

Raspberry Pi Companion Computer and Data Collection System for Space Educational Kits

By Brandon Hickey, Bachelor of Science

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of Master of  
Science in the field of Mechanical Engineering

Advisory Committee:

Mingshao Zhang, Chair

Jeffrey Sabby

Keqin Gu

Graduate School

Southern Illinois University Edwardsville

May 2022

## ABSTRACT

By Brandon Hickey

Chairperson: Professor Mingshao Zhang

As part of the Department of Defense's National Defense Education Program, we are tasked with developing educational kits for students to get firsthand experience with space-related system. To expand the potential usage of these kits, we developed a Companion Computer system capable of communicating with our kits while taking environmental data to be reviewed and analyzed after missions. While Companion Computers are common, there are no pre-developed systems available that can perform within our requirements, thus we looked to develop our own system. Using a standard Raspberry Pi microcomputer, we were able to successfully create a data collection system that stores mission information and video data alongside environmental sensor data. With the development of this system, we can expand the capabilities of our educational kits to include data collection that can be used in student-tailored research missions.

## Table of Contents

ABSTRACT .....	ii
LIST OF FIGURES .....	v
CHAPTER I.....	1
INTRODUCTION .....	1
CHAPTER II .....	4
BACKGROUND .....	4
Department of Defense National Defense Education Program.....	4
Navio2.....	4
Ardupilot.....	5
Expanding the Usage of Our Kits.....	6
Existing Technology.....	7
Our Goal.....	8
CHAPTER III .....	10
METHODOLOGIES.....	10
System Requirements .....	10
Hardware Specifications .....	12
Software Specifications .....	13
CHAPTER IV .....	16
IMPLEMENTATION .....	16
Hardware Implementation .....	16
Software Implementation.....	21
CHAPTER V .....	30
Results and Evaluation .....	30
Standalone Testing .....	30
Manual Outdoor Rover Testing.....	35
Waypoint Outdoor Rover Testing .....	44
Additional Sensor Capabilities.....	52
CHAPTER VI.....	55
Future Work.....	55
Try/Except/Else/Finally Implementation.....	55
External Indicator of Program Running .....	55

Container System .....	57
Additional Sensors.....	57
CHAPTER VII.....	59
Conclusion .....	59
CHAPTER IX.....	61
REFERENCES .....	61
APPENDICES .....	63
A: COMPANION COMPUTER CODE .....	63

## LIST OF FIGURES

Figure 1. 3D Printed Avionics Bay .....	3
Figure 2. Navio2 HAT [2] .....	5
Figure 3. Basic Hardware Connection Diagram .....	12
Figure 4. Basic Code Execution Diagram .....	15
Figure 5. QWIIC Connector Example [10] .....	18
Figure 6. Raspberry Pi GPIO Pinout [12] .....	19
Figure 7. Hardware Implementation Diagram .....	20
Figure 8. Main Code Loop Diagram .....	22
Figure 9. Mavlink Initialization Code Diagram .....	24
Figure 10. Armed Loop Initialization Diagram .....	26
Figure 11. Armed Loop Main Code Diagram .....	28
Figure 12. Mavlink Message Parse Diagram .....	29
Figure 13. Rover with Companion Computer .....	30
Figure 14. Indoor Rover CCS811 Data .....	32
Figure 15. Indoor Rover BME280 Data .....	33
Figure 16. Indoor Rover BME280 Data .....	34
Figure 17. Indoor Rover TMP117 Data .....	34
Figure 18. Indoor Rover VL53L1X Data .....	35
Figure 19. Navio2 Path Data .....	37
Figure 20. Combined Path and Video Data .....	38
Figure 21. Outdoor Camera Data .....	38
Figure 22. Outdoor Rover TMP117 Data .....	39
Figure 23. Outdoor Rover BME280 Data .....	39
Figure 24. Outdoor Rover CCS811 Data .....	40
Figure 25. Outdoor Rover Altitude Data .....	42
Figure 26. Outdoor Rover Testing Adjusted Altitude Data .....	43
Figure 27. Time Delay BME280 Testing .....	43
Figure 28. Failed Outdoor Waypoint Testing Map .....	45
Figure 29. GPS Location Drift and Rover Movement .....	46
Figure 30. Successful Outdoor Waypoint Testing Map .....	47
Figure 31. Outdoor Rover Waypoint GPS Satellite Count .....	47
Figure 32. GPS Horizontal Accuracy .....	47
Figure 33. Outdoor Rover TMP117 Data .....	48
Figure 34. Outdoor Rover BME280 Data .....	49
Figure 35. Outdoor Rover Waypoint Air Quality Data .....	50
Figure 36. Outdoor Rover Waypoint Altitude Data .....	51
Figure 37. Outdoor Rover Waypoint Offset Altitude Data .....	51
Figure 38. Outdoor Rover Waypoint Distance Sensor Data .....	52
Figure 39. Indoor ADC Testing .....	53
Figure 40. Indoor Micro Pressure Testing .....	54
Figure 41. Mavlink Disarmed Notification .....	56
Figure 42. Mavlink Armed Notification .....	57

## CHAPTER I

### INTRODUCTION

One of the most fundamental uses of remotely operated vehicles is to gather information. This is even more important when dealing with areas inhospitable to people such as space or different planets. However, the flight controller we have chosen for our project leaves little room for collecting extensive data during missions. This project aims to develop a self-contained data collection system that can gather data during a remote vehicle's operation.

Unmanned vehicles often employ multiple sensors like GPS or IMU to provide information regarding the state of the vehicle. This information can then be used by the vehicle for motion control and obstacle detection and also be displayed to the user in real-time. However, a vehicle could be developed with the goal of gathering information like temperature or humidity levels in various environments. This is not especially useful to the operation of an unmanned vehicle, and with each additional sensor added, the vehicle must take additional time per programming cycle to interface with each sensor while maintain its desired movement.

As part of the Department of Defense's (DoD) National Defense Education Program (NDEP), we are tasked with designing multiple educational kits that will provide a learning platform for students in the fields of space and rocketry. Using a single type of flight control system, we will design each individual kit to have an interchangeable "brain" that can be inserted into different kits with minor changes to its already developed configuration. While these kits are meant to provide students with a basic understanding of key principles and develop experience with the platform, we have begun to investigate potential ways to expand the uses of our kits from just

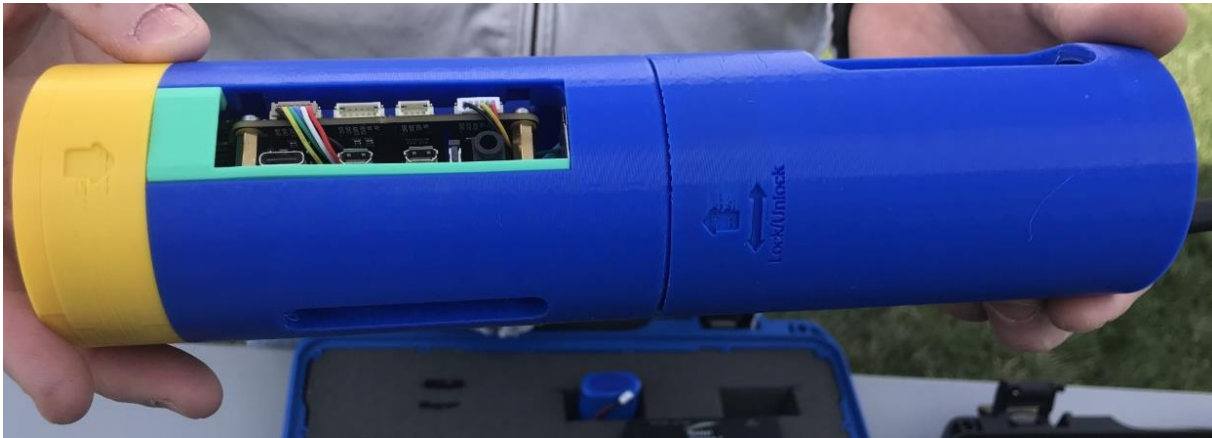
becoming familiar to the design and operation to also include possible real-world applications such as data collection.

To accomplish this goal, we will develop a data collection system that can be attached to our already existing flight control system and provide expanded data collection capabilities without heavily impacting the performance of our vehicle. The data collection system will also create easily accessible data logs including the expanded sensors and vehicle information such as location and orientation for later review and analysis, as well as provide real-time data to the user through a common ground control station.

The result of this project is a Companion Computer that can read multiple sensors for environments including ground, air and underwater with sensors that measure features such as temperature, pressure, humidity, air quality, and water quality. We have also included the capability to record video data using a Raspberry Pi based camera. Our data collection system is compatible with our existing flight controller and is therefore useable in all ten of our selected vehicle types.

As part of the Program, I have aided in the development of the Navio2 as the main control system, which can be used interchangeably between all current and future kits. This includes installation, configuration, and modification of the existing Navio2 systems to satisfy our performance requirements. In addition,, I developed documentation and video instructions on performing all required adjustments to the Navio2 for each kit. This information would then be used by project collaborators from University of Illinois Urbana-Champaign in the final

development of instructional videos regarding the assembly and usage of each kit. As part of our rocketry kits, I have designed the avionics bay, *Figure 1.*, that holds all rocketry electronics including our flight controller which acts as a data logger and source of telemetry data to our ground control station.



*Figure 1. 3D Printed Avionics Bay*



## CHAPTER II

### BACKGROUND

#### Department of Defense National Defense Education Program

This project is conducted as part of a research grant by the Department of Defense (DoD) through the National Defense Education Program (NDEP) with the objective of developing STEM education and outreach initiatives for Highschool and Undergraduate students with a focus on the fields of rocketry and space [1]. Our focus is on the development of educational kits that will provide students with the tools and instruction necessary to create vehicles associated with space and exploration. The expected kits include a variety of vehicles such as rockets, drones, rovers, and submarines. As part of this program, we will use a controller that can be employed by any kit with minor changes required to existing controller configuration. To satisfy this requirement, we have chosen to use the Navio2 Autopilot (Hardware Attached on Top) HAT and a Raspberry Pi 4 microcomputer as our flight controller.

#### Navio2

The Navio2 shown in *Figure 2* is a preconfigured HAT, which is an add-on board that can be mounted to a Raspberry Pi to expand its functionality. The Navio2 is designed to be a flight controller and contains additional sensors than the Raspberry Pi. These include a Global Navigation Satellite System (GNSS) Receiver for satellite tracking, a high-resolution barometer for altitude, dual inertia measurement units (IMU) that provide accelerometer, gyroscope, and magnetometer readings, and additional input and output connections. To operate the Navio2, the Raspberry Pi is required to run an autopilot software called Ardupilot.



*Figure 2. Navio2 HAT [2]*

## Ardupilot

Ardupilot is an open-source autopilot software that is commonly used throughout the drone and remote-control community [3]. It is configured to work with six different vehicles including copter, plane, rover, and submarine. Ardupilot is run on the Raspberry Pi along with a Robot Operating System (ROS) program known as MAVROS [4] which provides communication for autopilot systems using a communication protocol known as MAVLink [5]. MAVLink provides the ability to send and receiving data from the flight controller as well as sending commands from our ground control station to our vehicle.

A ground control station, or ground control software, (GCS) is the platform such as a phone or

computer application that is connected to the flight controller through wireless connection such as Wi-Fi or radio telemetry. A ground control station provides numerous benefits. We can observe real-time data of our flight controller to include features such as altitude, speed, heading, GPS location, and battery life. Additionally, we can configure missions and waypoints for autonomous movement and control the speed and distance of our vehicle with respect to the location of our ground control station.

### Expanding the Usage of Our Kits

Using just the Navio2 and Raspberry Pi, we can fully control our various kits as well as look at real-time data being transmitted back to our GCS. However, we lack the ability to add additional sensors to our flight controller. The Navio2 uses almost all available General Purpose Input Output (GPIO) pins of the Raspberry Pi, leaving only three available. Among these three, two of them are Universal Asynchronous Receiver-Transmitter (UART) pins, leaving only a single GPIO pin [2]. This means we are unable to connect sensors that require either Inter-Integrated Circuit (I2C) or Serial Peripheral Interface (SPI) communication protocol. While the Navio2 does have the capability of reading two analog pins, these pins are used in the power monitor capabilities of Ardupilot to provide the user with information regarding battery voltage and system current draw. This information is then used by the GCS to provide an estimated remaining flight time and acts as a trigger for warnings based on battery voltage level.

To expand the data acquisition capability of our system, we chose to develop a Companion Computer that would perform sensor measurements with minimal impact on vehicle performance. “A Companion Computer is a device that travels on-board the vehicle and controls/communicates with the autopilot over a low-latency link. Apps running on a Companion

Computer can perform computationally intensive or time-sensitive tasks and add much greater intelligence than is provided by the autopilot alone.” [6] What this means is that we can assign tasks that are not crucial to the operation of our vehicle to the Companion Computer instead. In our case, we will use the Companion Computer to measure a variety of environmental states and log this information along with the vehicle’s current location and orientation. Unlike many Companion Computers, we will not be using the computer to control the vehicle. Instead, we will be using it to collect data from both its own sensors and the broadcasted data from the vehicle and log both sets of data to the same file. This will provide the user with location and time markers for sensor readings, thereby improving data collection results and accuracy.

### Existing Technology

While the stem kits do not require these extra sensors, the development of this capability will allow the kits to be used for more than just learning and understanding in space and rocketry. Once a student is familiar with the principles and operation of a kit, they can use the kit and the Companion Computer to collect data about their vehicle’s surroundings, as if the vehicle is on a completely different planet. This Companion Computer provides students with another valuable experience in the potential applications of the systems they are learning about.

While there is a large amount of documentation regarding the development of Companion Computers, most are based off a similar and easily acquirable hardware system such as Raspberry Pi, Nvidia, or Arduino. They are then customized by the user for their desired performance. However, the available documentation for the development of data collection Companion Computer operating with a Navio2 flight controller is rather limited.

For example, the Nvidia Jetson TX2 Companion Computer is a powerful system that can perform complex functions such as deep learning applications [7]. However, it is a bulkier option, requiring a 12-Volt power supply with a minimum of 7.5 Watts while our vehicle will be operating on a 5-Volt regulated supply from a battery pack with voltage as low as 7.4 Volts. It also does not contain any breakout pins for ease of prototype and implement sensors. At a base cost of almost \$200, it is an expensive piece of equipment to use as part of educational kits while there are cheaper alternatives that can accomplish our goals.

There are a few publications detailing the use of a Navio2 autopilot alongside other computer systems, but most often these Companion Computers are operating as an intelligent controller of the vehicle, rather than a data gathering system. As part of research in autonomous navigation for aerial vehicles [8], one research paper discusses how to the Navio2 in combination with a Raspberry Pi and camera. As the drone moves, its onboard camera takes photos and the Raspberry Pi then determines how the drone should navigate safely. In this case, the Raspberry Pi is determining the drones' movement and then communicates these commands back to the Navio2 through the Mavlink communication protocol. This differs from our goal where the Raspberry Pi is not acting as a controller, but rather as its own system that collect and exchanges information with the flight controller.

### Our Goal

Constraints for this system include the Companion Computer will have a low impact on the flight controller latency due to additional sensor modules, be useable by different kits developed through the NDEP project, and store data from both the flight controller and the Companion Computer together for viewing. The issue with using only the Navio2 is the lack of sensors. The

sensors onboard the Navio2 are limited to flight related ones: GPS, Barometer, and IMUs. This provides little capability to collect additional data about the environment. Depending on how the kit is used, it can become a sensor collection platform.

The aim of this project is to develop a Companion Computer system to interface with a Navio2 flight controller and use additional sensors to collect environmental data throughout the vehicles operation and to log this information along with vehicle information either in real-time or after mission completion. We will determine the viability of this project by how easily the data collection system is able to interface with a variety of sensors without requiring constant program modifications.

## CHAPTER III

### METHODOLOGIES

#### System Requirements

Given the desired capability of our Companion Computer, there are several features we aimed to include in our system. The first and foremost is that our Companion Computer must be able to interact with the Navio2 flight controller. The Navio2 is use in all our kits, and therefore our Companion Computer must be compatible with it.

We are looking for a Companion Computer that required limited physical connections to the flight controller. Since all sensor measurements are done through the Companion Computer, we only need a power supply and an ability to communicate with the Navio2 and the ground control. With a single communication line we are able to interact with the Navio2, and take advantage of the features of the Mavlink communication protocol which can send and receive information from the GCS through the Navio2 telemetry link including status texts to the GCS to alert the user of the Companion Computer's status.

Along with the capability of performing all sensor measurements on just the Companion Computer, we also want to log and compare the system time from both the Companion Computer and the flight controller. This will ensure that environmental data collected will correspond to the correct location data provided by the flight controller.

Given that the STEM kits we are developing are made to cover several different environments including air, land, and water, it will require a Companion Computer that can be used in each location without complex changes in between missions. For example, if we want to measure an

environments pressure, it would require different type of pressure sensor depending on whether the kit was a drone flying through the air, or a submarine traveling beneath the water. In this case, the user should be able to change sensors in the companion system without any change in programs. Furthermore, it would be even more beneficial if we can change our sensors without shutting down the power (Hot-swappable).

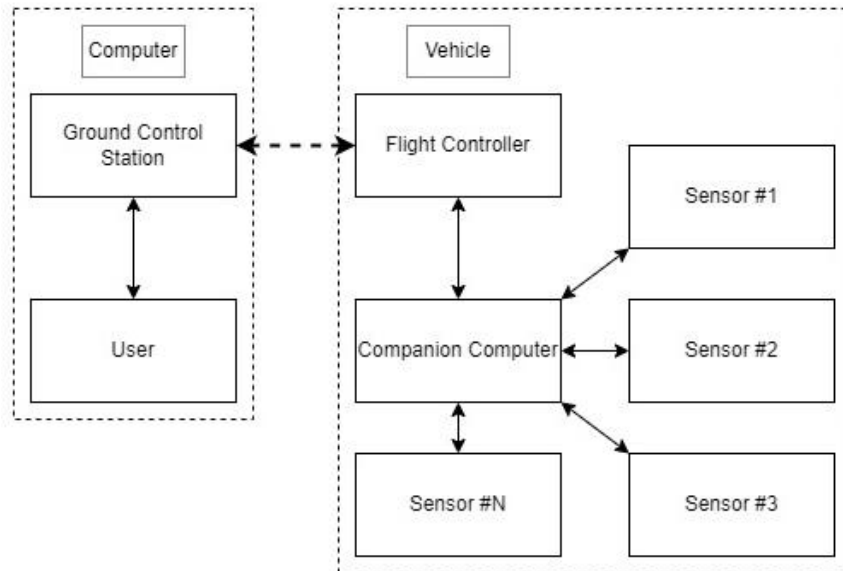
Since we want to log both the sensor data collected from the equipped sensors and data such as vehicle location and orientation, our system must be capable of sending and receiving data from both the physically connected flight controller and the wirelessly connected ground control station. Being connected to the ground control station means that our Companion Computer can send information back to the user in real-time such as alarms regarding water depth or obstacles in front of the vehicle.

Finally, all data from the Companion Computer and desired information from the flight controller should be logged in a single location for later review and analysis. The sensor and vehicle data should be reported in a format that is easy to analyze using software such as Excel.



## Hardware Specifications

The basic setup of our system will be like what is shown in *Figure 3* where the two main components are the GCS and the vehicle. For our computer, the focus is the ground control station software. The ground control station should be useable in remote locations without internet access, be compatible with the Ardupilot software and be able to connect with the Navio2 flight controller using a radio telemetry system.



*Figure 3. Basic Hardware Connection Diagram*

For our vehicle, we have two main components: Flight Controller, and Companion Computer. The Flight Controller will be a Navio2 Autopilot HAT mounted on a Raspberry Pi 4. This is the standard autopilot system used throughout all our kits. The required components for the flight controller will vary from kit to kit. However, the basic requirements are similar. The flight controller requires a sufficient power source capable of supporting the flight controller, all required motors and electrical components, as well as the Companion Computer and its attached sensors. The flight controller also requires a telemetry radio, which will transmit vehicle data

including location, speed, and orientation back to the user for viewing through the ground control station.

### Software Specifications

We have broken down the expected operation of our software into several key areas: scan for currently connected sensors, create connection with Navio2, and then record data from both sensors and Navio2 to a single data file while vehicle is armed. The program will be started by one of three methods: physically started by the user, electrically started by a signal sent from the flight controller, or software controlled by the user. In the physical starting method, we will have a switch or button connected to the Companion Computer. This will provide an electrical signal that can be interpreted to begin the program. In the electrical method, the flight controller will send a starting signal to the Companion Computer which will then begin the program. In the last method, we will connect to the Companion Computer using Secure Shell (SSH) and begin the program. Among those three methods, the most likely solution is the combination of physical and electrical. Having to manually connect to the Companion Computer to begin each program is tedious and unnecessary. For the combination method, the Companion Computer will not operate if the vehicle is not operating, and indication signals will be sent to the user regarding the program running status.

Once the program has begun, it will first start by looking for all available sensors. This will be mainly for checking the I2C connect sensors. Since we know each I2C sensor has an address specified during manufacturing, we can check for active addresses and compare with our list of acceptable sensors to confirm which are connected.

Based on the list of active connected sensors, we can start our log file with the data types and names of the sensors we will be measuring. This will allow us to conserve space of our log files and the system will not attempt to scan inactive sensors, resulting in faster data collection.

Before the system begin to read data from the flight controller and sensors, we must set specific conditions to identify when we actually want to log data. The most important indicator will be whether or not the flight controller is armed. Arming the flight controller indicate that it is operating and therefore we should begin to collect data. While active, the Companion Computer will interact with each connected sensor one at a time and read then store its available data. After all sensors have been read, the Companion Computer will then ask the flight controller its current location and orientation data and store that alongside the collected sensor data. After both portions are complete, the entire set of data will be logged and then this process will repeat. Since the armed signal can be confirmed through the Mavlink protocol, we are able to detect when the mission ends, marking the end of our data logging. At this point, we can either finish the missions, end the program and begin to analyze data or we can arm the flight controller again, starting an entirely new data log to separate from previous missions.

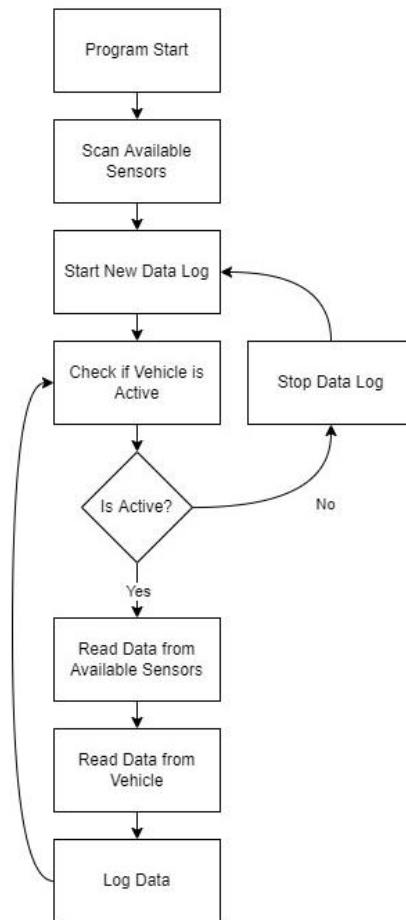


Figure 4. Basic Code Execution Diagram

## CHAPTER IV

### IMPLEMENTATION

#### Hardware Implementation

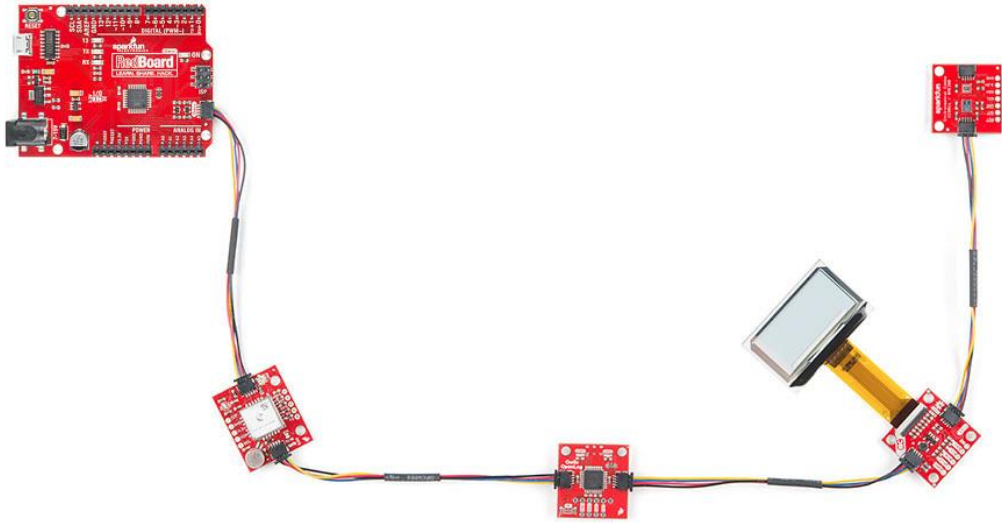
For the physical layout of our system, only a single connection is required between the Navio2 and our Companion Computer which deals with communication. Using the Ardupilot firmware, we can designate one of the Navio2's Universal Serial Bus (USB) ports as a serial communication line to transmit and receive data. However, to connect this USB port to the Companion Computer, we need to use a USB to UART adapter that would allow us to take the now converted UART connections and connect it to our Companion Computer at GPIO14 and GPIO15 using the Raspberry Pi's header.

The last external connection required for the Companion Computer is power and ground. We must maintain a common ground between both components, and they should both work on the same battery source. To do so, we used two battery eliminator circuits (BEC) [9] which will regulate the voltage of the battery pack to a stable 5V supply. While we are able to use a single BEC to power both sections, this increases the load on the BEC. Given additional components for operation of different kits, this might result in the risk of not being able to supply enough power to our Companion Computer and our flight controller.

For our Companion Computer, there are three main components we are dealing with: analog sensors, I2C sensors, and a camera. Beginning with the simplest connection, we have the camera. The camera is a standard Raspberry Pi v2 camera which was selected for its size, and the pre-existing libraries for interacting with the camera. The connection is done through the Camera

Serial Interface (CSI) Port on the Raspberry Pi with a 15cm ribbon cable, which provides us with some flexibility in terms of camera placement and viewing angle.

We have selected I2C sensors for the reason that a single I2C line allows connection to multiple sensors. The only potential conflict between I2C sensors would be those that share a same address. However, we have not encountered this issue with any of our currently selected sensors. Another benefit to our selected I2C sensors is that all are built by Sparkfun using a prototyping method they call “QWIIC” [10] as shown in *Figure 5* this means each sensor is equipped with two 4-pin Japan Solderless Terminal (JST) connectors, which allow you to chain multiple sensors together quickly and easily to a single I2C line. This reduces complexity in adjusting our Companion Computer between kits. All we need to do is quickly disconnect a sensor from the chain or add a new one on to the end with no additional soldering or components necessary. Normally, we would use GPIO2 and GPIO3 pins of the Raspberry Pi to interact with I2C devices. However, we had to switch to pins GPIO23 and GPIO24 due to conflicts with our attached camera, which we will discuss in the software implementation section of this report.



*Figure 5. QWIIC Connector Example [10]*

For analog sensors, some additional components are required. The Raspberry Pi does not have the capability of reading analog signals, meaning that we had to use a system capable of interpreting analog signals into a digital format. This is commonly known as Analog-to-Digital Converter (ADC) [11], which takes an analog input and converts it to a digital output. However, in a typical ADC, you are required to have amount of digital input pins that equal to the resolution. For example, if we attempted to develop our own ADC, we would require 8 available input pins on our Raspberry Pi along with 8 Operation Amplifiers (Op Amp) to then develop a digital encoder to provide an 8-bit resolution analog value. This limits the number of available pins you are left with and increases wiring complexity. To get around this, we use an ADC that can be interacted with the I2C communication protocol. This means we do not have to expend more GPIO pins as we are using the same communication protocol as our other sensors. Our chosen ADC also has 4 channels, meaning we can use 4 different analog sensors with only one I2C address, further reducing complexity.





















Pin#	NAME		NAME	Pin#
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I2C)		DC Power 5v	04
05	GPIO03 (SCL1 , I2C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)		(I2C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

Figure 6. Raspberry Pi GPIO Pinout [12]



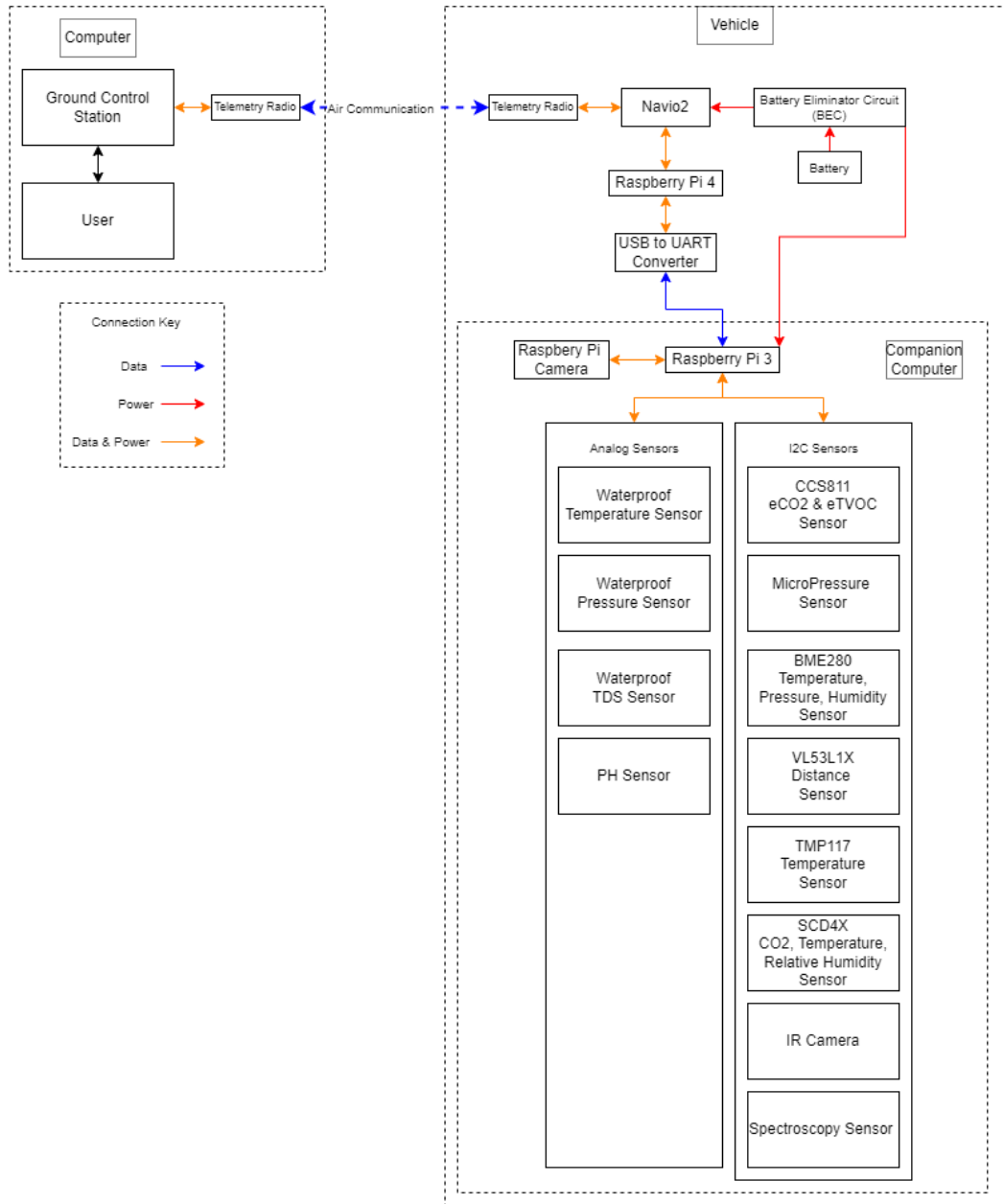


Figure 7. Hardware Implementation Diagram

## Software Implementation

There are three main components of our system, which are the onboard sensors, camera, and Mavlink communication. In this section, we will explain the structure of the main software, followed by explanations for the three main components. Each of these components contains their own challenges to the system's operation.

First, we have the main section of software shown in *Figure 8*. To ease the field operation, we felt it was necessary to design a method to start and stop the program without using a display and keyboard. The resulting programming choice was to design a service [13] that would start on boot when power is connected to the Raspberry Pi. This solves the issue of starting the program without using display and keyboard.

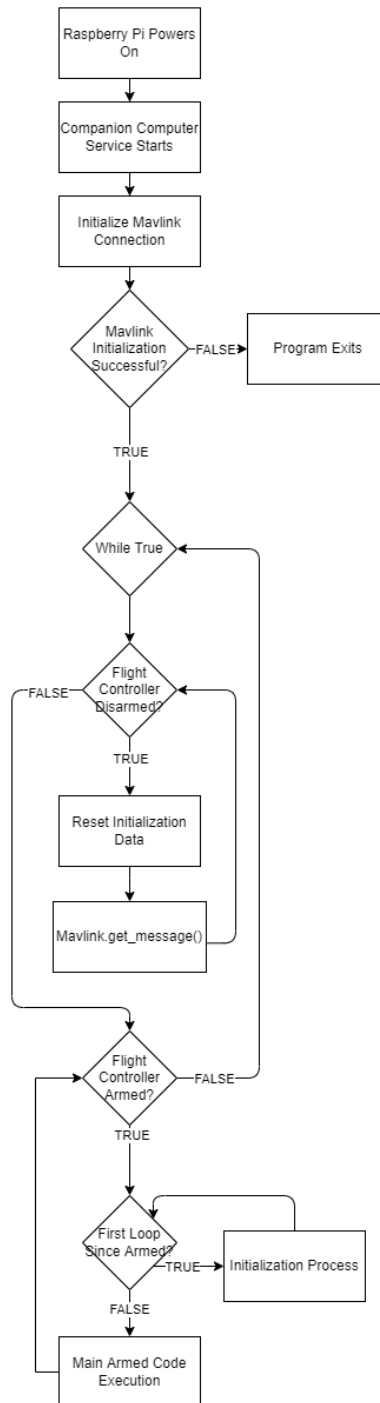
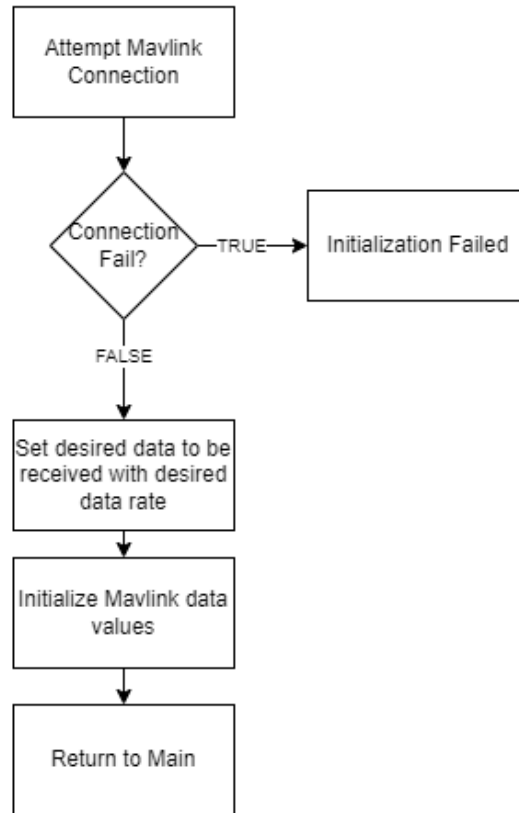


Figure 8. Main Code Loop Diagram

After the program started, we perform a first-time initialization attempt to our Mavlink communication. Using a package known as “pymavlink” [14] we can interface with our Navio2 by sending and receiving Mavlink protocol commands. This is how we will be collecting data

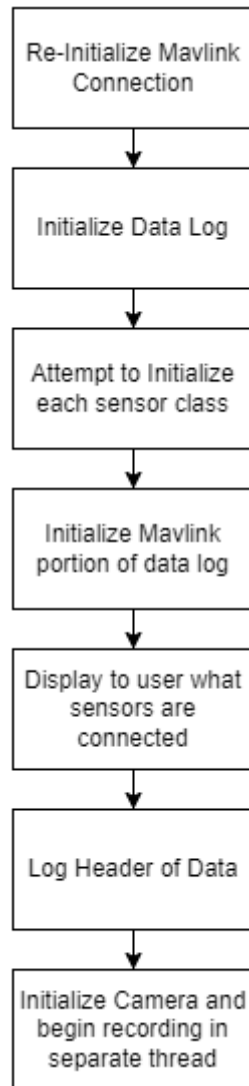
from our flight controller and send information back to the user to be displayed through the ground control software. The pymavlink package contains a class known as “mavutil”, which is used to recognize Companion Computer as a Mavlink device by both the GCS and Navio2. To successfully initialize this connection, we have to specify our desired baud rate and where our connection is. In our case, we use the UART pins designated on the Raspberry Pi pin header, which are GPIO14 and GPIO15 and our selected baud rate is 115,200. However, we also must make changes to the Navio2 configuration to enable the USB port we are connecting to and have it initialized as a serial communication port for Ardupilot at the same baud rate as our Companion Computer. To do this we had to make changes to the Navio2 using both the terminal command [2] and our ground control station [3]. This is because while we can initialize the serial port, we are not able to change the baud rate. Instead, we needed to connect to our Navio2 from ground control station and adjust the baud rate parameter for our selected serial port. Once we confirm connection to the Navio2, we can send a request to the MAVLink system to have the Navio2 send specific packets of information once a second containing our state information. We choose this data rate to prevent a queue of data from building up while waiting to be read by our Companion Computer.



*Figure 9. Mavlink Initialization Code Diagram*

After successfully initializing the Mavlink connection, we enter our main loop and start the first of our two main loops, the Disarmed Loop. In the Disarmed Loop, we are not taking reading from any of the available sensors and instead we are going to monitor messages sent from the Navio2 until we detect a change in the arming status which signifies if our vehicle is set to armed with motors enabled. To check arming status, we read the “HEARTBEAT” message, and focus on the data corresponding to the MAV\_MODE\_FLAG [5]. Depending on the value of the flag, we can determine whether our system is armed or not. If the flag is not armed, then the system continues to read messages until that status changes. Once we detect the arming flag, we exit the Disarm Loop and enter the Armed Loop sequence.

In our Armed Loop, we first begin with a list of first-time initialization tasks, which will occur every time the system is armed. The purpose is to scan the available sensors and detect which of those are active. Having this planned actions with each new mission allows us to quickly changes sensors without adjusting code or powering down our system. In our initialization, we first restart our Mavlink Connection to clear existing data from previous missions. Next, we test each possible sensor by attempting to initialize them. Using a “try” and “except” statement during the initialization of the sensor, we attempt to read data from the chosen sensor. If it produces an error such as invalid address, then we set sensor as inactive, which means even when we ask for a reading, the program will know that no sensor is attached, so it will skip the attempted reading. It also prevents the data from that sensor being logged to our data file, which will result in a large number of blank spaces. Once we finished initializing all of our sensors, the program will display the list of data the mission will be recording to the terminal and then log the header information as a comma separated values (CSV) in a text file. The last step is to begin a separate thread that will handle our camera recording along with handling what data is being recorded by the camera at a given time.

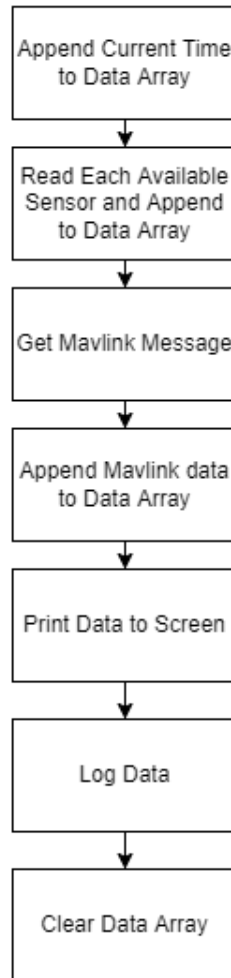


*Figure 10. Armed Loop Initialization Diagram*

Once we finish the initialization step, we begin the main portion of our Armed Loop: collecting data. At the start of each loop, we first store the time, which will be used to as a time stamp within the data log, as well as detect whether the data we are receiving from the flight controller is out-of-date. We will discuss this in the troubleshooting section. After gathering the current time, we will go through each individual sensor and read the available data and then store it in the next column of our data log. If a sensor encounters errors or begins to malfunction, which would cause an error during its reading, our program will ignore that data and list a blank space

instead, allowing our system to continue even when a sensor fails. Once each sensor has been read, the program will then read the next incoming message from the Navio2. In this portion, the name and contents of the message decide what our program will do. If the message is a “HEARTBEAT”, this will be used to check whether the system is now armed or disarmed, and furthermore decide whether we will exit the Armed Loop and begin the Disarmed Loop. If the message is “GLOBAL\_POSITION\_INT”, this message contains the current state of the flight controller. Inside this message is the latitude, longitude, altitude, heading, and velocity of our flight controller. All of this information will be stored alongside the rest of our data. After both the sensors are read and the incoming MAVLink message is parsed, the program will display the data acquired in this current iteration, then log the data as a single line in our data log, which can be viewed as a comma separated list in software such as Excel.





*Figure 11. Armed Loop Main Code Diagram*

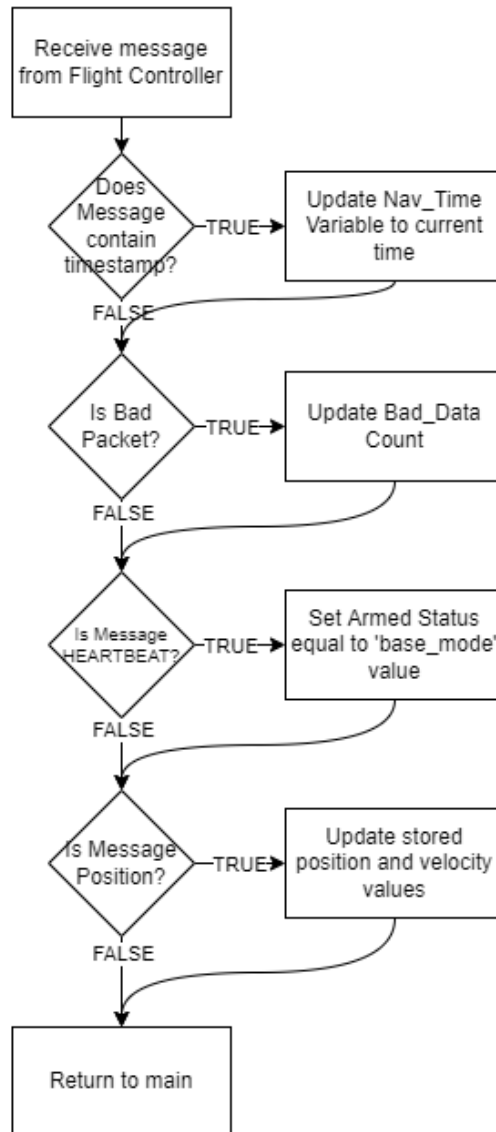


Figure 12. Mavlink Message Parse Diagram

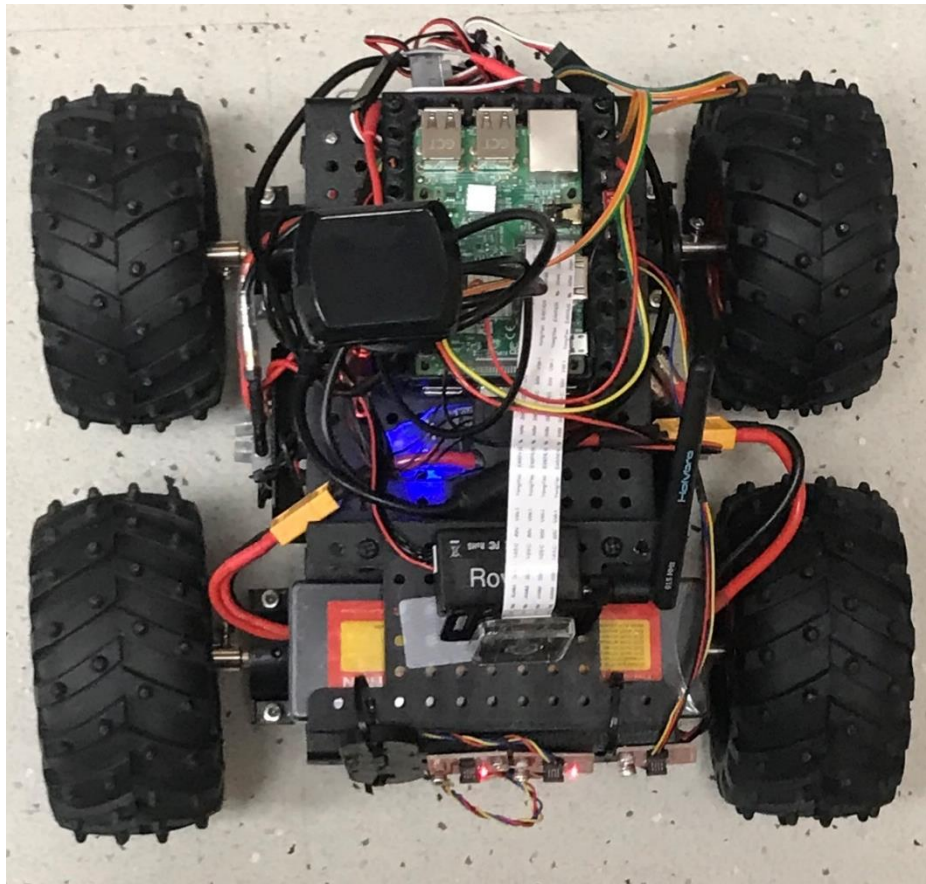
When the program detects that the system has been disarmed, it will end the video recording, and switch back to the Disarmed Loop, resulting in the end of the current data file for both data and video. This way, once we arm the system again, a new data log and video file will be created, which can be used as a mechanism to separate files from multiple missions.

## CHAPTER V

### Results and Evaluation

#### Standalone Testing

For testing, we decided to go with our land-based rover kit, which can finish multiple missions in one testing session without worrying about battery or environmental factors. Since we wanted to test in a safe location inside, we could not use kits such as the drone or submarine. Instead, we aimed at using the rover, activating our program, and then driving it manually around indoor environments while collecting sensor data.

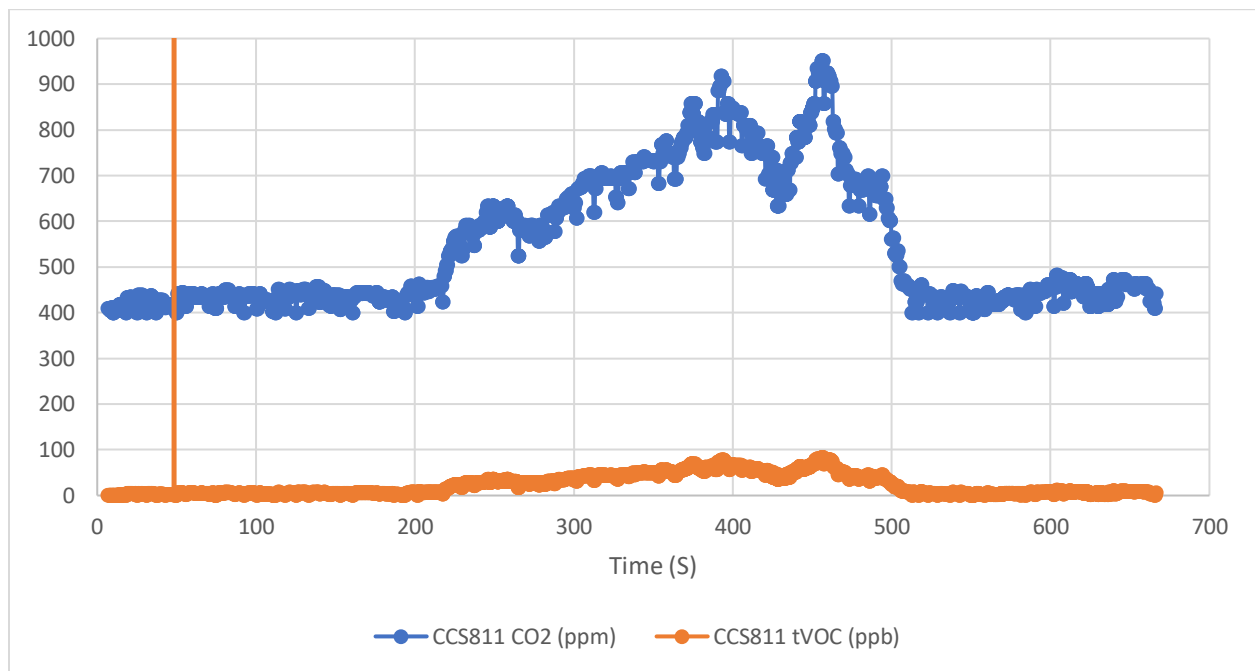


*Figure 13. Rover with Companion Computer*

During testing, the data collection of the Companion Computer and the performance of the rover were both operating correctly without any major issues. However, there is a small issue in terms of activating the program service for the Companion Computer. In normal operations, we would turn on the rover, then the Companion Computer, and the program would start in the disarmed mode and after we armed the rover using a ground control station, then the Companion Computer would arm and begin collecting data. During testing, after arming the rover, an error while calling an object would appear and halt the data acquisition from the sensor package. It is currently unknown what is causing this error, but we are able to circumvent it by turning on the rover, arming it, then turning on the Companion Computer. This immediately skips the disarmed waiting loop and starts collecting data.

After the mission, we were able to retrieve the data log from the Companion Computer and inspect the results to make sure the connected sensors were operating correctly. Given that we are in the indoor environment, we ignored all stored Navio2 data given that part of the state estimation is based on GPS data which is unavailable indoor. We can confirm Navio2 operation using our GCS and viewing the stored telemetry data. Looking through all other sensor data, we confirmed that all connected sensors were functioning. We did notice that at the start of the sensor readings there are two spikes in readings from the “total Volatile Organic Compounds” (tVOC) reading of our CCS811 Gas Sensor, and the altitude measurement of our BME280 sensor. Shown in *Figure 14*, we can see that the tVOC reading spikes to a value of 40,606. We predict this is likely an error with our logic and thus we will need to look for potential solutions that may include comparing the current reading to a moving average to eliminate spikes. The second sensor blip occurs on our first reading of the BME280 sensor, which we have noticed is

constantly present on the first reading of this sensor. With this in mind, we can safely choose to ignore this first data point.



*Figure 14. Indoor Rover CCS811 Data*

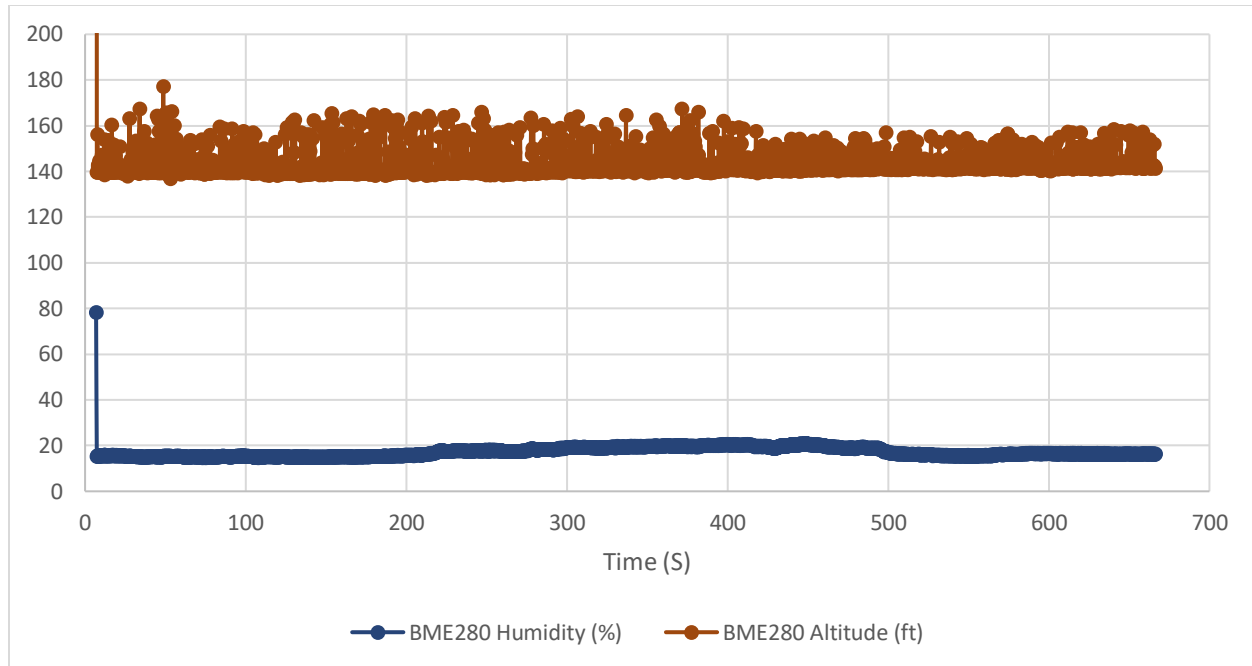


Figure 15. Indoor Rover BME280 Data

Looking at the performance of our other sensors, we can compare the results of the same data measured from different sensors. For example, compare our BME280 temperature data in *Figure 16* and our TMP117 temperature result shown in *Figure 17*, we see a much noisier reading with a higher average temperature from our BME280 sensor when compared to the TMP117. This is because the BME280 sensor is mounted on the same board alongside the CCS811 sensor which requires a small heating element to accurately measure CO<sub>2</sub> and tVOC. This flaw has been noted by Sparkfun that “The BME280 is unable to compensate for the heat produced by the CCS811. This can cause the displayed temperature to be up to 15° higher than the actual ambient temperature” [15]. Other users of the board have noted this heat to result in a higher than actual temperature reading, which we can compare with the TMP117 which is listed as more accurate stand-alone temperature sensor.

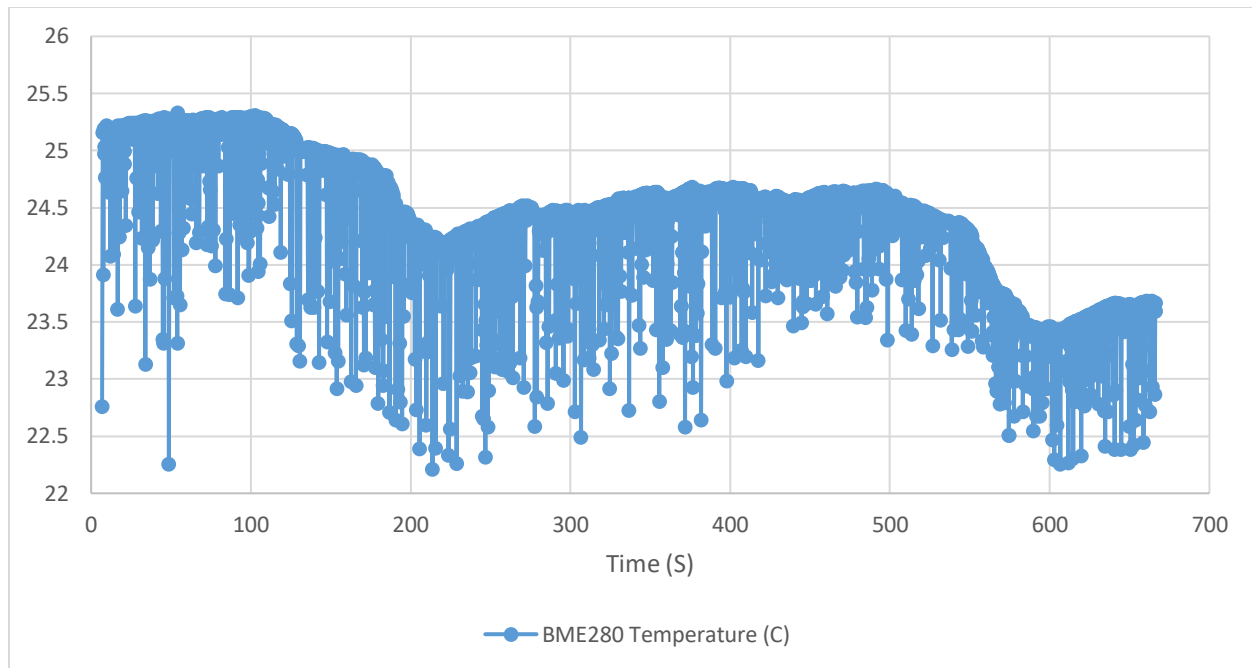


Figure 16. Indoor Rover BME280 Data

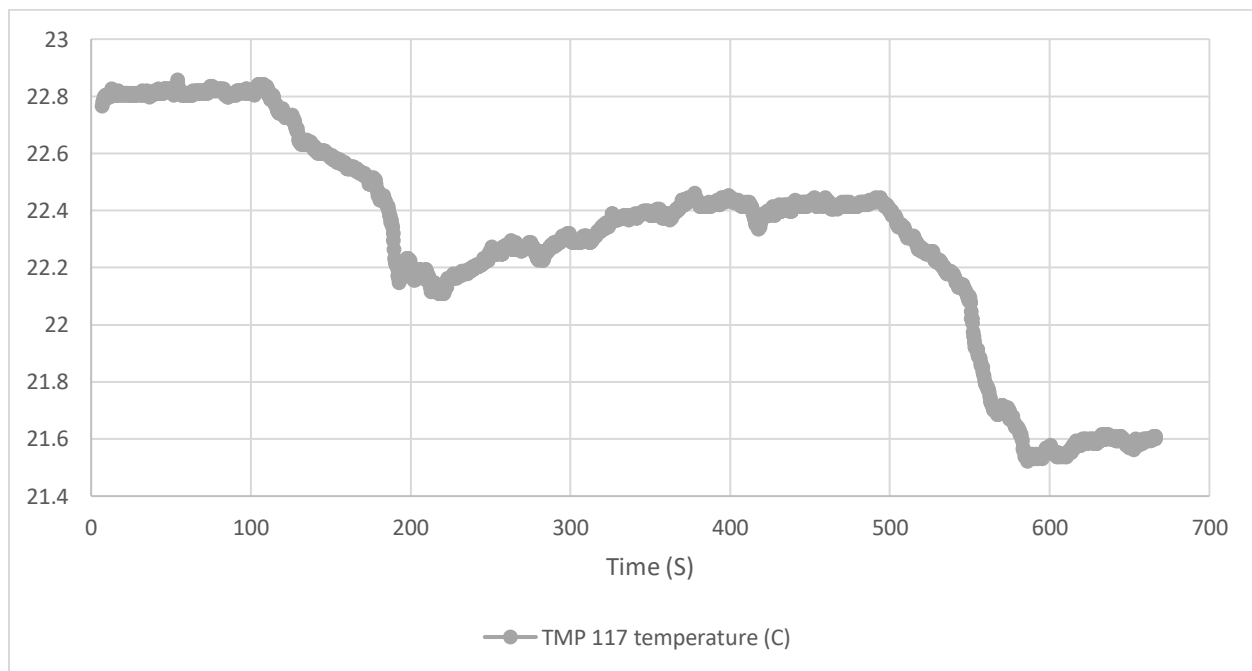
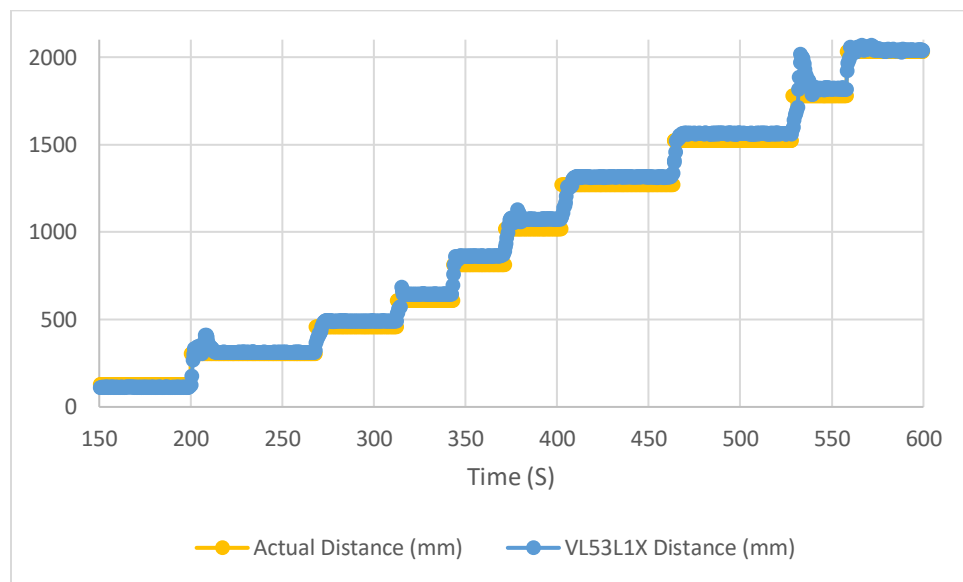


Figure 17. Indoor Rover TMP117 Data

The last sensor we examine in the indoor testing is the VL53L1X Distance sensor. This measures the distance to the nearest object in front of the rover up to roughly 2 meters. These readings can

be used to send commands to the Navio2 to halt movement when an obstacle comes too close to the vehicle. During testing, we started at a distance of roughly 5 inches from a wall and began slowly reversing away from the wall to various distances to measure the accuracy of your sensor. What we notice is that as we are moving further away from the wall, we had an increasing difference between measured and actual distance, as well as increasing spikes at the beginning of each movement. This is due to the rover movement as we reverse and needing to adjust the rover orientation to face directly to the wall after reversing. We can note from the data in *Figure 18* that distances much further away than two meters will show a reading of over 2000 mm and thus should be ignored as they simply represent a long stretch of obstacle-free space.



*Figure 18. Indoor Rover VL53L1X Data*

### Manual Outdoor Rover Testing

Our next testing is to compare the performance of our sensor package and rover operation in an outdoor environment. We chose to operate the rover on paved surfaces in direct view of our ground control station and operator. Similar to the indoor testing, we will power on and arm our rover before turning on our sensor package unit. This will avoid the arming error and allow us to



immediately begin collecting data. As this will be our first outdoor test, we will only operate the vehicle manually using a remote control.

In terms of the rover operation, there are minor issues regarding the camera data. What we noticed when playing back the video feed is that even on paved surfaces the rover produces a large amount of vibration that is visible on the camera. The camera view itself provides a wide enough field of view to see any obstacles that would enter in the path of our rover. However, since this is only a video played back after the end of the mission, it is not as helpful in the real-time control of applications such as submarines, in which we cannot get a line-of-sight view of the vehicle and its surroundings. In terms of reducing vibration, we can potentially include better wheel suspension and/or a different mounting method for the camera.

Looking at the *Figure 19* which shows the recorded path of our rover throughout the mission, we also notice that the position data only started to be known once we had established enough satellites connections for an accurate position estimation. This is due to the time required to update the GPS information of the rover. Even without GPS signal, the rover is able to be successfully operated manually without position information. In the case of *Figure 19*, we see that our only path data is shown in red, which represents manual operation. The white dashed line indicates the direction towards where the system last believe it armed from, which is further away than the actual location due to the information from the GPS slowly reacquiring true position.



*Figure 19. Navio2 Path Data*

Using the stored video feed and the stored path data from the Navio2, we can set both sources side-by-side and play them at the same time to give users a video feed that represents where they are on the map, refer to *Figure 20*. In addition, we have also included a small functionality within the camera module that allows us to display our available sensor data at the top of the camera screen and to cycle through all available data one at a time so that users can see location-specific data readings.



Figure 20. Combined Path and Video Data



Figure 21. Outdoor Camera Data

With the data from the sensor package, we can once again compare the performance of the different sensors. We can also compare the available Navio2 data to other sensor sources.

When comparing the temperature, we see that there is still the same volatility in the BME280 sensor that expected due to its proximity to the CCS811 sensor. The temperature difference between the TMP117 sensor shown in *Figure 22* and the BME280 sensor shown in *Figure*

23remains closely the same. This difference is likely due to the self-heating flaw noted regarding the BME280/CCS811 sensor board.

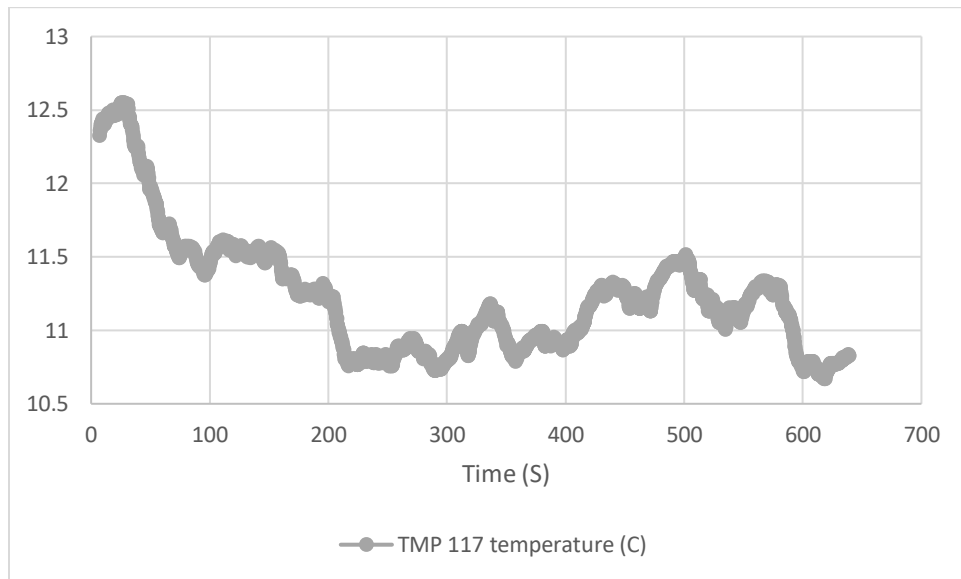


Figure 22. Outdoor Rover TMP117 Data

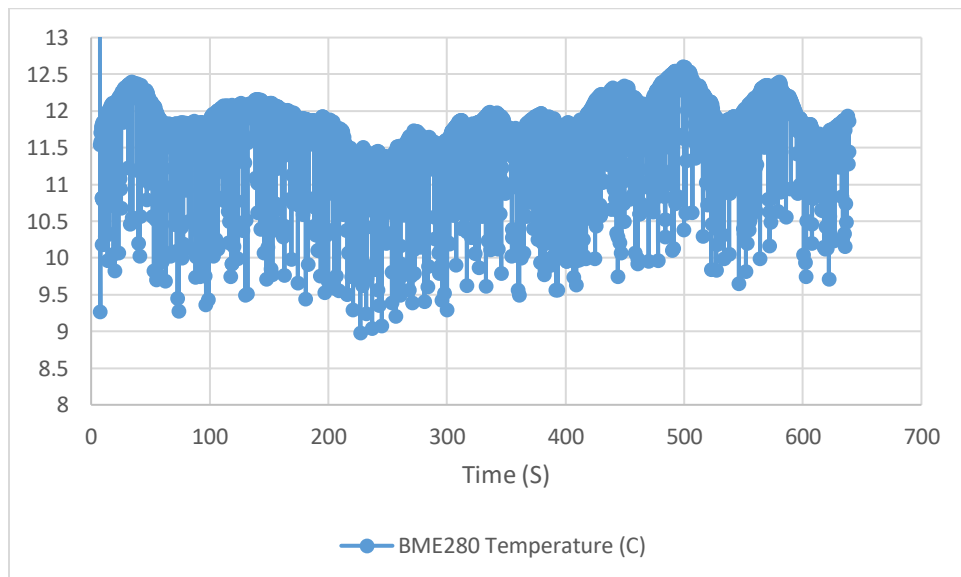


Figure 23. Outdoor Rover BME280 Data

Looking at the CO<sub>2</sub> and tVOC levels of the surrounding air shown in *Figure 24*, we can see that they are significantly lower than our indoor testing. In terms of CO<sub>2</sub>, the readings show an almost 50% decrease when compared to indoors. This can be explained that people exhale

carbon dioxide containing about 35,000 to 50,000 ppm of CO<sub>2</sub> [18] and with the air being unable to flow easily indoors, especially our lab where our indoor testing took place which has no windows and a single door, it is expected that the CO<sub>2</sub> levels would be higher when compared to outside.

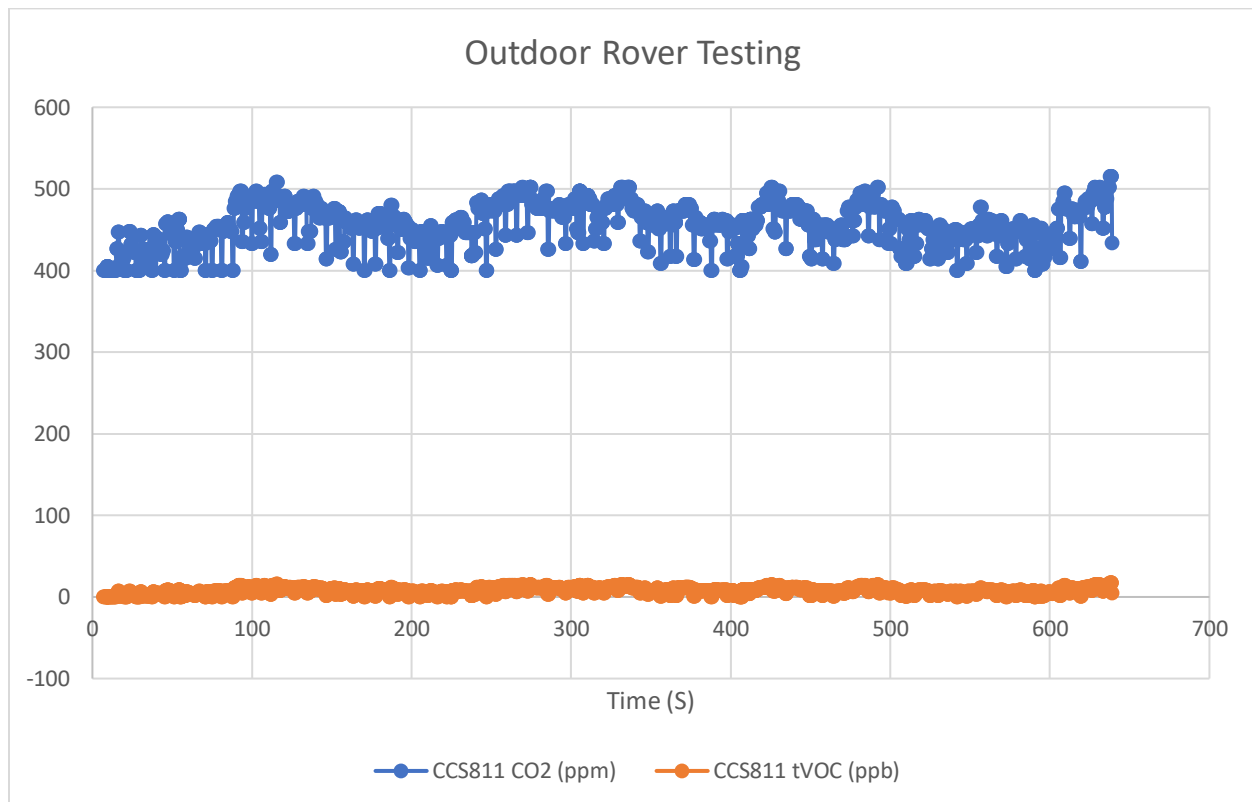


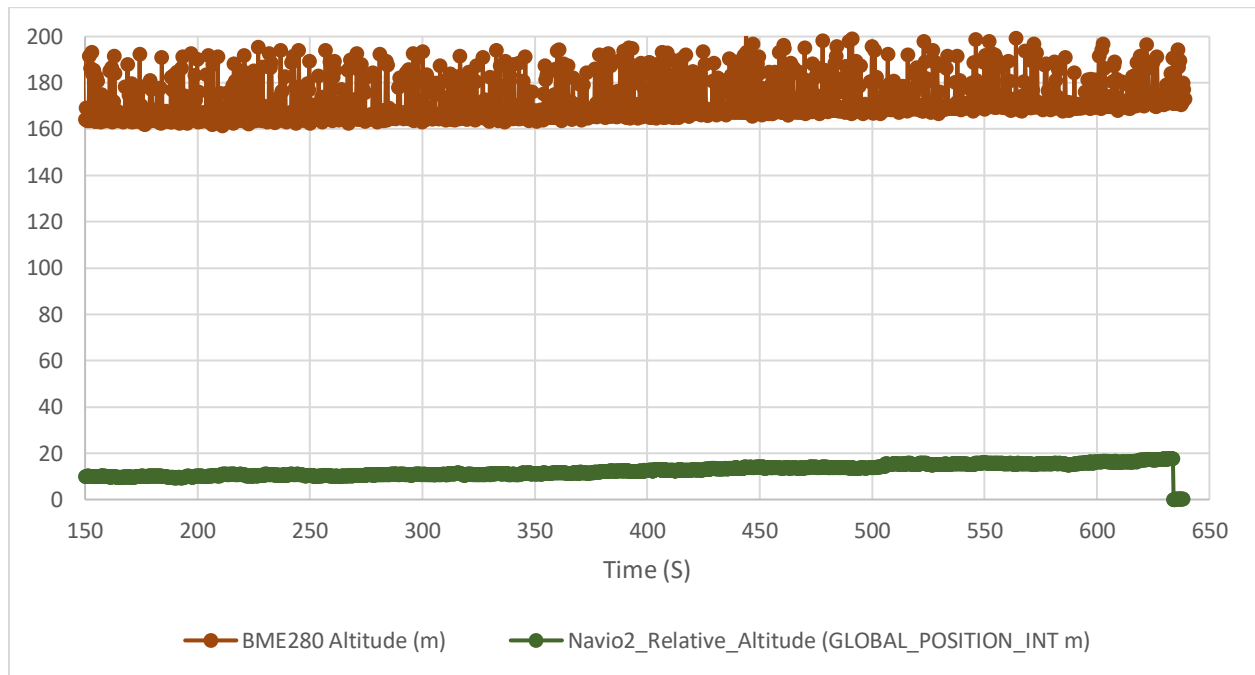
Figure 24. Outdoor Rover CCS811 Data

With access to our GPS data, we can also compare the Navio2 relative altitude provided by the GPS system. What we notice is a difference in how the zero altitude is decided between our BME280 sensor and the Navio2 relative altitude. When looking at the BME280, you will notice a much higher altitude measurement. This is because the BME280 is reading altitude above sea-level compared to the Navio2 relative altitude which provides an offset based on the altitude of when the system is armed. What this means is that the Navio2 relative altitude is the altitude above what is considered the “home-position” or where it first is armed from. Given that our

system was first armed while at a lower altitude, we can see that throughout the operation of the rover we are above our home position by about 11-meter offset when compared to the BME280 which represents an altitude above sea-level of roughly 160 meters. We can compare this value to some given elevation data of the area around Southern Illinois University Edwardsville (SIUE) and see that the known elevation is roughly 174 meters [19]. What we also see is that at the end of the mission when we disarm the rover, we can see the Navio2 Relative Altitude drop to zero, representing the system updating its relative altitude offset for the next time the system is armed. Using a known elevation for SIUE, we can offset the relative altitude of the Navio2 reading and then compare the difference in the two altitude readings again to see how much they actually differ which is shown in *Figure 26*. What we notice is that our Navio2 altitude readings are close to 185 meters above sea level while our BME280 sensor readings at that same time are roughly 165 meters above sea level. Both readings are off by roughly 10 meters to our expected value of 175 meters. Similar to the temperature measurements, we see that the altitude readings of the BME280 are showing signs of noise related once again to the CCS811 sensor causing temperature fluctuations that also affect our sensor's altitude calculations.

My theory for this noise is that when the CCS811 sensor is activated, the heat generated causes the BME280's temperature readings. I decided to test what would happen if adding a delay between measurements of the CCS811 measurement and the BME280 one. My expectation was that if I provide a long enough delay, the ambient temperature should fall back down closer to ambient temperature. To do this, I set a varying delay time between the reading of zero, five, and ten seconds and then recorded the BME280 data. When plotted in *Figure 27*, we saw that the longest delay appears to limit some of the noise. However, increasing the time between readings will reduce the amount of data we can collect, it would be a better alternative to develop a filter

such as a moving average of the last several readings, or to perform several measurements and report the average of those readings.



*Figure 25. Outdoor Rover Altitude Data*

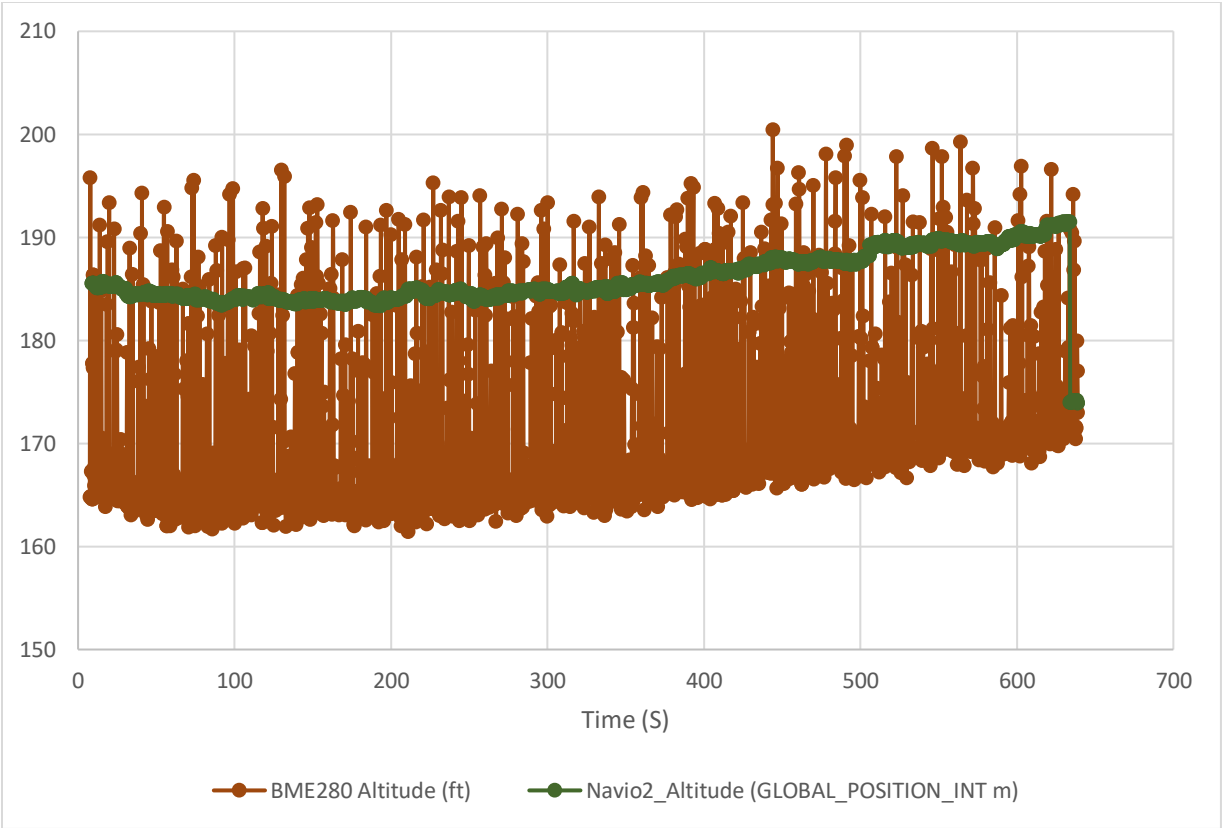


Figure 26. Outdoor Rover Testing Adjusted Altitude Data

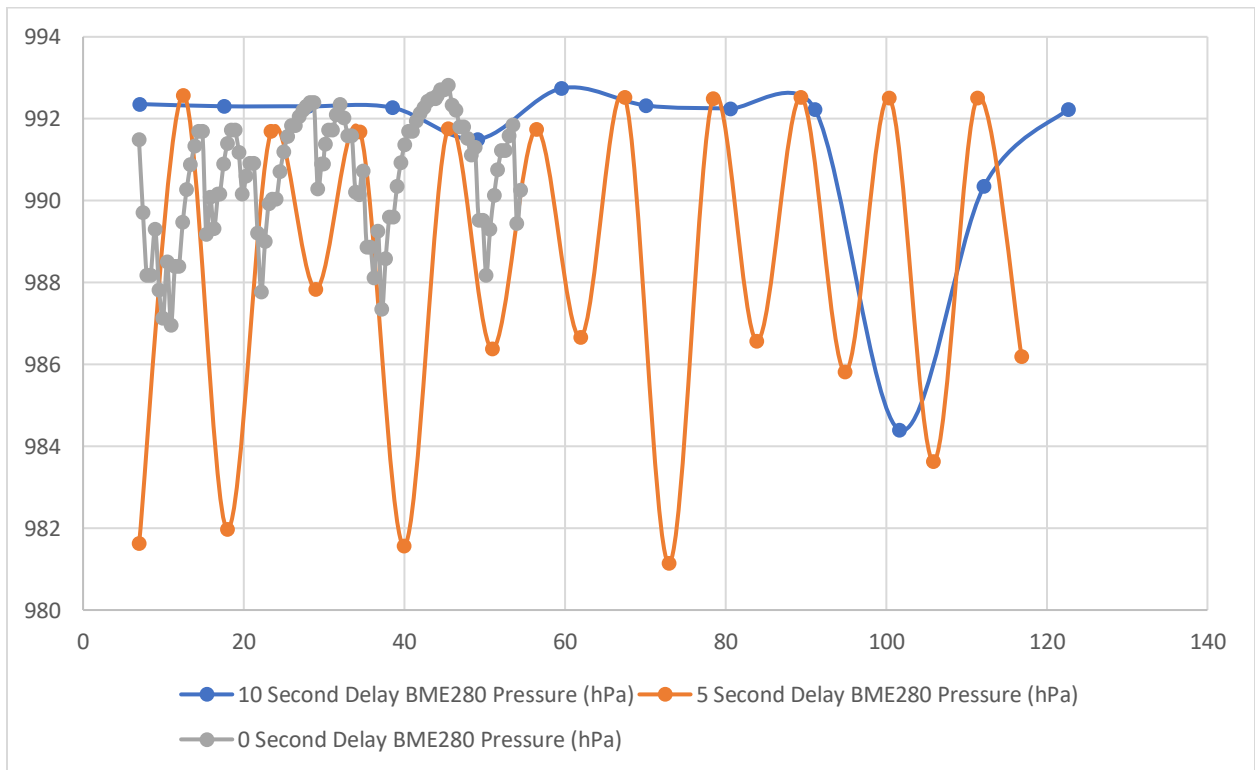


Figure 27. Time Delay BME280 Testing

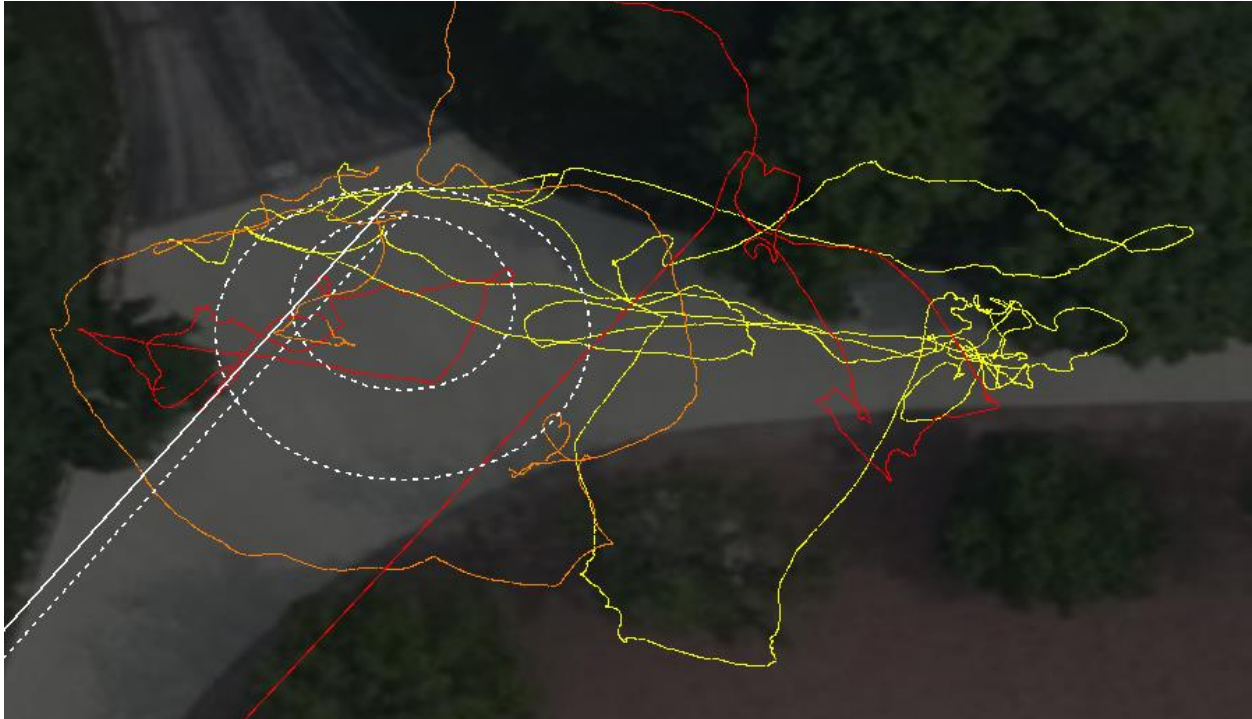


## Waypoint Outdoor Rover Testing

The last testing conducted was using the rover's waypoint navigation capabilities to collect data through a series of movements. Using a set of user-defined GPS coordinates, the Rover uses its current location and desired location to calculate the movement required to reach the desired location. In our GCS, we are capable of adjusting parameters of the navigation including top speed, turning radius, and other performance characteristics. Using these settings, the rover provides commands to the individual motors in order to successfully navigate towards each waypoint autonomously. This is similar to the manual outdoor rover testing previously discussed, with user-input only at starting and ending of the mission. While the sensors were able to successfully gather data, we are still facing an issue with arming and disarming the Companion Computer as well as performance issues with the waypoint system.

In terms of waypoint performance, the main issue occurs when the rover is turning to move to the next waypoint marker. We believe that this is likely caused by the implementation of zero-point turning. Our rover is designed as a skid-steer vehicle, which means instead of a front steering axle, our system turns with the combination of two sets of wheels that create rotations by creating forward and reverse movement on separate wheels. However, the settings and information regarding tuning skid-steer vehicles in Ardupilot is not extensive and some functionality is hard-coded into the Ardupilot Firmware. We believe that by continuing to adjust these settings, we can improve the rover's ability to follow complex waypoint missions. Looking at our map in *Figure 28* we can see three noticeable colors: red, orange, and yellow. The red path is manual operation using the remote controller (RC). The orange path is Acro mode, which is also using the RC sticks and then the input is interpreted as desired angular velocities for the rover and includes more control from the GCS. The last color is yellow which is Auto; this mode

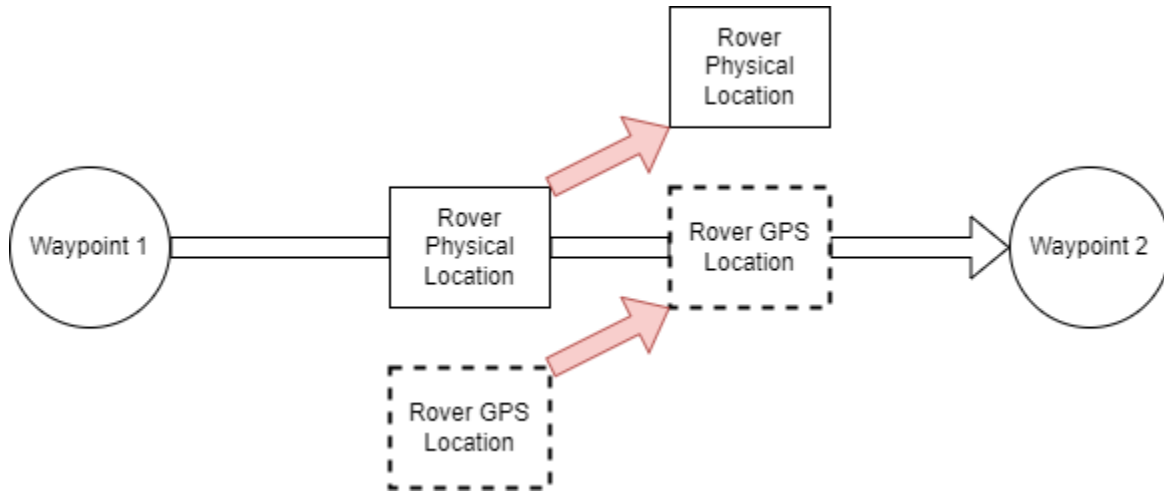
is the full autonomous movement of the rover using turning and throttle control values and GPS waypoints. What we noticed during our first waypoint testing shown in *Figure 28* is that our system was unable to perform the desired waypoints navigation. Our system often attempted to reach a waypoint and would begin spinning out in a continuous circle or would drive past the waypoint then attempt to make a circle to get back to it.



*Figure 28. Failed Outdoor Waypoint Testing Map*

To correct this, we went through the tuning process for the waypoint movement controls. This included reducing the turning rate and our vehicle's max speed. As shown in *Figure 30*, the rover stayed close to the designated path for most of the waypoints. However we noticed that as we moved closer towards buildings, our rover's GPS position began to drift away from its physical location. This incorrect GPS location caused the rover to believe it had deviated from its desired path when in fact it had not. The false error then caused the rover to begin moving towards where the GPS coordinates of the waypoint which resulted in the rover traveling away from the

path. We show an example of this in *Figure 29* where the physical location of the rover is on the path. However, as our GPS location drifts away from the path, the rover moves towards the direction of the path, placing the physical rover location away from the path.



*Figure 29. GPS Location Drift and Rover Movement*

What we believe happened is that the loss of GPS Satellites signal due to the surrounding buildings created a drift in GPS location. During the mission from *Figure 30*, there are two points where the rover began to drift away from the preset path. These occur in the middle of the path as well as the end point, which can be found on the middle and on the right side of the figure. We can see from *Figure 31* there is a drop from a consistent connection of 11 satellites down to a low of 8 active satellites. Ardupilot requires a minimum of 6 satellites to provide a useable GPS position. This drop in satellites could potentially contribute to the decrease in the horizontal accuracy our position estimation shown in *Figure 32* which reaches an error value of almost five meters during the waypoint mission.



Figure 30. Successful Outdoor Waypoint Testing Map

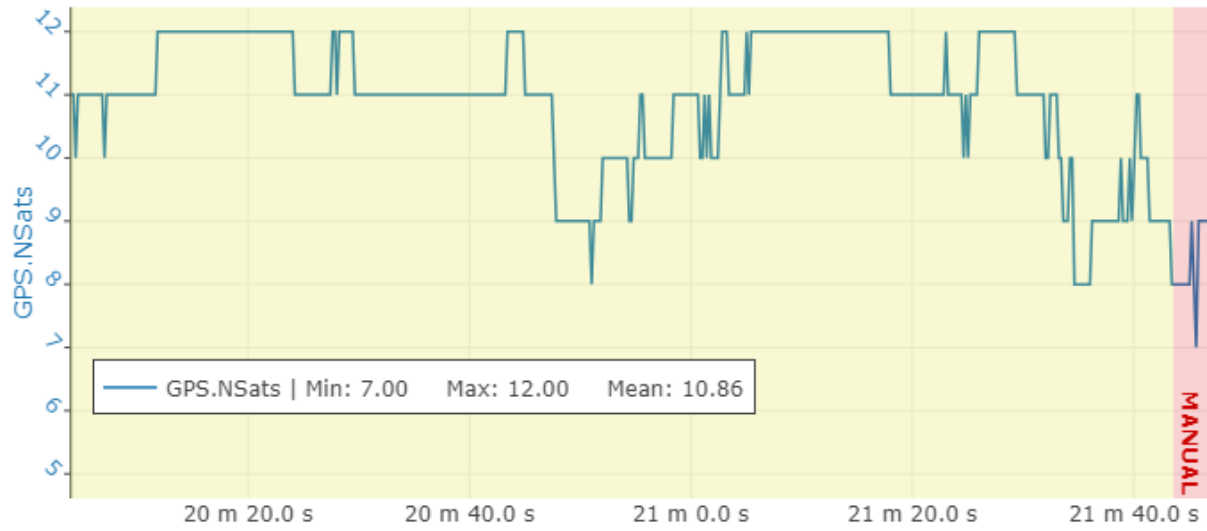
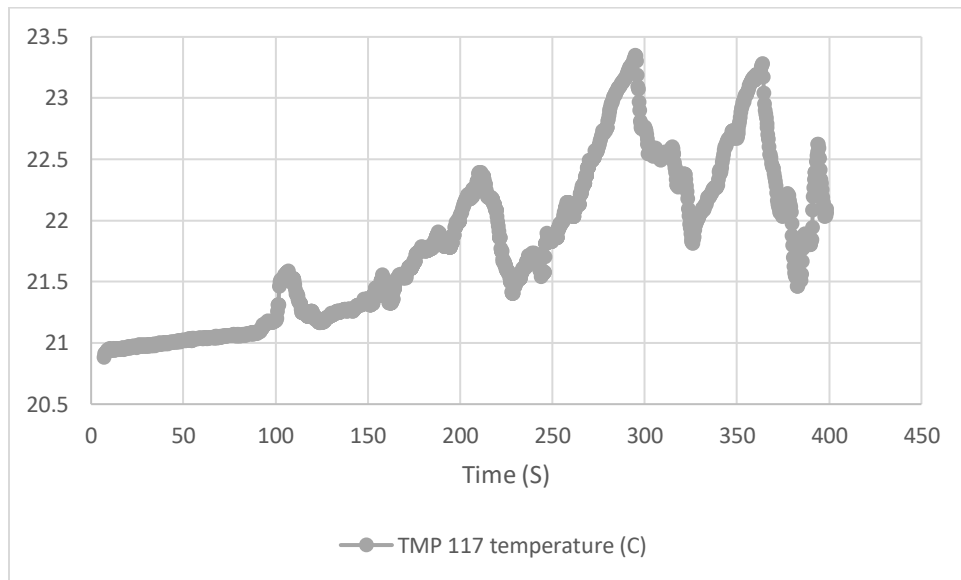


Figure 31. Outdoor Rover Waypoint GPS Satellite Count

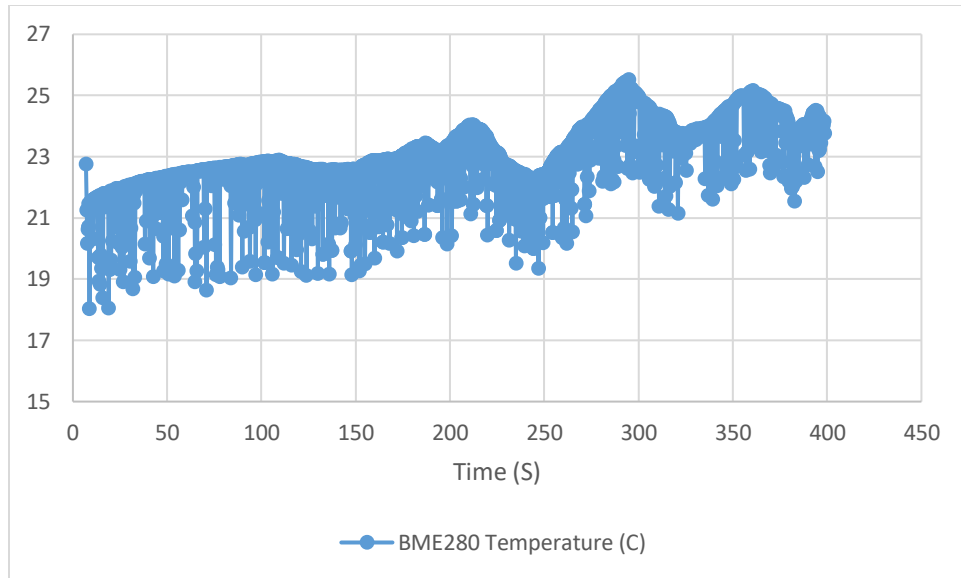


Figure 32. GPS Horizontal Accuracy

As for data collection, we can once again compare the results of various sensors which we have displayed below. When compared to our other outdoor testing, we can see that they both behave similarly. The temperature readings of our TMP117, shown in *Figure 33*, and BME280, shown in *Figure 34*, are close to each other, but once again the BME280 produces a more noisy data compared to our TMP117 due to the self-heating flaw discussed previously.



*Figure 33. Outdoor Rover TMP117 Data*



*Figure 34. Outdoor Rover BME280 Data*

Looking at our air quality sensor, CCS811, in *Figure 35* we can compare the two outdoor data files such as the difference in CO<sub>2</sub> value. In the colder weather testing of the first set of data, our CO<sub>2</sub> value reached a peak of roughly 500 parts per million (ppm) whereas the warmer weather data shows a peak of 650 ppm. However, this is not a drastic difference and compared them to the indoor CO<sub>2</sub> values, we can see that our outdoor data is consistent.

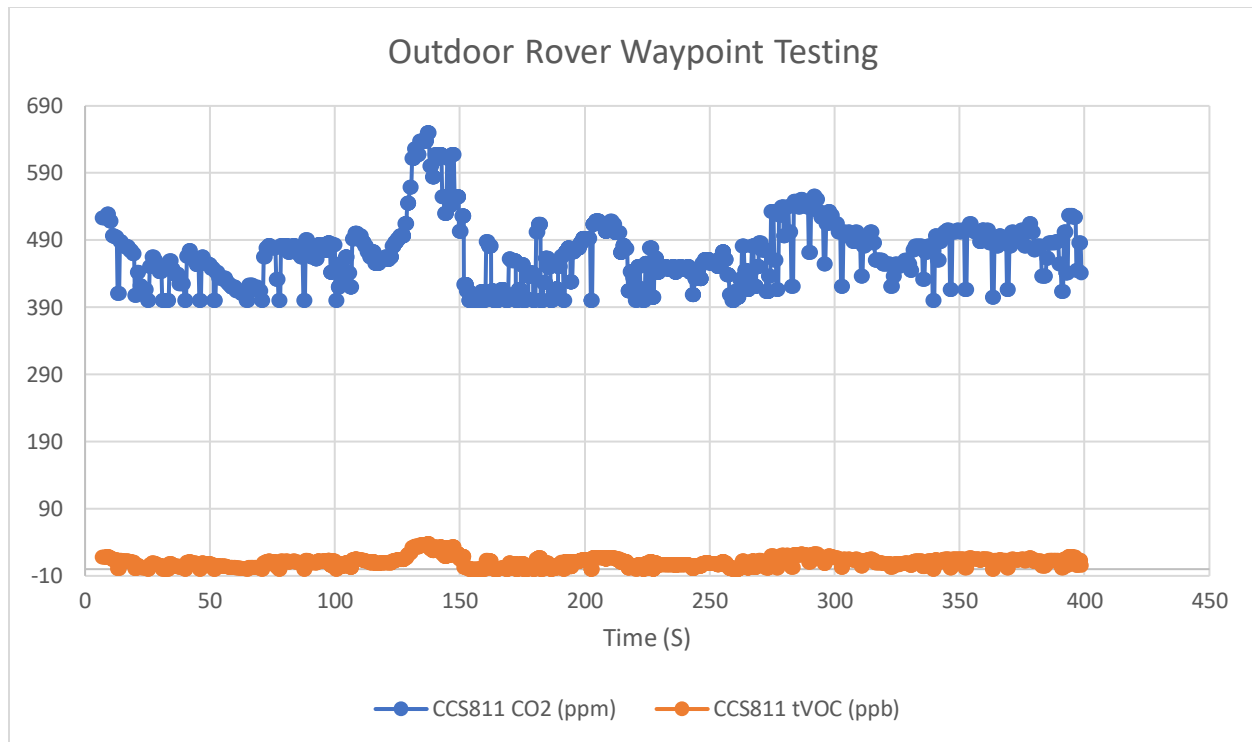


Figure 35. Outdoor Rover Waypoint Air Quality Data

When looking at our altitude data, we notice that there is no major change in altitude shown from either the Navio2 or our Companion. This is due to the testing taking place on a relatively flat location with minor altitude changes. Similar to the above mentioned BME280 sensor, the data consists of a noticeable amount of noise. We plotted two graphs to show the difference between relative altitude and true altitude above sea level given a 174 meter offset. Shown in *Figure 36* and *Figure 37*, we can see that our altitude readings between both sensors are close to each other and to the expected altitude of 174 meters.

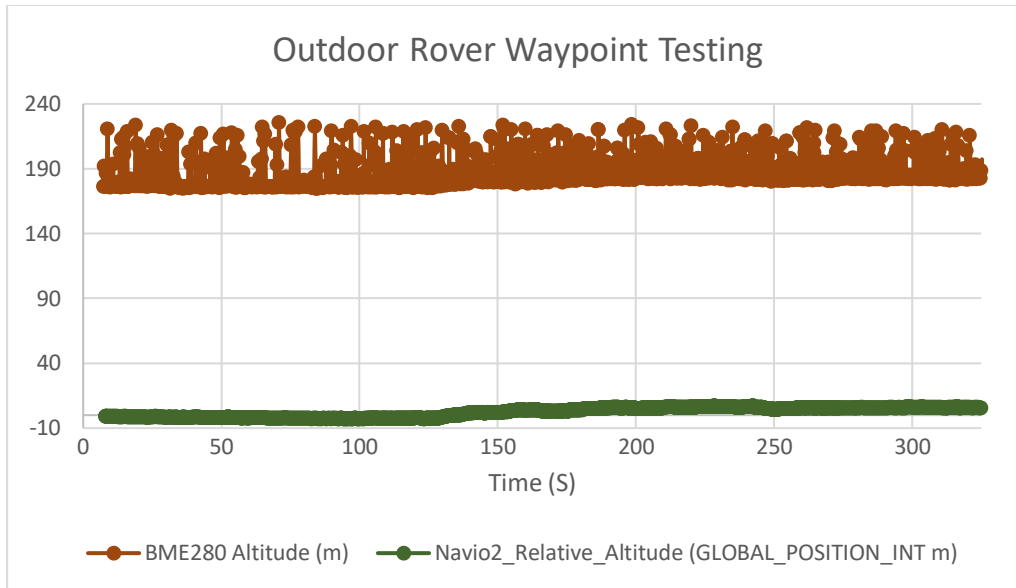


Figure 36. Outdoor Rover Waypoint Altitude Data

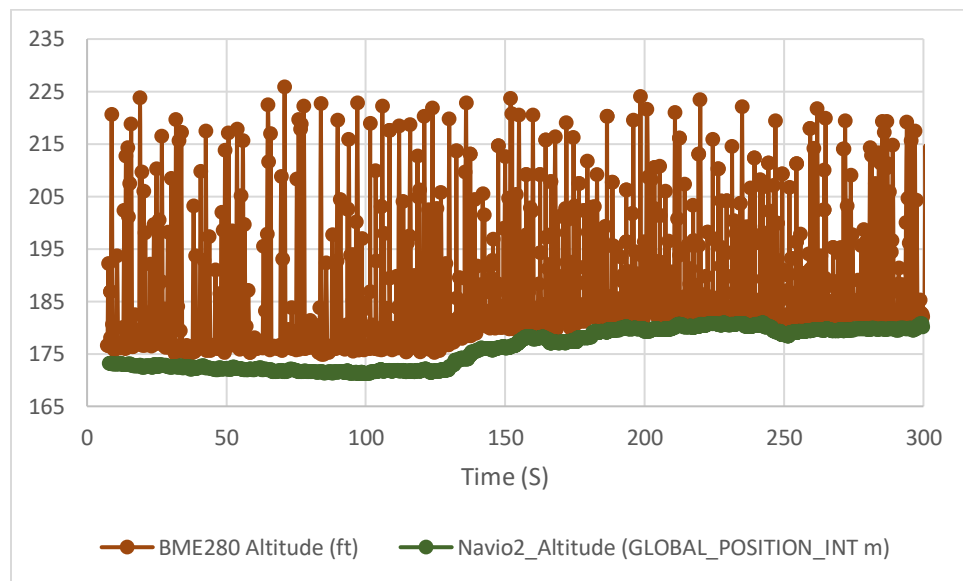


Figure 37. Outdoor Rover Waypoint Offset Altitude Data

Unlike the first outdoor testing set, we did experience occasions where the forward-facing obstacle detection sensor such as our VL53L1X was beneficial. During the performance of our waypoint system, we run fairly close to obstacles on two occasions that required user-intervention to avoid damage. The potential for using this sensor as an emergency stop could prevent collisions and alert the user to attend to the vehicle.



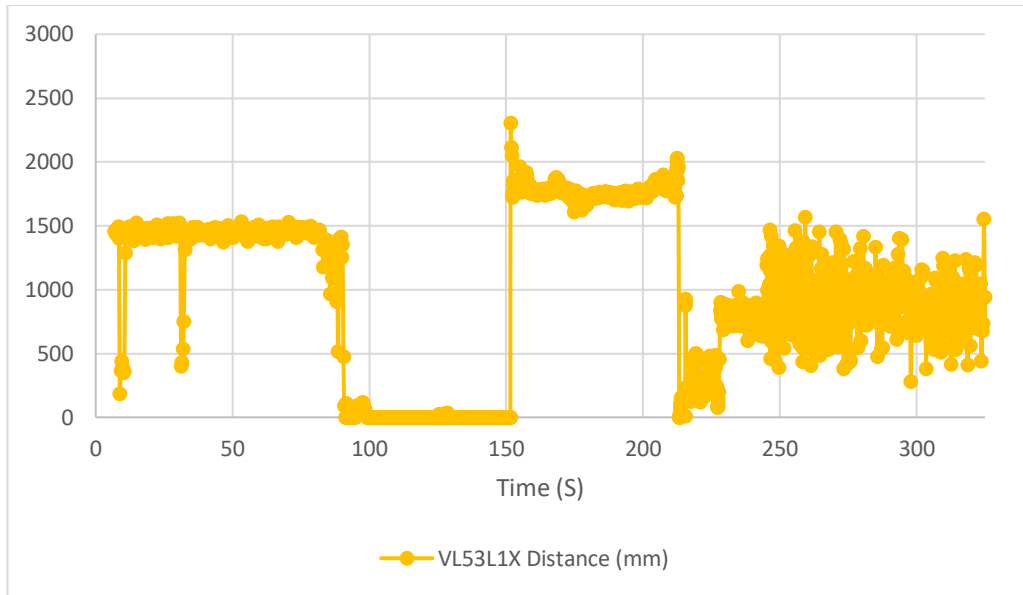
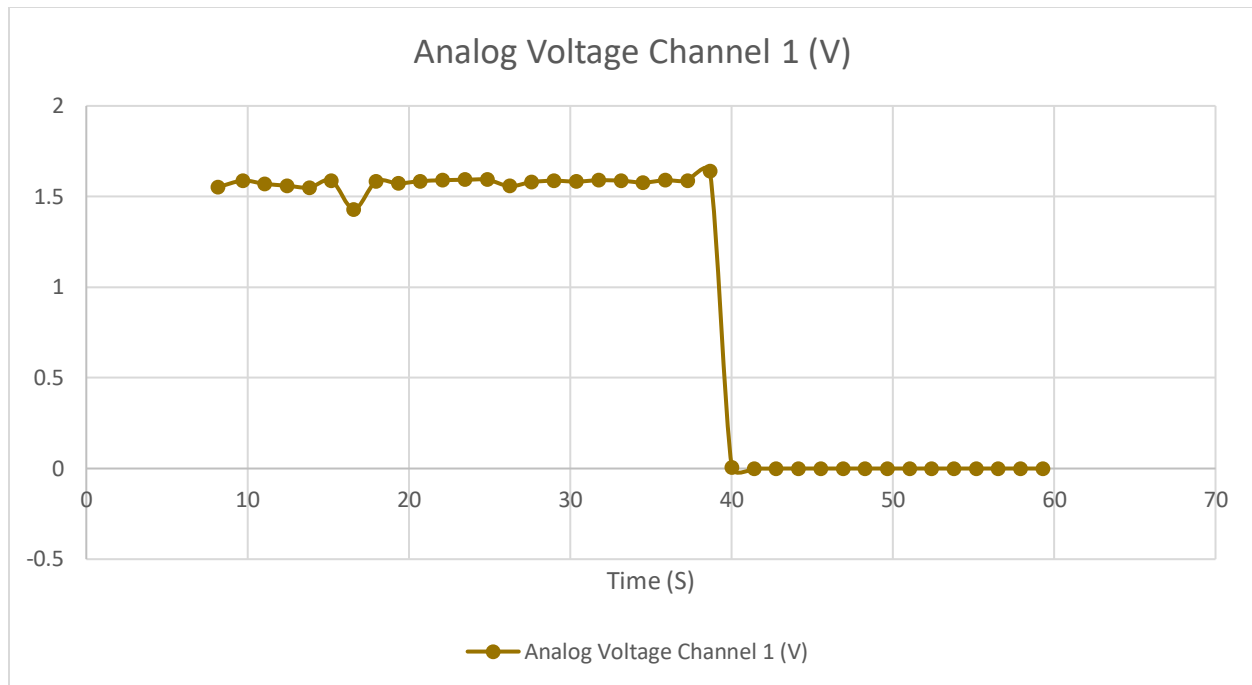


Figure 38. Outdoor Rover Waypoint Distance Sensor Data

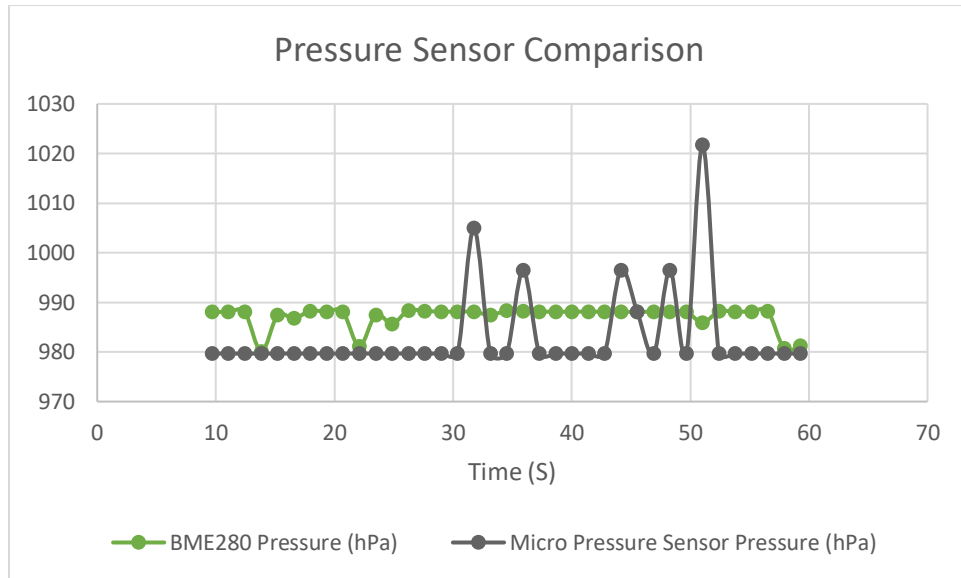
### Additional Sensor Capabilities

At last, I want to mention the other sensors we have developed with this system, but are not used yet in the current testing. Although currently unused, we have begun implementing analog sensors that can measure qualities such as water pressure, water temperature, and total dissolve solids (TDS). As mentioned above, the Raspberry Pi does not have an onboard ADC and therefore is incapable of interacting with analog sensors without additional equipment. To Solve this, we have selected an ADC that is interfaced through the I2C protocol, which has already been adopted for our current sensors. Our ADC is capable of measuring voltage with an 8-bits resolution . This means that given a supply voltage of 3.3 volts, our system can detect a difference of 13 mV. We performed a testing using a 1.6 V input and measuring the average voltage reading, which can be seen in *Figure 39*. While our multimeter read a value of 1.6 Volts, we see that the measured voltage of our sensor ranges from 1.43 Volts to 1.64 Volts.



*Figure 39. Indoor ADC Testing*

The second sensor implemented is a micro pressure sensor with a pressure sensing range of 60 mbar to 2.5 bar. We can compare the results of this pressure sensor with the BME280 pressure sensor as well as the Navio2's barometer. We notice that there is noticeable noise in the micro pressure sensor. We found that the spike appeared when the sensor was moved, in which case the noise is expected. However, we can perform a number of data processing techniques to filter out the noise.



*Figure 40. Indoor Micro Pressure Testing*

## CHAPTER VI

### Future Work

While the system is operating as intended, there are several areas of its capabilities that I believe could be improved as well as features not yet implemented that would be beneficial to have.

#### Try/Except/Else/Finally Implementation

Currently, the code used for the sensor package contains many “Try and Except” statements which I used to perform a function such as initializing a sensor. If the initialization fails due to errors like the sensor not being connected, I can use the “Except” portion to list the sensor as inactive without the entire program halting from the error. However, I have found that there are more options including “Else and Finally” which I can use to clean up the exception of certain functions. For example, currently the system requires a Mavlink connection, however it would be beneficial to allow the sensor system to operate as a stand-alone package if the user want to user a different type of vehicle that does not use the Mavlink communication protocol.

#### External Indicator of Program Running

Currently, I really have no way of knowing whether my program is running without connecting to it using a monitor or SSH. I want to include some LED blinking patterns to indicate if it is operating/armed/disarmed/error. With current system, the vehicle is sending status text messages to the ground control station to tell what the sensor package is doing. This is convenient for me to determine whether my sensor package is operating or not. However, it would also be beneficial to include the LED portion to indicate its state. The reason for including this LED is also due to an additional feature I wish to include, which is a physical switch to force the system to begin collecting data for the sensor package without needing confirmation from Mavlink. We

have developed a prototype version of an external indication of the program running by sending status messages from the Companion Computer to our GCS. In the status message we are able to tell the user whether the system is disarmed, *Figure 41*, or recording data, *Figure 42*.

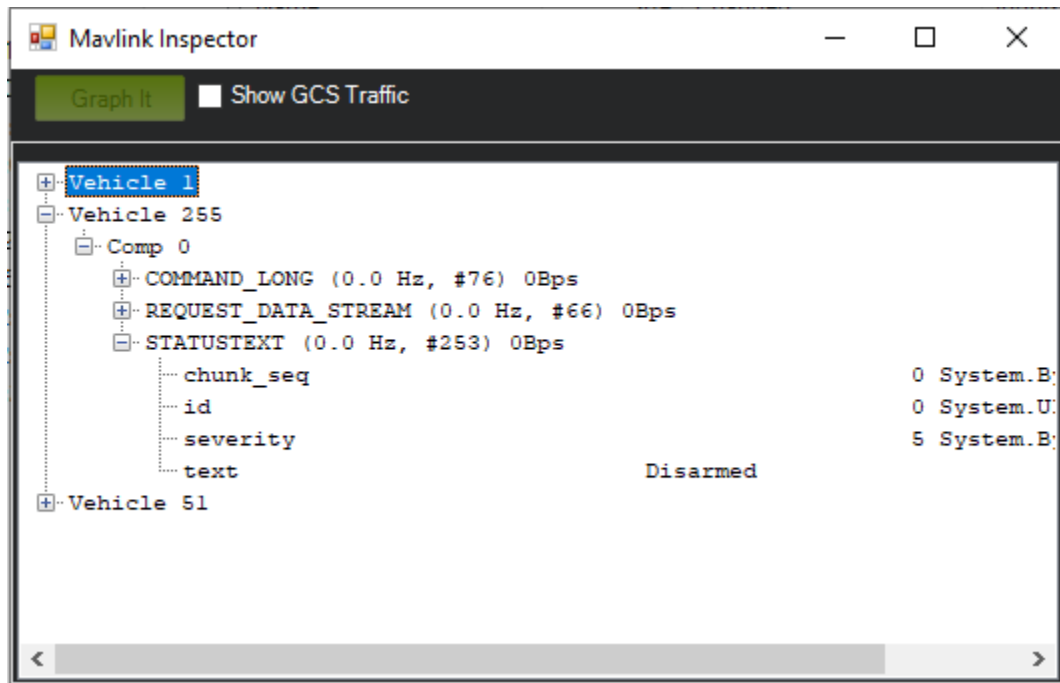


Figure 41. Mavlink Disarmed Notification

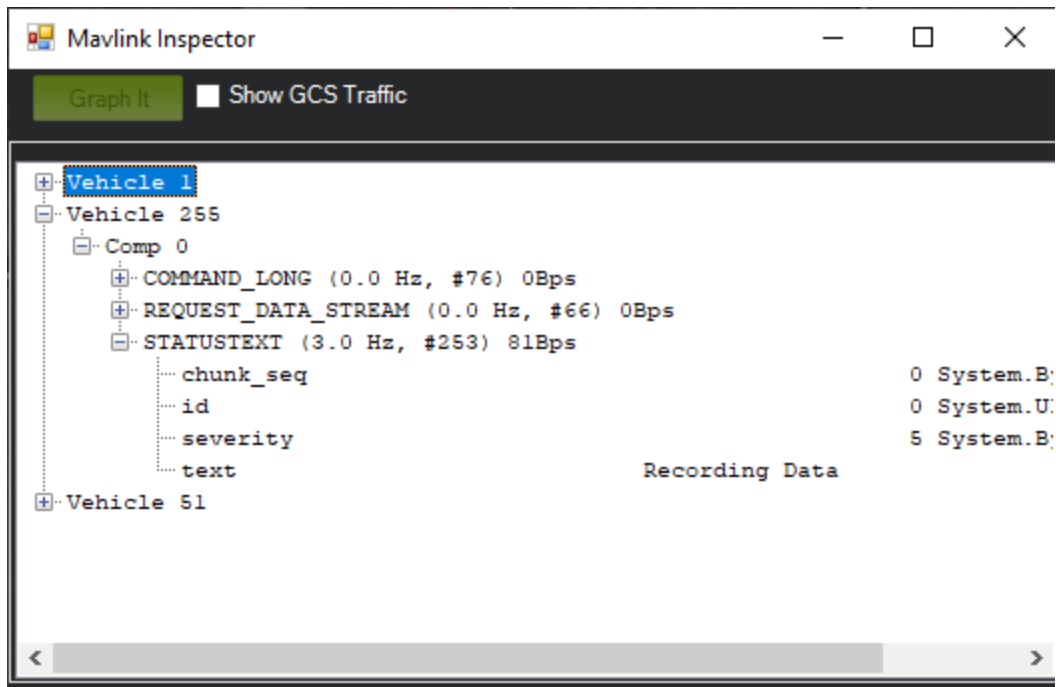


Figure 42. Mavlink Armed Notification

### Container System

Currently, the method of mounting the sensor package to the Navio2 is through a four-post screw mounted to the top of the Navio2. All other sensors are then Velcroed or zip-tied to the frame of the rover. This is inconvenient and more prone to vibration and damages. To resolve these issues, A simple method would be to design a 3D printed holder that the Companion Computer can be set inside and then have the sensors mounted on the casing. This will keep the system compact regardless of what vehicle it is attached to, as well as providing protection against the elements and potential vibration resistance.

### Additional Sensors

We have additional sensors acquired that we have yet to implement within our companion system. This includes additional water-based sensors such as a potential hydrogen (pH), water temperature, and water depth sensors. Once we have further developed our submarine system,

we can perform water quality testing through manual operation or potentially with autonomous navigation through Ardupilot's waypoint functionality.

## CHAPTER VII

### Conclusion

The purpose of this project is the development of an on-board Companion Computer system and additional sensors for data collection of a variety of environmental factors such as temperature, pressure, humidity, air quality, and obstacle detection. Using a Raspberry Pi, we can perform these sensor measurements alongside a connection to our Navio2 flight controller that is capable of sending and receiving data, allowing us to collect information such as speed and position data. With this we can combine the two sets of data and place them in a single data log alongside a camera recording providing visual information as well as the Navio2 position data. During testing, we used a land-based rover to test the system in three scenarios: indoor environment with manual control, outdoor environment with manual control, and outdoor environment using waypoint control.

Through our testing, we successfully ran our data collection system while performing our rover navigation missions. While all of our data collections were operating successfully, we also noticed several items that could be changes in future modifications to greatly improve the overall system performance. This includes hardware modification such as vibration reduction for our camera system and a casing for our Companion Computer, as well as software modification including changing sensor initialization process and adding hardware-based options such as switches and LEDs to physically enable the Companion Computer system and the LED to identify the current status of the Companion Computer.

Moving forward, we will continue to expand the number of compatible sensors and educational kits our system can safely operate with. We believe that this system provides the capabilities of expanding the potential use of all of our planned kits, allow students and other individuals to take



the basic operation they learn in each kit and then combine with the sensor package in real-world applications. For example, an additional kit we plan to develop is a high-altitude balloon. Using the air quality sensors and pressure sensors, we can use the Companion Computer system to measure the environment as the balloon rises through the atmosphere to detect changes such as pressure and CO<sub>2</sub>, which can then be used in comparing pressure-to-altitude calculations with experimental data. Overall, our Companion Computer system provides a versatile platform for students to learn about the multitude of available space-related systems and how they are used.

## CHAPTER IX

### REFERENCES

- [1] U. S. o. Defense, "Funding Opportunity Announcement (FOA) for the National Defense Education Program," Washington Headquarters Services, 2021.
- [2] Emlid, "Emlid Navio2 Documentation," [Online]. Available: <https://docs.emlid.com/navio2/>. [Accessed 1 March 2022].
- [3] Ardupilot Dev Team, "Ardupilot Documentation," [Online]. Available: <https://ardupilot.org/ardupilot/>. [Accessed 1 March 2022].
- [4] V. Ermakov, "mavros," 13 January 2022. [Online]. Available: <http://wiki.ros.org/mavros#Overview>. [Accessed 1 March 2022].
- [5] "Mavlink Common Message Set," [Online]. Available: [https://mavlink.io/en/messages/common.html#MAV\\_MODE\\_FLAG](https://mavlink.io/en/messages/common.html#MAV_MODE_FLAG) . [Accessed 1 March 2022].
- [6] "3DR Dronekit," 2016. [Online]. Available: <https://dronekit-python.readthedocs.io/en/latest/develop/companion-computers.html#:~:text=A%20companion%20computer%20is%20a,provided%20by%20the%20autopilot%20alone..> [Accessed 1 March 2022].
- [7] "Jetson TX2," [Online]. Available: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/>.
- [8] E. Nowak, "Autonomous Navigation of an Unmanned Aerial Vehicle Using Infrared Computer Vision," University of British Columbia, 2018.
- [9] "Battery Eliminator Circuit," 4 June 2021. [Online]. Available: [https://en.wikipedia.org/wiki/Battery\\_eliminator\\_circuit](https://en.wikipedia.org/wiki/Battery_eliminator_circuit). [Accessed 1 March 2022].
- [10] "QWIIC," [Online]. Available: <https://www.sparkfun.com/qwiic> . [Accessed 1 March 2022].
- [11] "Analogue to Digital Converter," [Online]. Available: <https://www.electronics-tutorials.ws/combinational/analogue-to-digital-converter.html> . [Accessed 1 March 2022].
- [12] "GPIO Pinout," [Online]. Available: <https://www.raspberry-pi-geek.com/howto/GPIO-Pinout-Rasp-Pi-1-Model-B-Rasp-Pi-2-Model-B>. [Accessed 1 March 2022].
- [13] "Make a Raspberry Pi 4 Program Start on Boot," [Online]. Available: <https://roboticsbackend.com/make-a-raspberry-pi-3-program-start-on-boot/> . [Accessed 1 March 2022].

- [14] "Pymavlink," [Online]. Available: <https://github.com/ArduPilot/pymavlink> . [Accessed 1 March 2022].
- [15] Englandsaurus, "CCS811/BME280 (Qwiic) Environmental Combo Breakout Hookup Guide," [Online]. Available: <https://learn.sparkfun.com/tutorials/ccs811bme280-qwiic-environmental-combo-breakout-hookup-guide/all#resources--going-further>.
- [16] R. Prill, "Why Measure Carbon Dioxide Inside Buildings?," 2000. [Online]. Available: <https://www.energy.wsu.edu/documents/co2inbuildings.pdf>. [Accessed 1 April 2022].
- [17] "Elevation of Southern Illinois Univeristy Edwardsville," [Online]. Available: [https://elevation.maplogs.com/poi/southern\\_illinois\\_university\\_edwardsville\\_1\\_hairpin\\_dr\\_e\\_dwardsville\\_il\\_usa.383157.html](https://elevation.maplogs.com/poi/southern_illinois_university_edwardsville_1_hairpin_dr_e_dwardsville_il_usa.383157.html) .
- [18] M. Coleman, "RaspiHUD - Ground Based HUD without FPV," 2 August 2014. [Online]. Available: <https://diydrones.com/profiles/blogs/raspihud-ground-based-hud-without-fpv>. [Accessed 1 April 2022].
- [19] M. Rowan, "219 Design Raspberry Pi 4 3D Printed Case," 30 March 2020. [Online]. Available: <https://www.martinrowan.co.uk/2020/03/219-design-raspberry-pi-4-3d-printed-case/>. [Accessed 1 April 2022].
- [20] J. Slabbert, "Raspberry Pi Multiple I2C Devices," 2020. [Online]. Available: <https://www.instructables.com/Raspberry-PI-Multiple-I2c-Devices/> . [Accessed 1 March 2022].
- [21] K.-P. Lindegaard, "smbus2," 17 January 2021. [Online]. Available: <https://pypi.org/project/smbus2/>. [Accessed 1 March 2022].

## APPENDICES

### A: COMPANION COMPUTER CODE

```
#####
# Current Date of Version: 03/29/2022
# Rover IP: 10.0.1.15
#####
# Libraries
import os
from picamera import PiCamera
from picamera import Color
import threading
import VL53L1X
import time
from smbus2 import SMBus
from smbus2 import i2c_msg
from datetime import datetime
from bme280 import BME280
from pymavlink import mavutil

#####
logged_data = ["time(s)"] # Array for Data that will be logged
data = []
#####
# Threading Testing
bus = SMBus(4)
class microPressure:
    def __init__(self):
        try:
            global logged_data
            #address is 0x18
            data1=[]
            msg=i2c_msg.write(0x18,[0xAA,0X00,0x00])
            data1=list(msg)
            bus.i2c_rdwr(msg)
            time.sleep(1)
            val=bus.read_byte(0x18)
            print(val)
            msg=i2c_msg.read(0x18,4)
            bus.i2c_rdwr(msg)
            data1=list(msg)
            status=data1[0]
            print(bin(status))
            self.Pressure=0
            self._running=True
            logged_data.append("Micro Pressure Sensor Pressure (psi)")
            #print("Did I get Here")
        except:
            self._running=False
            print("Exception in Initialization")
    def read_pressure(self):
        try:
            if self._running:
                global data
                #address is 0x18
                cont=[]
```

```

        msg=i2c_msg.write(0x18,[0xAA,0X00,0x00])
        cont=list(msg)
        bus.i2c_rdwr(msg)
        time.sleep(1)
        val=bus.read_byte(0x18)
        #print(val)
        msg=i2c_msg.read(0x18,4)
        bus.i2c_rdwr(msg)
        cont=list(msg)
        outputs=cont[1]<<16 or cont[2]<<8 or cont[3]
        #print(outputs)
        # 10% to 90% calibration
        output_max=0xE66666
        output_min=0x19999A
        Pmax=25.000 # max psi
        Pmin=0.000 # min psi
        #print("Calc")

        self.Pressure=(outputs-output_min)*(Pmax-Pmin)
        self.Pressure=(self.Pressure / (output_max-output_min))+Pmin
        print(self.Pressure)
        data.append(self.Pressure)

    except:
        print("Failed Pressure Reading")

class CCS811:
    def __init__(self):
        # # Keep an eye on this, I might need to do a general call for the
        # Bus to the entire program
        global logged_data
        # try activating the CCS811 Sensor
        try:
            global eco2
            global eTVOC
            # Will need to wait 20 minutes for decent readings

            # Done at Startup, exit boot
            bus.write_byte(0x5B, 0xF4)

            # Set Mode
            bus.write_byte(0x5B, 0x01)
            bus.write_i2c_block_data(0x5B, 0x01, [0b00010000])

            # Go to Data
            # Ignore first 3, they are calibrations
            # bus.write_byte(0x5B,0x02)
            eco2 = 0
            while eco2 == 0:
                bus.write_byte(0x5B, 0x02)
                val = bus.read_i2c_block_data(0x5b, 0x02, 4)
                eco2 = ((val[0] << 8) | (val[1]))
                eTVOC = ((val[2] << 8) | (val[3]))
                time.sleep(1)
            logged_data.append("CCS811 CO2 (ppm)")
            logged_data.append("CCS811 tVOC (ppb)")

```

```

        self._running = True

    except:
        print("CCS811 not Detected")
        self._running = False
        return

    def read_gas(self):
        global eco2
        global eTVOC
        if self._running:
            global data
            # Go to Data
            # Ignore first 3, they are calibrations
            bus.write_byte(0x5B, 0x02)
            val = bus.read_i2c_block_data(0x5b, 0x02, 4)
            eco2 = ((val[0] << 8) | (val[1]))
            eTVOC = ((val[2] << 8) | (val[3]))
            data.append(eco2)
            data.append(eTVOC)

class bme280_sensor:
    def __init__(self):
        # # Keep an eye on this, I might need to do a general call for the
        # Bus to the entire program
        global logged_data
        # Acivate BME280 I2C
        try:
            global bme280_sensor
            bme280_sensor = BME280(i2c_dev=bus, i2c_addr=0x77)
            logged_data.append("BME280 Temperature (C)")
            logged_data.append("BME280 Pressure (hPa)")
            logged_data.append("BME280 Humidity (%)")
            logged_data.append("BME280 Altitude (ft)")
            self._running = True
            self.altitude=0
        except:
            print("BME280 not Detected")
            self._running = False
            return

    def get_bme280_data(self):
        try:
            if self._running:
                global data
                temperature = bme280_sensor.get_temperature()
                data.append(temperature)
                pressure = bme280_sensor.get_pressure()
                data.append(pressure)
                humidity = bme280_sensor.get_humidity()
                data.append(humidity)
                self.altitude = bme280_sensor.get_altitude()
                data.append(self.altitude)
        except:
            if self._running:
                print("Connection to Environmental Sensor Lost")

```

```

        append_blanks(6)

def append_blanks(num_blank):
    global data
    for i in range(num_blank):
        data.append("")

class VL53L1X_distance_sensor:
    def __init__(self):
        # Keep an eye on this, I might need to do a general call for the Bus
        to the entire program
        global logged_data
        try:
            global tof
            bus.read_byte_data(0x29, 0x00)
            #print("Received a Response")
            tof = VL53L1X.VL53L1X(i2c_bus=4, i2c_address=0x29)
            tof.open()
            logged_data.append("VL53L1X Distance (mm)")
            self._running = True
        except:
            print("VL53L1X Sensor not Detected")
            self._running = False
            return

    def measure_distance(self):
        try:
            if self._running:
                global data
                tof.start_ranging(1)
                distance = tof.get_distance()
                # print(distance)
                tof.stop_ranging()
                data.append(distance)
                return distance
        except:
            if self._running:
                print("Connection to VL53L1X Lost")
                append_blanks(1)

class Tmp117_Sensor:
    def __init__(self):
        # Keep an eye on this, I might need to do a general call for the Bus
        to the entire program
        try:
            global logged_data
            tmp117_addr = 0x48
            tmp117_reg_temp = 0x00
            tmp117_reg_config = 0x01
            bus.read_byte_data(tmp117_addr, 0x00)
            #print("Received a Response")
            logged_data.append("TMP 117 temperature (C)")
            self._running = True
        except:

```

```

        print("TMP117 Sensor not Detected")
        self._running = False
        return

def terminate(self):
    self._running = False

# Calculate the 2's complement of a number
def twos_comp(self, val, bits):
    if (val & (1 << (bits - 1))) != 0:
        val = val - (1 << bits)
    return val

def tempReading(self):
    try:
        if self._running:
            # Initialize I2C (SMBus)

            global data
            # TMP117 Registers
            tmp117_addr = 0x48
            tmp117_reg_temp = 0x00
            tmp117_reg_config = 0x01
            # Read the CONFIG register (2 bytes)
            val = bus.read_i2c_block_data(tmp117_addr, tmp117_reg_config,
2)

            # print("Old CONFIG:", hex(val[0]),hex(val[1]))

            # Set to 4 Hz sampling (CR1, CR0 = 0b10)
            val[1] = val[1] & 0b00111111
            val[1] = val[1] | (0b10 << 6)

            # Write 4 Hz sampling back to CONFIG
            bus.write_i2c_block_data(tmp117_addr, tmp117_reg_config, val)

            # Read CONFIG to verify that we changed it
            val = bus.read_i2c_block_data(tmp117_addr, tmp117_reg_config,
2)

            # Print out temperature every second
            # Read temperature registers
            val = bus.read_i2c_block_data(tmp117_addr, tmp117_reg_temp,
2)

            temp_c = (val[0] << 8) | (val[1] >> 0)
            temp_c = self.twos_comp(temp_c, 16)
            # Convert registers value to temperature (C)
            temp_c = temp_c * 0.0078125

            data.append(temp_c)
            return temp_c
    except:
        if self._running:
            print("Connection to TMP117 Lost")
            append_blanks(1)

class SCD4X_CO2:

    def __init__(self):

```



```

global logged_data
try:
    self.address=0x62
    time.sleep(1) # Startup time
    bus.write_byte(0x62,0x21b1) # Begin Periodic Measurements
    time.sleep(0.1)
    bus.write_byte(0x62,0xec05)
    time.sleep(0.005)
    read=i2c_msg.read(0x62,9)
    bus.i2c_rdwr(read)
    data=list(read)
    #print(data)
    self.CO2=data[0]<<8 or data[1]
    self.Temp=data[3]<<8 or data[4]
    self.RH=data[6]<<8 or data[7]
    self.Temp=-45.0 + 175*self.Temp/65536
    self.RH=100.0*self.RH/65536
    #print("CO2: " +str(CO2))
    #print("Temp: " +str(Temp))
    #print("RH: "+str(RH))
    logged_data.append("SCD4X CO2 PPM")
    logged_data.append("SCD4X Temperature (C)")
    logged_data.append("SCD4X Relative Humidity %")
    self._running=True
except:
    self._running=False
    print("SCD4X Failure")

def readSensor(self):
    global data
    if self._running:
        time.sleep(5)
        bus.write_byte(0x62,0xec05)
        time.sleep(0.005)
        read=i2c_msg.read(0x62,9)
        bus.i2c_rdwr(read)
        result=list(read)
        #print(data)
        self.CO2=result[0]<<8 or result[1]
        self.Temp=result[3]<<8 or result[4]
        self.RH=result[6]<<8 or result[7]
        self.Temp=-45.0 +175.0*self.Temp/65536.0
        self.RH=100.0*self.RH/65536.0
        data.append(self.CO2)
        data.append(self.Temp)
        data.append(self.RH)

class FullLog:
    def __init__(self):
        # Performance initialization
        # Put the created log document in here
        # probably initialized each arm, or create a re_initialize function
        self.name_file = datetime.now().strftime('%Y%m%d_%H%M%S')
        os.mkdir("/home/pi/Documents/logs/" + self.name_file)
        self.video_file = "/home/pi/Documents/logs/" + self.name_file + "/" +
self.name_file + ".h264"
        self.log_file = self.name_file + "/" + self.name_file + ".txt"

```

```

def log_data_file(self, data):
    # Log Current Data
    # log data to an file as an append
    file = open(r"/home/pi/Documents/logs/" + self.log_file, "a")
    for L in range(len(data)):
        if L == len(data) - 1:
            file.write(str(data[L]) + "\r\n")
        else:
            file.write(str(data[L]) + ",")
    file.close()

def initialize_log_file(self):
    # might be the new function for re-initializing
    # log data to an file as an append
    self.name_file = datetime.now().strftime('%Y%m%d_%H%M%S')
    os.mkdir("/home/pi/Documents/logs/" + self.name_file)
    self.video_file = "/home/pi/Documents/logs/" + self.name_file + "/" +
self.name_file + ".h264"
    self.log_file = self.name_file + "/" + self.name_file + ".txt"

class MavConnection:
    def __init__(self, time_gap=1):
        global master

        try:
            master = mavutil.mavlink_connection("/dev/ttyS0", baud=115200)
            print("Waiting for Heartbeat")
            val = master.wait_heartbeat(timeout=10) # Try adding a timeout
to see if we actually get through or not
            print(val)
            # time.sleep(5)
            if val == None:
                self._running = False
            else:
                print("Heartbeat Received")
                master.mav.request_data_stream_send(master.target_system,
master.target_component,

mavutil.mavlink.MAV_DATA_STREAM_ALL, 1, 0)
                time.sleep(0.1)
                master.mav.command_long_send(master.target_system,
master.target_component,

mavutil.mavlink.MAV_CMD_SET_MESSAGE_INTERVAL, 0,

mavutil.mavlink.MAVLINK_MSG_ID_GLOBAL_POSITION_INT, 1e6 / 1, 2, 0, 0, 0, 0)
                time.sleep(0.1)
                master.mav.command_long_send(master.target_system,
master.target_component,

mavutil.mavlink.MAV_CMD_SET_MESSAGE_INTERVAL, 0,

mavutil.mavlink.MAVLINK_MSG_ID_GPS_STATUS, 1e6 / 1, 2, 0, 0, 0, 0)
                time.sleep(0.1)
                self.nav_time = ''

```

```

        self.nav_lat = ''
        self.nav_long = ''
        self.nav_rel_alt = ''
        self.nav_alt = ''
        self.nav_vx = ''
        self.nav_vy = ''
        self.nav_vz = ''
        self.nav_hdg = ''
        self.bad_data = 0
        self.check_once = 0
        self.base_time = 0
        self.overtime = 0
        self.over_time = 0
        self.time_gap = time_gap
        self.heartBeatCount = 0
        self.heartBeatArmed = 129
        self._running = True
    except:
        print("Failed to Initialize Mavlink")
        self.running = False

    def initialize_mavlink_log(self):
        if self._running:
            global logged_data
            logged_data.append("Navio2_Time (GLOBAL_POSITION_INT, ms)")
            logged_data.append("Navio2_latitude (GLOBAL_POSITION_INT,
degE7)")
            logged_data.append("Navio2_longitude (GLOBAL_POSITION_INT,
degE7)")
            logged_data.append("Navio2_Relative_Altitude
(GLOBAL_POSITION_INT, mm)")
            logged_data.append("Navio2_Altitude (GLOBAL_POSITION_INT, mm)")
            logged_data.append("Navio2_Ground_X_Speed (GLOBAL_POSITION_INT,
cm/s)")
            logged_data.append("Navio2_Ground_Y_Speed (GLOBAL_POSITION_INT,
cm/s)")
            logged_data.append("Navio2_Ground_Z_Speed (GLOBAL_POSITION_INT,
cm/s)")
            logged_data.append("Navio2_Heading (GLOBAL_POSITION_INT, cdeg)")

    def clear_buffer(self):
        if self._running:
            overtime = 1
            while overtime == 1:
                check = master.recv_match().to_dict()
                try:
                    self.over_time = check['time_boot_ms']
                except:
                    pass
                if (data[0] - (self.over_time - self.base_time) / 1000) <
self.time_gap:
                    overtime = 0
    def send_message(self, flag):
        global heartBeatArmed
        if flag==1:
            master.mav.statustext_send(mavutil.mavlink.MAV_SEVERITY_NOTICE,
"Recording Data".encode())

```

```

else:
    master.mav.statustext_send(mavutil.mavlink.MAV_SEVERITY_NOTICE,
                              "Disarmed".encode())

def get_message(self):
    global check_once
    global bad_data
    global data
    global heartBeatArmed
    global test_mode

    if self._running:
        try:
            msg = master.recv_match().to_dict()
            time.sleep(.25)

            try:
                self.over_time = msg['time_boot_ms']
            except:
                print("Overtime Exception")
                pass

            if test_mode == 1:
                print(msg['mavpacketttype'])

            try:
                if msg['mavpacketttype'] == 'BAD_DATA':
                    self.bad_data = self.bad_data + 1
                    print("Consecutive Bad Data: ", bad_data)
                    #print(msg)
            except:
                print("Bad Data Exception")
                pass

            try:
                if self.check_once == 0:
                    self.base_time = msg['time_boot_ms']
                    self.check_once = 1
            except:
                print("Check Once Exception")
                pass

            try:
                if (data[0] - (self.over_time - self.base_time) / 1000) >
5:
                    self.clear_buffer()
            except:
                print("Clear Buffer Exception")
                pass

            if msg['mavpacketttype'] == 'NAV_CONTROLLER_OUTPUT':
                try:
                    print(msg['wp_dist'])
                except:
                    print("Time Failed to Load")

            if msg['mavpacketttype'] == 'HEARTBEAT':
                try:

```

```

        # print(msg)
        if msg['base_mode'] < 190:
            self.heartBeatCount = self.heartBeatCount + 1
            print("Count Incremented")
        else:
            self.heartBeatCount = 0
            print("Count Reset")

        if self.heartBeatCount > 10:
            self.heartBeatArmed = 0
        if msg['base_mode'] > 100:
            self.heartBeatArmed = 129
        # heartBeatArmed=msg['base_mode']
        #print(self.heartBeatCount)
    except:
        print("Heart Beat Exception")
        pass

    if msg['mavpackettype'] == 'GLOBAL_POSITION_INT':
        try:
            bad_data = 0
            self.nav_time = msg['time_boot_ms']
            self.nav_lat = msg['lat']
            self.nav_long = msg['lon']
            self.nav_rel_alt = msg['relative_alt']
            self.nav_alt = msg['alt']
            self.nav_vx = msg['vx']
            self.nav_vy = msg['vy']
            self.nav_vz = msg['vz']
            self.nav_hdg = msg['hdg']
            # print(msg['time_boot_ms'])
        except:
            print("Global Position Failed ")
    except:
        print("Exception in Mavlink")
        # append_Blanks(9)

def append_data(self):
    if self._running:
        data.append(self.nav_time)
        data.append(self.nav_lat)
        data.append(self.nav_long)
        data.append(self.nav_rel_alt)
        data.append(self.nav_alt)
        data.append(self.nav_vx)
        data.append(self.nav_vy)
        data.append(self.nav_vz)
        data.append(self.nav_hdg)

def increment_Sense(counter):
    global listing
    global result_listing
    global incremental
    display_text=[]
    if(len(result_listing)>0):
        print("Counter: " + str(counter))
        print(listing[counter])

```

```

        print(len(listing))
        print(result_listing[counter])
        print(len(result_listing))
        display_text=str(listing[counter])+": "+ str(result_listing[counter])
        if incremental==len(result_listing)-1:
            incremental=0
        else:
            incremental=incremental+1
    return display_text

class ADC():
    def __init__(self):
        try:
            global logged_data
            self.addr=0x48
            write=bus.write_byte_data(self.addr,0x40,0xff)
            #read=i2c_msg.read(self.addr,3)
            #bus.i2c_rdwr(read)
            #for value in read:
                #print(value*(3.3/255.0))
#            write=i2c_msg.write(addr,[0x00])
#            #write=i2c_msg.write(addr,[0x44])
#            time.sleep(0.1)
#
#            bus.i2c_rdwr(write)
#            read=i2c_msg.read(addr,3)
#            bus.i2c_rdwr(read)
#            for value in read:
#                print(value)
#bus.write_i2c_block_data(addr,0b0000000,0)
#bus.read_byte_data(addr,0)
            self.expectedValue=0
            self.ave=0
            self.maxDiff=0
            self.running=True
            logged_data.append("Analog Voltage Channel 1 (V)")
        except:
            self.running=False

    def read_ADC(self,channel):
        global data
        try:
            if self.running:

                write=i2c_msg.write(self.addr,[64+channel])
                #read=i2c_msg.read(self.addr,channel)
                ave=0
                readings=5
                for i in range(readings):

                    read=i2c_msg.read(self.addr,3)
                    bus.i2c_rdwr(write,read)
                    msg=list(read)
                    #print(data[2]*3.3/255.0)
                    ave=ave+msg[2]*3.2/256.0
                    time.sleep(0.01)
                ave=ave/readings

```

```

        #print("Average Voltage: "+str(ave))
        self.ave=ave
        #print("Difference: " +str(self.expectedValue-self.ave))
        #diff=self.expectedValue-self.ave
        #if diff>self.maxDiff:
        #    self.maxDiff=diff
        data.append(ave)

    except:
        print("failed ADC")
def set_DAC(self,value):
    write=bus.write_byte_data(self.addr,0x40,value)
    self.expectedValue=value*0.012566
    print("Expected Value: " +str(value*0.012566))
    #read=i2c_msg.read(self.addr,2)
    #bus.i2c_rdwr(write,read)
    time.sleep(0.1)

def camera_record():
    global Mavlink
    global Log
    global envSense
    global incremental
    camera = PiCamera()
    camera.resolution = (640, 480)
    camera.rotation=180
    camera.framerate = 20
    camera.start_recording(Log.video_file)
    camera.annotate_background = Color('black')
    camera.annotate_foreground = Color('white')
    print("Camera Recording")
    alt_text=increment_Sense(incremental)
    time.sleep(3)
    while Mavlink.heartBeatArmed == 129 or test_mode == 1:
        time.sleep(2)
        camera.annotate_text =alt_text
        alt_text=increment_Sense(incremental)
    camera.stop_recording()
    camera.close()

#####
# Adjustable Variables
test_mode = 1 # This is whether we ignore whether the system is armed or not
and performs the data logging
check_once = 0
bad_data = 0
incremental=0
listing=[]
result_listing=[]
#####
Mavlink = MavConnection()
Log = FullLog()
logged_data = ["time(s)"] # Array for Data that will be logged
heartBeatArmed = 0

```

```

initialize_items = 0
# Main Code
while True:
    start_time = time.time()
    while Mavlink.heartBeatArmed != 129 and test_mode == 0:
        print("Disarmed")
        logged_data = ["time(s)"] # Array for Data that will be logged
        initialize_items = 0
        time.sleep(.1)
        Mavlink.get_message()
        Mavlink.send_message(0)

    while Mavlink.heartBeatArmed == 129 or test_mode == 1:
        print("Logging")

        if initialize_items == 0:
            Mavlink = MavConnection()
            Log.initialize_log_file()
            distanceInitialize = 0
            check_global_pos_int = 0
            gas_sense = CCS811()
            tempSense = Tmp117_Sensor()
            distSense = VL5311x_distance_sensor()
            envSense = bme280_sensor()
            microPress= microPressure()
            adaSensor=SCD4X_CO2()
            ADCSensor=ADC()

            Mavlink.initialize_mavlink_log()
            print("Data Being Logged: ")
            print(logged_data)
            listing=logged_data
            print("\n")
            Log.log_data_file(logged_data)
            x = threading.Thread(target=camera_record)
            x.start()
            initialize_items = 1

        Mavlink.send_message(1)
        data = []
        data.append(round(time.time() - start_time, 3))
        # CCS811 Data
        gas_sense.read_gas()
        # Get TMP117 Data
        temperature = tempSense.tempReading()
        # Get VL53L1X Data
        distance = distSense.measure_distance()
        # Get BME280 and CCS811 data
        envSense.get_bme280_data()
        # Micro Pressure
        microPress.read_pressure()
        #Adafruit Sensor
        adaSensor.readSensor()
        #ADC
        ADCSensor.read_ADC(0)
        # Mavlink Get Message

```



```
Mavlink.get_message()  
Mavlink.append_data()  
  
try:  
    print(data)  
    result_listing=data  
except:  
    pass  
Log.log_data_file(data)  
# time.sleep(0.1)
```