CS Capstone Preliminary Design Document

December 1, 2017

30K Rocket Spaceport America

Prepared by

Joshua Novak	Signature	Date
Allison Sladek	Signature	Date
Levi Willmeth	Signature	Date

Revision History

Name	Date	Reason For Changes	Version
Levi Willmeth, Joshua Novak, Allison Sladek	11/26/17	Initial document draft	0.1

Abstract

This preliminary design document outlines the major technical challenges, design elements, and testing goals that team 41 will complete for Oregon State University's 30k Spaceport America Cup entry in 2018. The competition involves designing, building, and launching a student-made rocket to 30,000 feet, and is scored on several criteria including a software ground station which records and displays near real time telemetry from the rocket, and a separate scientific payload.

Contents

1 Project Overview

1.1 Introduction

This project will design, write, and test software that will fly on board the Oregon State University's entry to the Spaceport America Cup's 30k Challenge. The Spaceport America Cup is an international engineering competition to design, build, and fly a student-made rocket to 30,000 feet. The competition is scored on several criteria including software components like flight avionics, recording and displaying telemetry, and conducting a scientific experiment during flight.

1.2 Stakeholders

The project stakeholders are Oregon State University, our primary mentor Dr. Nancy Squires, our ESRA 2017-18 team members including Mechanical Engineering and Electrical Engineering student subteams, and our Computer Science capstone team 41.

1.3 Design Concerns

This project can be broken into four major design components, each with unique design challenges and considerations: rocket avionics, payload avionics, receiving telemetry and stored data, and displaying telemetry and stored data. The individual component design concerns will be covered in each component section.

The overall project design concerns are that the rocket is made by students, and that we are safe at all times during building and launch, that the rocket achieves a high score during the competition, and that we act as positive role models for future students considering joining Oregon State University's rocket program.

2 Major Design Components

2.1 Overview of design elements

This project requires that multiple pieces of software and hardware work together to accomplish our mission.

The rocket will carry at least two flight computers which will record on-board flight data, as well as multiple telemetry modules that will record and transmit flight data during flight. During the flight we will capture and display live telemetry data, which will be further improved after the flight by importing stored data from the flight computers and telemetry modules.

The on-board flight computers will also perform important tasks during the flight. The rocket avionics will use inertial measurement units and other sensors to detect apogee and trigger separation of the rocket, as well as deploy the drogue and main parachutes. The payload avionics will use accelerometers to measure gravity and engage a propeller to accelerate the payload downward, to create a micro gravity environment for the scientific experiment.

On the ground, a set of affordable computers will be connected to radios which will receive the live telemetry signals. These computers will parse out text from the incoming audio stream, and store that information over a local network and into a database.

Nearby team members and spectators will be encouraged to use their laptops, tablets and cell phones to connect to a WiFi network to access a web page containing a flight summary and graphs of acceleration, velocity, and altitude, among others. These graphs will be generated in nearly real time using information from the database.

After the rocket has landed and been recovered, we will use the parsers to import additional data from the on-board computers, into the database. This will provide a large amount of additional data and allow us to display additional graphs and information about the flight.

2.2 Avionics

2.2.1 Reading from Sensors

Joshua Novak will manage avionics code that handles reading from sensors.

There are a variety of sensors that will need to be read from for this project. These include accelerometers, altimeters, and GPS devices among other sensors. Avionics code will need to be written that reads from these sensors and can be called by functional avionics code for the rocket and paylod. The output of this code will need to represent the output of the sensor, interpreted according to the standards of the device.

The code will be written in C. This is because the libraries for most of the sensors will be in C, because the code will be running on a Raspberry Pi, and because the code will need to run as efficiently as possible. Libraries will be used when they are available and will not create conflict with the restrictions placed on the ECE team.

The code for reading from the sensors should be written as complete programs that can be called to target a particular sensor and which should be able to have their output piped into functional code for avionics. In some cases this will instead take the form of a program that reads all the inputs from an analog converter and feeds those into functional code, as there will be some analog sensors used as part of the project.

Alternatively, if this is found to have significant tradeoffs in terms of speed, memory, or stability, tests will be made using an object oriented solution in C++. If these tests find that there is no significant tradeoff in terms of speed, memory, or stability to use C++ with an object oriented solution, then we will go forward with C++. If the tradeoffs in terms of speed are too significant for both, we will look into different methods to improve encapsulation for testing purposes.

Some of the avionics for reading from sensors will be written by the ECE team, as there is a coding component to their capstone. Joshua will communicate with the ECE team to ensure that the code they write is compatible with that of the CS team, and will easily be able to feed into the functional avionics code. The CS team will also work alongside the ECE team to write this code, assisting them as necessary. Alternatively, the ECE team and CS team will write code separately, with CS avionics being used on the rocket and ECE avionics being written solely to satisfy their requirements.

Unit tests for the sensor avionics will be written alongside the code. Since Joshua is also responsible for testing avionics code, he will handle this as well, and ensure that any code written by the ECE team is thoroughly tested.

2.2.2 Rocket Avionics

The ECE team will be writing some of the avionics code, which will be coordinated between the two teams. Allison Sladek will manage writing the main rocket avionics on the CS side of this project.

The main rocket hardware components will include a Raspberry Pi Zero B, Stratologger, Beeline GPS, and redundant sensors, including at least a barometer, thermometer, gyroscope, accelerometer, and altimeter. Redundant sensors can be averaged to achieve a much more accurate reading than could otherwise be achieved with fewer sensors. This will allow better analysis of the data, and more accurate apogee detection.

Separation of the main rocket at apogee to deploy both the parachute and payload is a mission critical operation. If it is triggered at the wrong time, the rocket may not reach the goal altitude, or could be in the wrong position for a successful separation. As such, it will be controlled by the Stratologger, a commercial product that measures altitude, temperature, and battery voltage, and activates separation when apogee is detected. The main purpose of the Raspberry Pi will then be to record data logs of the main rocket sensors for later analysis, and for battery control. It will also be used as a comparison to the Stratologger. This will allow testing of the Stratologgers accuracy, and ensure that deployment will function as expected after pre-flight testing.

2.2.3 Payload Avionics

Levi Willmeth will manage writing the payload avionics for this project.

The scientific payload will carry a brushless motor and ESC, several IMU's, a Raspberry Pi Zero flight computer, a video camera, and a BeeLineGPS telemetry module. The flight computer will be turned on several minutes before launch and record up-close video of the scientific experiment throughout the flight. It will also read acceleration and attitude data from each of the IMU's to detect separation and measure the amount of gravity experienced by the experiment. After separation has been detected, the flight computer will begin increasing the speed of the motor to propel the payload downward, which will reduce acceleration felt by the experiment. The goal of the experiment is to achieve zero gravity for as long as possible, which we estimate will be 10-12 seconds.

Accurately detecting separation and ejection from the body of the rocket will be a critical task for the payload avionics. If the payload begins to spin the propeller before being ejected from the body of the rocket, it will immediately destroy the propeller and possibly damage other portions of the rocket. The motor is powerful enough that if held still while being powered, it could draw enough current to easily start a fire. It will be important to develop an accurate, not precise, method to determine when the rocket has separated and the payload has been ejected from the body of the rocket.

Because we are unable to add physical components like a manual switch or wire connecting the payload to the body of the rocket, we will need to rely on our sensors. The payload will carry multiple 6 degree of freedom IMU's which will give us acceleration and gyroscopic readings in the x, y, and z planes. We will be able to use these sensors to remain in an idle state during launch, glide, and finally to detect the rather large impulse forces created during separation and ejection.

We plan to modeling the avionics system as a state machine with pre-launch, launch, glide, separation, microgravity, and parachute states. This type of direct, forward progression allows us to design a state machine which prevents the payload avionics from inadvertently returning to a previous state, and gives us a solid idea of what behaviors to look for in order to transition to the next state.

$$\begin{tikzcd}
\text{start} \arrow[r] & \text{Pre-Launch} \arrow[r] & \text{Launch} \arrow[r] & \text{Glide} \arrow[r] & \text{Separation} \arrow[r] & \text{Microgravity} \arrow[r] & \text{Parachute}
\end{tikzcd}$$

Fig. 1: Conceptual state diagram for payload avionics.

For example, during the pre-launch state we can effectively ignore all sensor readings until we see a very high acceleration on the z axis, which signals launch. We can begin logging all sensor values from that point on (or even retroactively record a set of rolling pre-launch values) until we see the z axis acceleration end, which signals that we have entered the glide state. Obviously we will need to look at more than a single sensor value to transition between states, but the general idea is to understand what we can expect during each phase of launch, and encode those conditions as state transitions.

The major design concern for payload avionics are that accuracy is more important than precision. A false early start would risk damaging or completely destroying the rocket with the payload propellers, but a late start would only risk missing some or all of the experimental data.

2.2.4 Avionics Testing

The CS and ECE teams will be writing avionics for the rocket and payload which will be able to determine when separation should occur. Due to the high degree of precision that this requires and the significant negative consequences of failure for this code, ensuring the functionality of this code is of the utmost importance. It is also standard within the aerospace industry to very thoroughly test any and all avionics code, meaning that writing a thorough test suite will be good practice for working within the aerospace or related industries. There are three types of testing that the team has decided to carry out, hardware tests, unit tests, and simulation with robustness tests. The CS team will cooperate with the ECE team to perform the first and likely the second to some degree, and will manage the third.

Hardware tests will involve making certain that each sensor is functional, powered, and giving reasonable readings. This will largely be handled by the ECE team, but Joshua will take responsibility to ensure that the ECE team has thoroughly tested every piece of electronics on the Payload and the Rocket.

Unit tests will involve making sure that the basic functionality of the code is intact. This will mean ensuring that the programs and functions written to read from sensors give accurate output to a given input. These will mostly be used to test code reading from sensors, or that the code will trigger certain events when given input that should do so (such as triggering separation when inputs suggest the rocket has reached its apex). Some tests may be written to ensure a minimum degree of robustness, such as throwing out inputs that are invalid (such as an input that is a string of characters that should be a number). Joshua will oversee the writing of unit tests on avionics code. He will check to ensure that the goals for line coverage are met by the unit tests, and aim to have the unit tests ensure all code functions correctly to expected inputs.

The final aspect of testing the avionics code is to simulate a launch with robustness testing. This will involve encapsulating the functionality of the code for triggering flight events in some manner, and then testing that code against simulated inputs. These inputs will then be randomly altered in a variety of manners. If possible, this will be extended to include testing of code for sensor interaction, with the code interacting with simulated sensors rather than actual sensors. The randomization will likely take one of the following forms, but may include other forms if the code calls for it.

- Throwing a value that is out of bounds (this included the following)
  - A value that is of the wrong type
  - A value that is too large
  - A value that is too small
- Having a sensor cease responding
- Having a sensor continually send the same value
- Offsetting a sensors outputs by some set amount for the rest of the launch
- Having a sensor feed a value that is not out of bounds, but is not correct
- Having a sensor feed values for the rest of the launch that are incorrect and
  - in bounds
  - out of bounds
  - either in or out of bounds

The number of randomized failures will be able to be set at the start of the test, but not timing or the types of failures. Some failures may be weighted as being more significant, and therefore valued as a larger number of failures. The test will not be given a clear pass fail, but will instead push the outputs of the avionics code to a csv, which will be compared against the csv output of the code with no randomization as well as the inputs given by the simulation.

2.3 Telemetry and Parsing Flight Data

2.3.1 Transmitting Telemetry

The ECE subteam selected the BigRedBee BeeLineGPS telemetry module which cannot be used with additional sensors and only transmits using APRS formatting. This means that we cannot control which fields are being transmitted, or the packet formatting of the telemetry data. Each transmitter will send a packet once per second, containing the latitude, longitude, and altitude of the transmitter. These packets will be transmitted as audio data and can be decoded into a string in the APRS format.

There will be two telemetry transmitters in use during the flight. One will be in the nosecone of the rocket, and the other will separate with the payload. They will be on slightly different radio frequencies because the timing is not guaranteed, so the signals may overlap.

2.3.2 Receiving Telemetry

Levi Willmeth will manage receiving telemetry and parsing flight data for this project.

We will be using a software TNC called Direwolf to decode the audio tones into strings of text. Because the timing is not guaranteed and the signals may overlap, we will be using one primary receiver per radio signal, with an additional secondary receiver for redundancy. We expect to use four raspberry pi zeros to receive two incoming radio signals, to provide redundancy during signal processing.

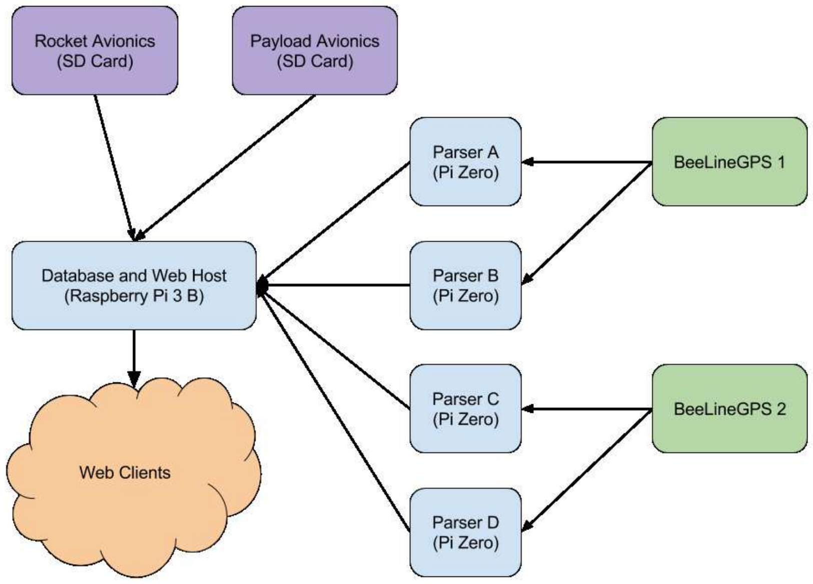

Fig. 2: Flow of information into the parser.

When the parsing computers boot up, they will immediately begin running the Direwolf software that allows them to receive an audio source and parse it into a string of text.[1] They will also begin running a Python script to take the input string from Direwolf, validate it for content, and insert individual fields into a database on the local network. The database will be described in section 2.4 of this document. These programs will be started as services so that they can be stopped, started, or automatically restart themselves as needed.[2]

```
2017−11−28 14:15:56UTC,ESRA−1,131812h3521.00N/07820.00W,496650,8642,446.875,PL
```

Fig. 3: Example APRS packet.

Each incoming radio packet may be from our rocket, or it may be from an unrelated transmitter on the same frequency. This is because we are using uncontrolled analog radio frequencies that may contain information from other, unknown users. We will use Python to parse the string, validate that it contains the correct radio call sign for our payload or rocket, and perform a checksum to test for corruption during transmittal.

If the packet does not pass checksum, contains an invalid call sign, or does not contain the correct data fields, a warning will be added to the parser log and the string will be inserted into a special database table for invalid inputs. This will allow us to inspect errors without discarding any data.

2.3.3 Parsing Data

The parsing program described in 2.3.2 will be configured to either process incoming strings from Direwolf, or to accept a well formatted file. This will allow the same parsing program that listens for radio inputs to parse the data recorded and stored on board the rocket and payload during flight.

After the flight, a human operator will be able to remove the SD card from the rocket and payload avionics flight computers, plug the SD card into a laptop, and point the parsing program at the SD card. The parsing program will scan the directory (or individual file) and determine from the header which device created it. Using the appropriate table name and steps similar to those used while parsing radio inputs, the parsing program will split the strings into individual fields and insert them into the database over the network.

```
timestamp,acc_x1,acc_x2,acc_x3,acc_y1,acc_y2,acc_y3
2017-11-28 14:15:56UTC,27.2,56.7,32.2,234.2,235.7,463.4
2017-11-28 14:15:57UTC,27.1,56.7,32.2,234.2,235.7,463.4
2017-11-28 14:15:58UTC,27.2,56.8,32.2,234.2,235.7,463.4
```

Fig. 4: Example avionics output file.

2.4 Database

Levi Willmeth will set up and manage a MariaDB database for this project.

This project will collect several hundred thousand lines of sensor data, using several different sources. All of this data will need to be sortable and searchable in order to interpret the results of our flight. Furthermore, we want to be able to record and analyze different flights independently. Using a database to store our flight data will allow us to accomplish all of these goals.

Because the rocket avionics records different types of sensors than the payload avionics or the radio telemetry modules, we will use one table per type of data source. That means one table will hold all data received from a BeeLineGPS telemetry module, another table will hold all data imported from the payload avionics, and so on. These tables will also need to relate to each other in order to allow the display program to make appropriate connections between different sensors recorded on different hardware. This will be done using a Flight table with a primary key that will be used as a foreign key on each of the individual data tables.

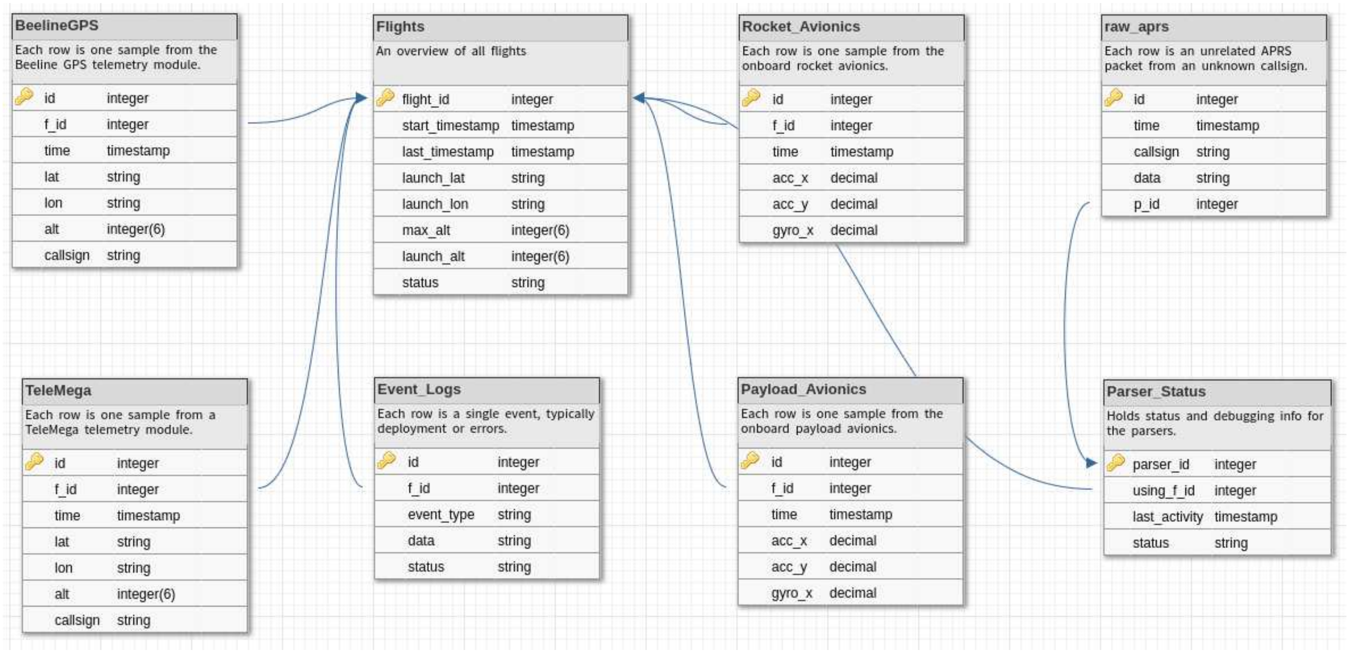

Fig. 5: Early draft of the database schema.

When the telemetry parsing Python script begins, it will check the Flights table of the database to see if there is a flight record with an Active status. This table and field will be used to synchronize the parsers so that records from one radio frequency can be related to records from another frequency at the same time. The Flights table has a primary key called 'flight_id' that is used as a foreign key on other related tables.

To eliminate duplicate rows caused by using redundant receivers, we will use a MySQL query that first checks if a row exists, before inserting a new row. This can be done as a single query: [3]

```
INSERT INTO
        BeelineGPS (f_id, time, lat, lon, alt, callsign)
SELECT
        * FROM (SELECT $cur_time, $cur_flight_id) AS tmp
WHERE NOT EXISTS (
    SELECT id FROM BeelineGPS WHERE time = $cur_time
) LIMIT 1;
```

There may be situations where a given timestamp contains data across multiple tables. For example, the rocket avionics may record some types of data at 100 Hz, and other types of data at only 10 Hz. Some of these records may share an identical timestamp, others may not. That is why we are using multiple tables and a foreign key instead of one larger table and assuming our timestamp is unique.

While graphing this data, we are likely to want to relate data from different tables. For example if we recorded latitude and longitude in the BeelineGPS table, and acceleration in the Rocket_Avionics table, we may want to combine that data to see our acceleration at different points across a map. This is where the database becomes exceptionally useful compared to flat files. We can use a join query to select records from both tables, based on the knowledge that they will have identical f_id's and similar timestamps. The syntax would look something like this:

```
SELECT
          BeelineGPS.lat, BeelineGPS.lon, Rocket_Avionics.acc_x
FROM
          BeelineGPS
INNER JOIN
          Rocket_Avionics
ON
          BeelineGPS.f_id=Rocket_Avionics.f_id
WHERE
          BeelineGPS.flight_id=$cur_flight;
```

In this example the latitude may be sampled at only 1 Hz while acceleration could be sampled at closer to 100 Hz. We could choose to average out many acceleration values to find the best fit at the moment we recorded the latitude, or we could let MySQL find a single value using the Rocket_Avionics timestamp closest to the BeelineGPS timestamp. The syntax for that query could look something like this: [4]

```
SELECT
          BeelineGPS.lat, BeelineGPS.lon, Rocket_Avionics.acc_x
FROM
          BeelineGPS
INNER JOIN
          Rocket_Avionics
ON
          BeelineGPS.f_id=Rocket_Avionics.f_id
WHERE (
          SELECT
                    timestamp
          FROM
                    Rocket_Avionics
          WHERE
                    BeelineGPS.timestamp >= (Rocket_Avionics.timestamp-100)
          AND
                    BeelineGPS.timestamp <= (Rocket_Avionics.timestamp+100)
          ORDER BY
                    abs(BeelineGPS.timestamp - Rocket_Avionics.timestamp)
          LIMIT 1
);
```

2.5 Display Components

2.6 Networking

The ground station will consist of a Raspberry Pi 3, several radio receivers, several Raspberry Pi Zeros, and a battery. The Raspberry Pi zeros will parse data received through the radios before sending it to the database and server hosted on the Raspberry Pi 3. The server will allow users to connect through an ad hoc network and display the graphical flight data on mobile devices.

USB OTG (On-The-Go) will be used to connect the four Pi zeros to the Pi three, which can then assign them IP addresses and power the other Pis. Setting up this connection is a relatively straightforward process that involves editing configuration files on the zeros to give them static IP addresses, and configuring the host Pi to forward network information and assign static IP addresses to each wired connection. [5]

The connected Pis will also serve the graphed flight data to mobile devices around the ground station. This will be accomplished through an ad hoc network. The Pi3 has an integrated wifi module and changing the interface configuration file at /etc/network/interfaces will allow the the Pi to operate in ad hoc mode. [6]

2.6.1 Web hosting

The dynamic graphing display will be hosted by an Apache server hosted on a networked Raspberry Pi 3. Apache's CGI module, the common gateway interface, can be used to serve dynamic content.[7] The flight data graphs generated by the dynamic graphing program will be served as HTML pages to connected mobile devices.

2.6.2 Dynamic graphing

Dynamic Graphing will be managed by Joshua Novak.

The dynamic graphs will be written in JavaScript, using the non-commercial version of CanvasJS. The graphs will be able to be displayed in an HTML page, and will update through queries to a database. These updates will be once a second, though new data may not be displayed if it is not available.

At least the following charts will be displayed in the form of dynamic graphs

- Location of the rocket on a map
- Altitude of the rocket
- Vertical velocity of the rocket
- Orientation of the rocket

The following will be displayed for launches where such data is available

- Temperture
- Velocity
- Acceleration
- Spin
- Barrometric pressure

The first kind of graph that will be displayed is the map. The map will be generated with basic JavaScript. An image of a map around the launch site will be used, with javascript overlaying dots representing the recorded locations of the rocket. The dots will have an on hover effect to display the GPS coordinates of that point, and the timestamp associated with it. Some additional effects may be added to make the graph more readable, such as gradually changing the hue of dots over time, so that newer points appear different from older ones. This will only be used to show the location of the rocket on a map. The points at which particular flight events occur may be represented with a different image, such as a broken circle for separation, or a half circle once the parachute is deployed.

The second kind of graph that will be used is an orientation graph. This will show the relative orientation of the rocket against a compass. The graph will be circular with an arrow overlayed on the graph. The point will be directed toward the bearing of the rocket relative to absolute north. This graph may use CanvasJS or other libararies, but should be able to be easily generated using basic JavaScript functionality. On hover, the graph will display a numerical value for the bearing. This graph will only be used to show the orientation of the rocket.

The final kind of graph that will be used is a basic value vs time graph. This graph will likely be generated using CanvasJS. It will display a maximum number of points relative to the size of the window, so that the graph scales well to different display sizes. These points will be the last several datapoints for the flight. The time will be determined by the timestamp. The relevant value to be graphed opposing time will generally be taken directly from the database query where it is available, and calculated from other data when it is not avaiable but can be calculated with a reasonable degree of accuracy. This will likely be used for all other fields.

REFERENCES

[1] J. Langner, "wb2osz/direwolf," Apr 2017. [Online]. Available: https://github.com/wb2osz/direwolf

[2] "Run a script as a service in rasbian." [Online]. Available: http://www.diegoacuna.me/how-to-run-a-script-as-a-service-in-raspberry-pi-raspbian-jessie/

[3] "Mysql: Insert record if not exists in table." [Online]. Available: https://stackoverflow.com/questions/3164505/mysql-insert-record-if-not-exists-in-table

[4] "Mysql: Select from multiple tables using a foreign key." [Online]. Available: https://stackoverflow.com/questions/24332294/how-to-get-all-data-from-2-tables-using-foreign-key

[5] A. Ellis, "Create a raspberry pi otg network," Apr 2017. [Online]. Available: https://github.com/alexellis/docker-arm/blob/master/OTG.md

[6] "Ad hoc setup in rpi 3." [Online]. Available: https://raspberrypi.stackexchange.com/questions/49660/ad-hoc-setup-in-rpi-3

[7] "Apache tutorial: Dynamic content with cgi," 2017. [Online]. Available: http://httpd.apache.org/docs/current/howto/cgi.html