metadata so that we had context for the data. Since the ChipSats will be listening to commands (see next section), we decided to include two 4-bit rollover counters that are incremented each time a valid or invalid command is received so that we could verify if the commands were being correctly processed. We also downlink a byte of flags, such as the ChipSat's 2-bit ID so that we could distinguish each ChipSat's downlinks from each other, GPS and IMU validity flags so we know whether to trust the downlinked values, a boot mode flag so we know if the GPS is on (see future section), and a receive flag so we know if the ChipSat is listening for commands. After implementing this, we measured the time it took for the radio to transmit the 15 byte packet to an ESP32 receiver we had in the lab, which was 1.15 seconds.

Since this took less time than we had expected, during the code review we decided to add X, Y, and Z magnetometer readings since we had extra space in the packet and it might be scientifically valuable. Testing revealed that the new 18 bytes packet took 1.3 seconds to transmit, well within the requirements.

Since the ChipSats transmit on an amateur radio band, Josh needed to apply for a IARU license in order for us to legally transmit from LEO. So as part of the IARU regulations, we needed to downlink our callsign, KD2WTQ, every 10 minutes. This was implemented using a separate 6 byte ASCII packet and an if statement which replaces the normal report with the callsign every 10 minutes.

Finally, while in orbit the ChipSat's downlinks will be received by TinyGS, which is a global network of LoRa ground stations. As part of the work to prepare for an end-to-end test, I wrote and tested a parser using Kaitai Struct which TinyGS will use to decode the received packets.

Byte #	Bits	Data	Min	Max	Mapped Min	Mapped Max	Units
0-1		GPS Latitude	-9000	9000	-90	90	degrees
2-3		GPS Longitude	-18000	18000	-180	180	degrees
4-5		GPS Altitude	0	65535	0	655350	meters
6		Gyro X	0	255	-245	245	deg/s
7		Gyro Y	0	255	-245	245	deg/s
8		Gyro Z	0	255	-245	245	deg/s
9		Acc X	0	255	-20	20	m/s^2
10		Acc Y	0	255	-20	20	m/s^2
11		Acc Z	0	255	-20	20	m/s^2
12		Mag X	0	255	-100	100	uT
13		Mag Y	0	255	-100	100	uT
14		Mag Z	0	255	-100	100	uT
15		Temperature	0	255	-40	125	°C
16	0-3	Valid Uplinks	0	15			
	4-7	Invalid Uplinks	0	15			
17	0-1	ChipSat ID	0	3			
	2	GPS Valid	0	1			
	3	IMU Valid	0	1			
	4	Boot Flag	0	1			
	5	Receive Flag	0	1			

Figure 1, ChipSat Downlink Packet Format

Uplink Packet

The finalization of the ChipSat's uplink packets also occurred this semester. Although uplinks will not be needed during nominal ChipSat operations, FCC regulations require us to be able to command the ChipSats to stop downlinking if needed. These requirements provided the basis for the 2 simple commands the FSW supports, the no-op and the changing of the downlink frequency. The uplink packet is 3 bytes long, with a 1 byte opcode and a 2 byte big endian data field. The No-op command has opcode 0x00 and increments the valid command counter, allowing us to verify that we can communicate with the ChipSats. The downlink frequency modification command has opcode 0x11, with the data field being interpreted as the new downlink frequency in seconds. One orbit is ~45 minutes = 2700 seconds, so if this command is uplinked one per orbit, the ChipSats should not downlink. The ChipSats listen for commands every 10 minutes for a 30 second period. Within this period, each receive attempt listens for 100 LoRa symbols (~3.2 seconds) before timing out and retrying if no packet was received. The uplink processing logic was rigorously tested by transmitting packets with both of the 2 valid opcodes as well as various invalid opcodes in order to ensure correctness.

Boot Mode

While in orbit, the amount of power supplied by the ChipSat's will vary depending on many factors, such as the orientation of its solar panel relative to the sun. Therefore, we are not guaranteed to receive the baseline amount of power needed to operate all the sensors on the ChipSat along with radio downlinking and receiving. A situation where the ChipSat repeatedly starts up and immediately experiences a brownout since it did not have enough power to operate all the sensors and downlink was very much possible. Additionally, Josh told us that the GPS is the most power-consuming and the least mission critical sensor to receive data from. So, in order to save power and prioritize the downlinking of mission critical sensor readings first, ChipSats will enter Boot mode for the first 30 seconds after they are powered on where the GPS is powered off but all other sensors behave normally. After this 30 second period is over, the GPS is powered on.

This was implemented by immediately pulling the GPS reset pin high to disable it and TX pin low to cut power to it when the ChipSat's software powers on. After Boot mode ends, the reset pin is pulled low to turn on the GPS. Testing using a multimeter shows that adding Boot

mode reduces the ChipSat's base power consumption (so without radio transmits and listens) from ~79mA to ~22mA, a 70% reduction!

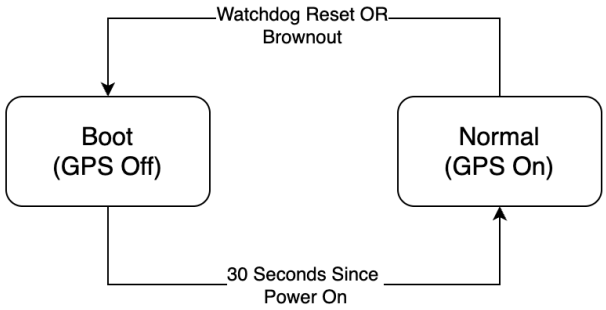

Figure 2, GPS State Diagram

Downlink Windows

Another change that we made this semester was the overhaul of the radio's state machine. The original state machine had the ChipSats downlinking once per a set time period. However, during the mission the light sail will have two ChipSats on each side, which means two of them will be on at the same time. Since Chipsats cannot communicate with each other, sticking with the original model meant that if both ChipSats turned on at the same time, there would be a high likelihood that they both downlink at the same time, causing one of the downlinks to be lost.

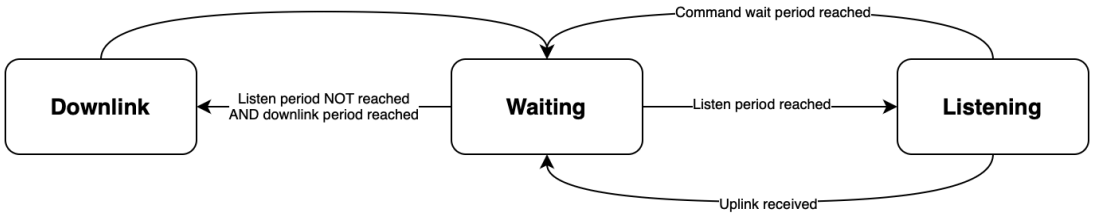

Figure 3, Original Radio State Diagram

To reduce the chance of collisions, we proposed adopting a simplified version of time-division multiple access (without synchronization). With this approach, we now have a 10 second downlink window, which is made up of five two-second slots, since each downink takes 1.3 seconds. This generalizes to arbitrarily long downlink windows: If the downlink window is `x` seconds long, we will create `x/2` two-second slots. Before each window starts, each ChipSat will randomly choose which slot it will downlink in. And after each window is over, the ChipSat

will either start another window or listen for commands if the listen interval is reached. Additionally, to ensure that each draw is completely random across boots, we seed the random number generator with noise from an unconnected analog pin. Doing the collision probability math, we get a 20% chance of collision. This is because if the first ChipSat chooses a slot, 2 out of the 5 slots would now cause a collision if they are chosen by the second ChipSat. The verification testing we performed is described in the next section.

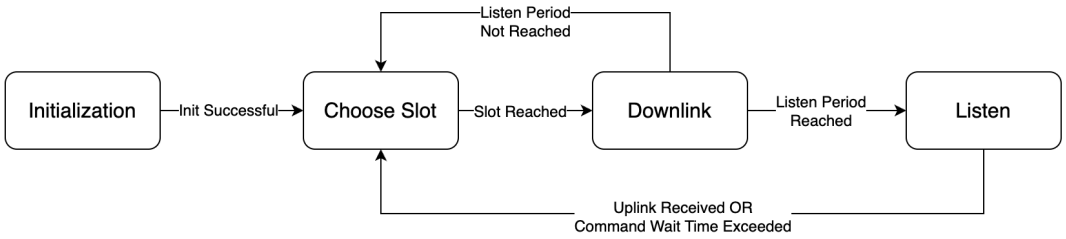

Figure 4, New Radio State Diagram

ChipSat FSW Code Review and Testing

Near the end of the semester, we turned our attention to reviewing and testing our codebase. First, we developed a requirements document for the FSW so we could evaluate whether our code would enable the ChipSat to fulfill mission requirements. We then held a ChipSat FSW code review with the leads (Josh and Lauren) and other team members (Zach), where Cameron and I went through the code subsystem by subsystem and described the design decisions we made for each. By the end, everyone had gained a good understanding of how our code met mission requirements and what improvements/tests needed to occur. In particular, we implemented the following changes: adding magnetometer readings from the IMU to the downlink packet, changing IMU initialization from an attempt limit of 5 to attempting to initialize it indefinitely, adding a maximum buffer read variable to the GPS parser to avoid going into infinite loops, and replacing the initialization state enum was a simpler boolean.

In addition to the GPS and radio testing mentioned above, we conducted a long duration test to ensure that the FSW did not crash/restart over a 2 hour period as well as a rapid power cycling test to ensure that the FSW boots to a safe state even when power is cut during initialization or radio transmissions. Finally, we also conducted a multi-ChipSat downlink test,

where we set up two ChipSats and a ESP32 receiver to verify our randomized transmission algorithm. After turning on both ChipSats at the same time, the first one transmitted 31 packets while the second sent 30. The ESP32 received 23 packets from the first ChipSat and 16 from the second, which is a success rate of 64%, just a bit lower than what we predicted. We also found that even if the downlinks of two ChipSats collided, one downlink would still be received by the ground station.

IV. CubeSat Ground Station

Ground Station Re-deployment

Near the end of last semester, I noticed that the 4GB DigitalOcean droplet we were running the CubeSat ground station on did not have enough memory to consistently host Elasticsearch, Kibana, the Flask backend server, and the React frontend server. As a refresher, Elasticsearch is used to store the CubeSat's downlinks, Kibana is used to create dashboards and graphs to visualize telemetry data, the React frontend is used to send commands and view the downlinked images, and the Flask backend server processes the raw downlink reports received from the RockBlock portal and handles user management. As a result of the limited memory available, Elasticsearch would occasionally crash with an out of memory error. Since upgrading the droplet to have more memory was deemed to be too expensive, our solution was to run the ground station on a computer in the lab instead, which had more than enough memory. After setting up the Linux partition and talking with Cornell IT about opening networking ports to the internet, we learned that unlike on DigitalOcean where we could have all the ports publicly accessible, IT would only allow us to keep port 80 open to the internet. As a result, I worked with Sean Zhang to set up a nginx configuration file which would redirect traffic from port 80 to each of the services depending on the route. For example, the "`<ip_address>/ui`" route would redirect traffic to the React server on port 3000, "`<ip_address>/api`" would redirect traffic to the Flask server on port 8000, and "`<ip_address>/kibana`" would redirect traffic to the Kibana server on port 5601. The updated ground station architecture is shown in figure 5 below and was verified through manual testing to ensure all functionality behaved exactly as on DigitalOcean. Green arrows show the path of telemetry data, red arrows show the path of commands, and black arrows correspond to user actions.

Now that we moved the ground station in-house, the security and maintenance of it was now our responsibility. Digital security was implemented by protecting both the Linux user account and all the web services with passwords as well as using SSH key based authentication for SSH access. Physical security was somewhat easier since the computer was already located in the SSDS lab, which requires card access. In terms of maintenance/reliability, I created `systemd` services for each web service. Besides starting all the web services automatically when the ground station is rebooted, this also consolidates all the application logs in one place (although we did have issues where the computer would just shut down randomly).

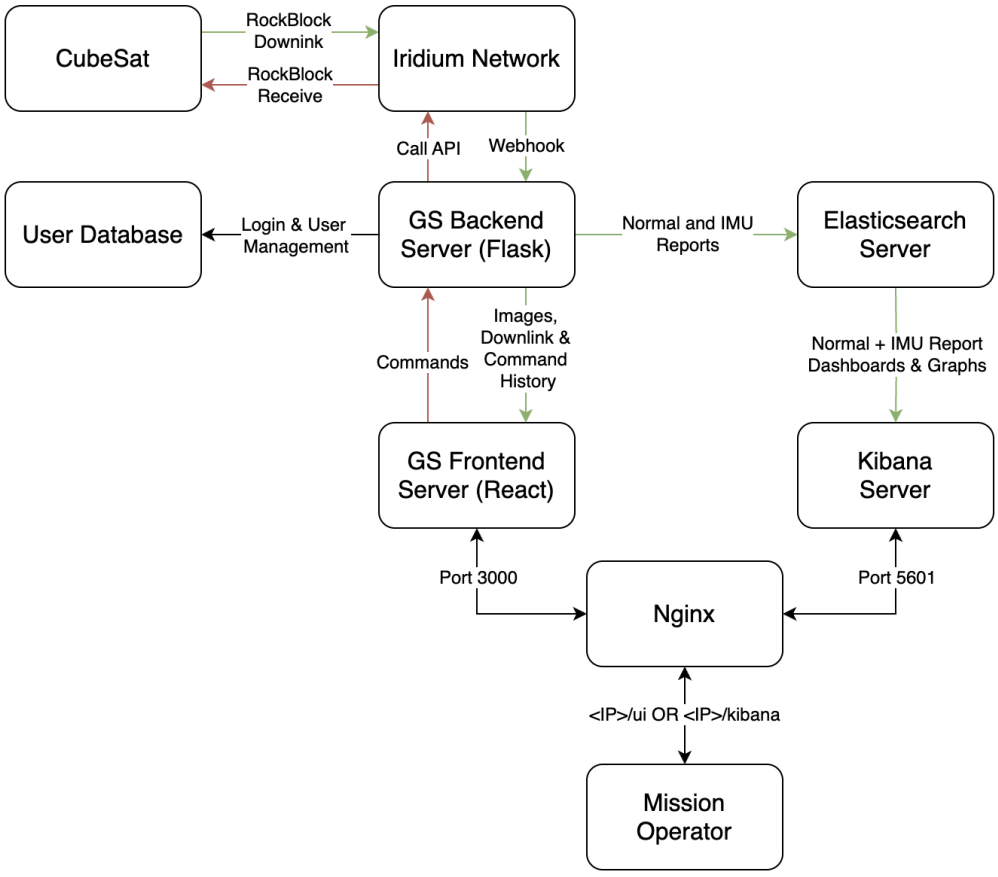

Figure 5, CubeSat Ground Station Architecture Diagram

Kibana Dashboards

Throughout the semester, I also worked with Sean to improve the Kibana dashboards that we had made in a previous semester. Previously, we only had graphs for the IMU reports and dashboards that showed most recent normal report values. This semester, we received feedback

from the leads that it would be helpful for the dashboards to also have a graph of how each SFR, SensorReading and Fault value changed over time. As seen in the figure below, this is exactly what we did, in addition to a new table to view processed commands and mission mode history. We also added filters so that users could filter between data with different RockBlock IMEI numbers, allowing us to keep flight and test data separate.

Figure 5, Revamped Kibana SFR Fields Dashboard

V. Development Process

For ChipSat FSW development, Cameron and I usually pair programmed as we worked together in the lab, which worked out well as we would usually meet 1 or 2 times a week and take turns between being the driver and the navigator. Once in a while, we would also develop features asynchronously and test them in the lab, but this was usually only done for large features such as downlink windows and the addition of the TinyGPS++ library. This was because we found that pair programming in the lab allowed us to catch errors more easily and iterate faster since we could directly test our changes on the ChipSat hardware. We also heavily utilized GitHub for version control, developing major features on separate branches and merging them into main only after thorough testing and review of the pull request. For the most part, this development model worked well. However, after implementing the GPS boot mode, we realized that we could not test it since our developmental ChipSat did not have the GPS reset pin like the newer versions of the ChipSat. As a result, we decided not to merge the branch in and continued development on the branch, causing the last PR of the semester to be larger than we would have

wanted. Throughout the semester, we also made sure to keep the GitHub wiki up to date with our changes to the ChipSat's downlink and uplink packets.

VI. Learning Goals

At the beginning of the semester, I outlined 3 CS learning goals to focus on:

1. Embedded flight software design and interface with avionics hardware
2. Software testing procedures for mission readiness
3. Principles of HCI through feedback received from ground station UI/Kibana visuals

I believe I made substantial progress towards these 3 goals this semester. For the first goal, I made major progress on the development of ChipSat FSW, which involved all aspects of embedded software design. This included designing new features under the constraints of the ChipSat's hardware, dealing with the challenges of interfacing with avionics sensors, and developing effective debugging strategies. The process of finalizing the downlink packet, designing the randomized downlink window algorithm, and creating the GPS boot mode are good examples of how radio transmission as well as power constraints led to trade-offs which influenced the final design. These trade-offs, such as balancing the amount and precision of data sent in downlinks, the frequency of downlinks, and how important some sensor data is compared to others are decisions which are unique to embedded software design. Additionally, working on the GPS parser was a good example of the challenges of working and debugging with avionics hardware. The limited flash and RAM size constraints drove me to create code that was as simple as possible, leading me to initially turn to writing a custom GPS parser instead of using a library. Then, the shortcomings of that code, a buffer overflow which was caused by the high rate of corrupted GPS messages, lead me to develop effective debugging strategies for embedded programming, such as evaluating the RAM and power usage as well as narrowing down the root cause using serial prints. While I wished that I had initially designed the GPS parser to be more robust, I feel like this was a valuable lesson that led me to gain a lot of intuition for debugging embedded systems, something which will certainly be useful in the future.

For the second goal, this was accomplished by participating in code reviews as well as conducting extensive testing on the ChipSat FSW. For the code reviews, I participated in two of them, one for CubeSat FSW and the other for ChipSat FSW. The CubeSat review showed the

importance of having multiple smaller reviews throughout the development process instead of having a big one at the end while the ChipSat review helped me gain experience with writing mission requirements. In terms of testing, we conducted many different types of integration tests involving both the ChipSat hardware and software. Whether it be testing the GPS parser with malformed messages, simulating rapid power cycling, or downlinking with multiple ChipSats, planning and executing the tests have helped me to learn how to create effective tests to verify that software is both correct and meets mission requirements.

Finally, for the third goal, I worked with Sean and other team members to finetune the design of the revamped dashboards so that they were as informative and useful to mission operators as possible. While I was not able to dedicate as much time to this goal as the other two, I still learned valuable lessons, such as the importance of balancing the amount of information shown on a single page at a time.

VII. Conclusion

With most of the major elements of ChipSat FSW finalized, development work is nearly complete. Future work for next semester includes possible adjustments to the LoRa parameters and the randomized transmission algorithm based on high altitude balloon testing over the summer. So with ChipSat hardware frozen and verified, power budget and ACS testing on the CubeSat FSW progressing smoothly, and final assembly scheduled to be completed over the summer, the team looks to be in good shape for a September delivery and December launch!