University of Michigan

Final Report
Space 584 W18

Team Too
Mrigank Gupta
Abhiram Krishnan
Alan Rosenthal
Jacob Sigler
Jonathan Zarger

Contents

List of Figures

List of Tables

Nomenclature

ADC Analog to Digital Converter

$APRS$ Automatic Packet Reporting System

CAD Computer Aided Design

CNC Computer Numerically Controlled machining

FTU Flight Termination Unit

GFL Gorguze Family Laboratory

GPS Global Position System

HAB High Altitude Balloon

I^2C Inter-Integrated Circuit

IMU Inertial Measurement Unit

LNA Low Noise Amplifier

$MBuRST$ Michigan Balloon Recovery and Satellite Testbed

PCB Printed Circuit Board

SOP Standard Operating Procedure

SPI Serial Peripheral Interface

SRB Space Research Building

$UART$ Universal Asynchronous Receive Transmit

1 Introduction

HABs provide a platform to study the atmosphere in-situ. The core purpose of this project was to develop a HAB payload that measures atmospheric pressure, temperature, and humidity. Additionally, the payload included a GPS module, IMU, and two cameras. The payload sensors were mounted on a custom PCB. In order to communicate with the ground, several redundant systems were used, all of which rely on APRS. The Trackuino is an APRS tracker based on the Arduino Uno, and the MicroTrak is a commercial APRS tracker built by Byonics. Two Trackuinos and one MicroTrak were integrated with the payload train in several places. To ensure flight termination, an independent FTU was developed to cut the payload from the balloon, and was designed to activate after the expected balloon burst altitude.

2 Payload Subsystems

2.1 Communications

A single Trackuino, assembled by the team, was built into the payload package. The Trackuino is an open source shield that sits on an Arduino Uno. It records GPS position and transmits on the American APRS frequency of 144.39 MHz once per minute, starting once position lock is acquired. The Trackuino, along with its battery, GPS antenna, and APRS antenna, was built into the payload package.

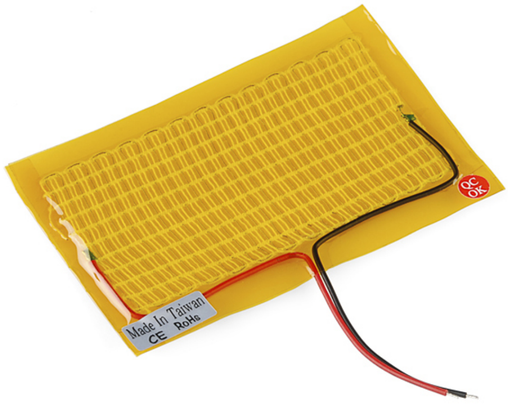
2.2 Sensors

The payload contained a sensor suite consisting of the following components:

- Texas Instruments HDC1080 Humidity Sensor [4]
- TE Connectivity MS5607-02BA03 Pressure Sensor [6]
- Analog Devices TMP36 Temperature Sensor [7]
- PT103J2 Thermistor
- Invensense MPU9250 IMU [3]

These sensors were chosen for a variety of reasons. The HDC1080 humidity sensor was chosen over the lab HIH4030 because it was designed for a 3.3V power supply, and it had a slightly better precision (±2% vs ±3.5% RH). It also had a digital I^2C interface instead of an analog interface. The MS5607 pressure sensor was chosen over the lab MPX5100 because it was designed for a 3.3V power supply, it had a much better range of data (1.5kPa vs 15kPa for the MPX5100), and it had a digital SPI interface instead of an analog interface. The TMP36 temperature sensor tested in the lab was also used, except a surface

(a) Image of A66 with case and mount.



(b) Image of heating pad.

Figure 1: A66 camera and heating pad (not to scale).

mount version was chosen instead of the provided through-hole version. The lab provided PT103J2 thermistor was also used, except the analog signal was wired through a voltage buffer instead of connecting it directly to the microcontroller. The Invensense MPU9250 IMU was chosen primarily because several team members had experience with it. It is a 9-degree of measurement system compared to the labs 3-degree of freedom accelerometer, and it had a digital SPI interface instead of the lab provided ADXL335 analog interface. All of these sensors were connected to the payload as separate surface mount components, except for the thermistor, which was attached through a wiring harness.

2.3 Cameras

Two Apeman A66 Action Cameras were built into the payload, one facing downwards and one facing to the side. They were installed with cases, for protection against moisture and shock (as shown in Figure 1a), but without the base mount. Both cameras were configured to record 1080p video. The A66 is rated to a minimum temperature of 10°C, significantly higher than the expected low temperature during flight of -40°C. As a result, each camera was wrapped with a 5V DC heating pad (COM-11288 on SparkFun, shown in Figure 1b), powered by the primary payload batteries. These heaters kept the entirety of the payload container warm during tests and flight.

As a result of an oversight in the payload mechanical design, the side camera was oriented in portrait mode. This resulted in a less than convenient video from the flight. Beyond this, there were no adverse effects caused by the design decision.

2.4 GPS & Tracking

The payload box included two GPS modules, one on the payload PCB and one on the Trackuino.

The payload PCB used an Adafruit Ultimate GPS Breakout Board, mounted on the motherboard with headers and a pair of threaded steel fasteners. The Ultimate GPS Breakout is built around the MTK3339, and used the internal patch antenna. Latitude, longitude, and altitude were logged from the Ultimate GPS Breakout.

The Trackuino used a SparkFun Venus GPS board, based on the Venus638FLPx receiver. The receiver was connected to an embedded GPS antenna with an LNA. Once GPS lock was acquired by the Venus, the Trackuino began transmitting its location over APRS once per minute. If lock was lost, the Trackuino continued transmitting its last known position every minute.

2.5 Flight Termination Unit (FTU)

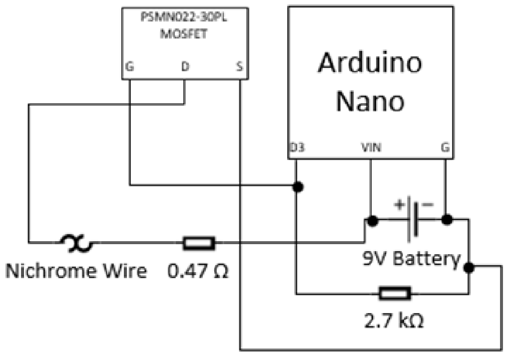
The FTU is responsible for ensuring the payload train is cut from the balloon within a reasonable time frame (2 hours from powering on the FTU, roughly 80 minutes into flight). A nylon rope will connect the payload train to the balloon, and the FTU will be placed at the top of the payload train. The FTU system is comprised of a microcontroller timer (built on an Arduino Nano) and a circuit to run electrical current through a piece of nichrome wire. After the microcontroller timer expires, it will turn on the circuit to run current through the nichrome wire and cut through the rope.

The circuit design for the FTU is shown in the block diagram in Figure 2a, with the fully assembled protoboard shown in Figure 2b. A power resistor is used to limit current through the nichrome wire, and a pulldown resistor is used to ensure that the FTU remains off when the Arduino Nano is not actively driving a signal to the MOSFET gate. The code for the FTU can be found in Appendix D.4. The FTU uses the header file described in Section 5 that describes the flight code and can be found in Appendix D.4.

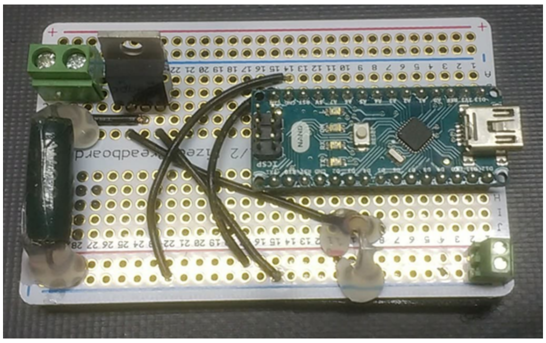
Within the FTU package, components were primarily restrained using Velcro (protoboard, battery), with secondary retention provided by duct tape (battery). Additional mechanical support was provided by potting components on the protoboard with long leads - the pulldown and power resistors - with hot glue. The FTU package was built out of polystyrene foam, and for safety and environmental concerns, was tested to ensure that it would melt/char from the heat of the nichrome, rather than actively burn.

2.6 Structures

Primary payload structures were machined from polystyrene foam and assembled using hot glue. After assembly, the full box was wrapped with duct tape for protection against



(a) FTU block diagram.



(b) Assembled FTU protoboard.

Figure 2: FTU circuit.

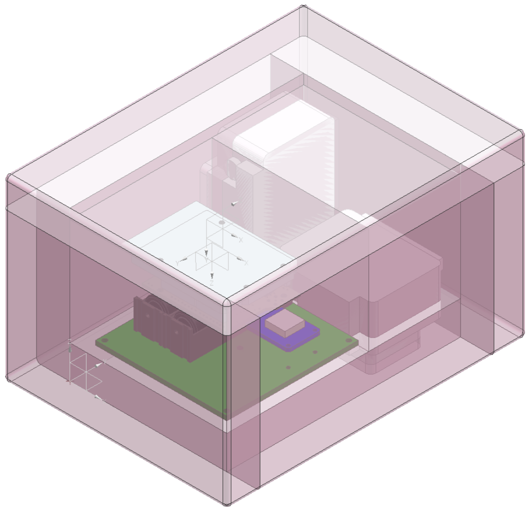
chipping and moisture. An image of the assembled payload, modeled in Siemens NX 11, is shown in Figure 3a. Figures 3b, 3c, and 3d show major components that were also modeled in NX. These models were used to determine the box dimensions required to house all components and to assemble the major components within the box.

Initial designs called for the Trackuino and payload PCB to be restrained by threaded fasteners screwed into nuts or inserts glued to the box. However, the foam material for the box was unable to support this technique. A new design was revised to use Velcro to fasten the Trackuino and PCB to the box.

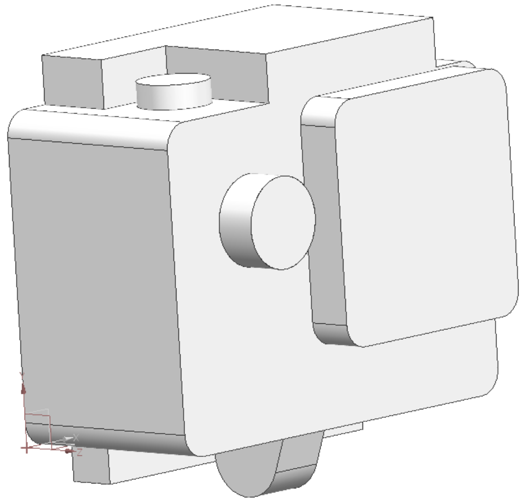
The cameras were held in place by fitted pockets that were machined out using a router. A fitted foam support would constrain the cameras (also machined using a router), and was in turn held in by the box lid. The camera support was also used to mount the GPS antenna and Trackuino battery. The GPS antenna was placed in a slot on top of the support (facing the sky for optimal reception), and the battery was placed in a slot in the middle of the support. Both locations were selected to optimize wire routing.

The thermistor and payload power switch were placed in holes in the walls of the package and hot glued in. The key difficulty with the thermistor was the risk of shorting the leads - this was addressed by insulating/hot gluing them prior to installation. The APRS antenna was installed in a hole through the bottom of the payload, oriented vertically to maximize the lateral range. It was taped to the box on the outside to prevent it from stressing the coaxial cable.

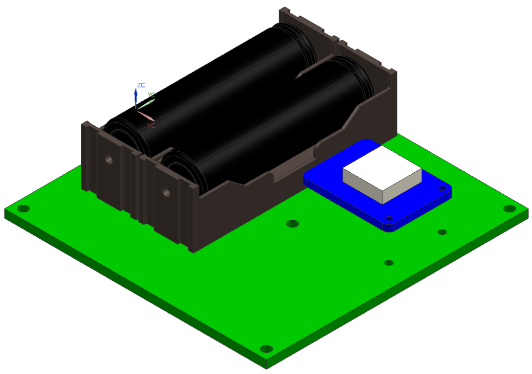
Box manufacturing was facilitated with the use of a Practical CNC router owned by the Department of Aerospace Engineering, located in the GFL. G-code was generated using CATIA v5r26, and modified by hand for use with the BobCAD-Computer Aided Manufacturing (BobCAD-CAM) software paired with the router. Sample toolpaths are shown in Figure 4, and consist of a roughing pass, followed by one or two Z-level or sweeping passes, depending on the needs of the part. All machining was completed with



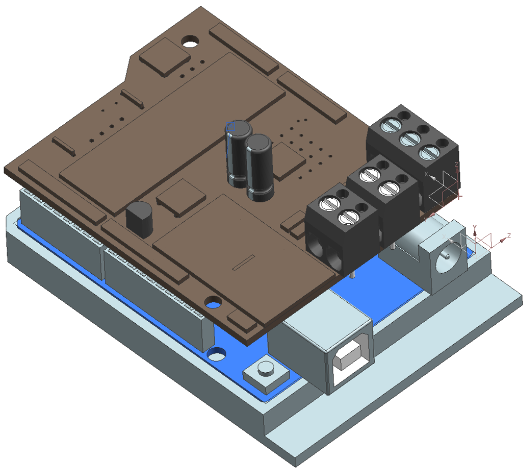
(a) Payload box.



(b) Apeman A66 action camera.

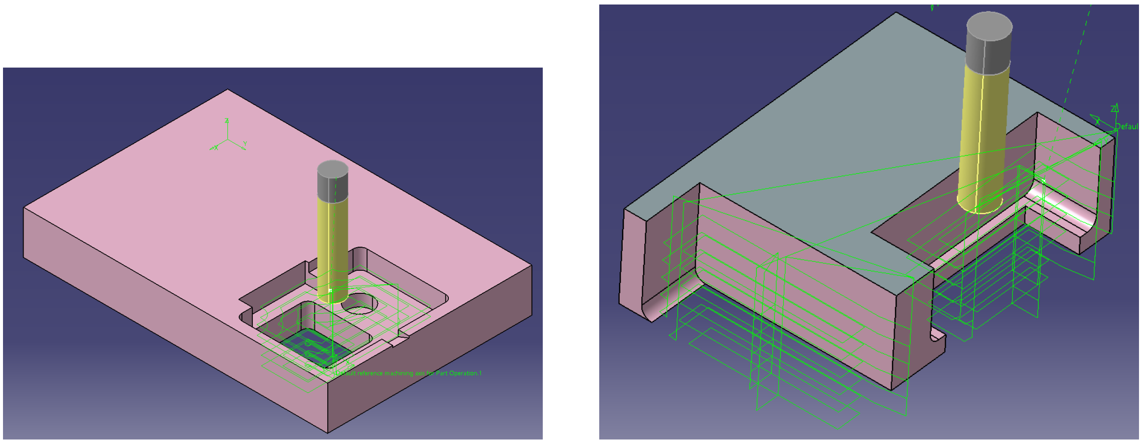


(c) Custom payload PCB.



(d) Trackuino shield on Arduino Uno.

Figure 3: Major components modeled in NX.



(a) Box base. (b) Portion of internal camera support.

Figure 4: Sample toolpaths produced in CATIA.

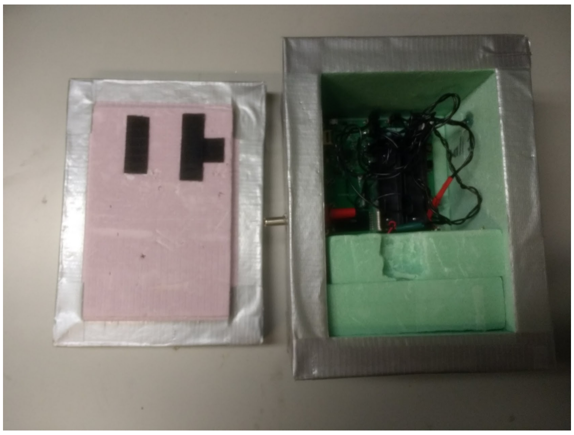


Figure 5: Inside of the payload box

a 0.5" ball nose end mill. Several test pieces were machined first to verify camera fit, and revisions were made to the camera CAD model based on the tests.

After machining, the individual parts were cut to size using a hot wire cutter, and assembled with hot glue. Final fit checks were conducted, and additional holes/adjustments were made by removing foam by hand with a screwdriver. The outsides of the box and lid were then covered in two layers of duct tape for additional protection.

The inside of the completed box is shown in Figure 5. This shows the foam frame, and the printed circuit board inside of the box. The thicker foam section at the bottom of the picture is the camera housing. Figure 6 shows the outside of the fully assembled box.

Figure 6: The fully assembled payload box

3 Mass & Power Budgets

3.1 Mass Budget

Group	Item	Mass [g]	Mass [lbm]	Technique
Payload	Board (with batteries)	165	0.364	Measured
	Camera (x2)	260	0.573	Measured
Power	Trackuino Battery	92	0.203	Measured
Structures	Box+Switch+Thermistor	209	0.461	Measured
	Lid	64	0.141	Measured
	Internal support	12	0.026	Measured
Communications	Trackuino	77	0.170	Measured
	Whip antenna	48	0.106	Measured
	GPS antenna	18	0.040	Measured
Misc	Heater (x2)	16	0.035	Measured
	Wiring/harnesses	50	0.110	Estimated
	Velcro	150	0.331	Estimated
	TOTAL	**1161**	**2.560**	Calculated
	TRUE TOTAL	**1010**	**2.227**	Measured

Table 1: Payload mass budget.

System	Total Mass [lbm]	Source
FTU	0.41	Measured
Parachute	0.43	Measured
Radar Reflector	0.48	Measured
MicroTrak	1.28	Measured
MBuRST Trackuino	0.7	Estimated
Balloon	2.23	Measured
Lines/Clips	0.81	Measured
Team Too Payload	2.23	Measured
ENGR 100 Payload (x3)	3.00	Estimated Max
TOTAL	**11.57**	Calculated

Table 2: Balloon/train mass budget.

3.2 Power Budget

This section will show the process used to determine the power budget of the payload. Battery capacities were determined empirically based on the endurance test. Rough calculations were made to estimate the power draw of components on the payload board to ensure that data could be logged through the entire flight.

The dominating power draw of the board is the heaters at approximately 4 Watts. The remainder of the PCB components were measured to draw a maximum of 0.5 Watts for a total power draw of 4.5 Watts. The payload used two 16850 Lithium-Ion cells wired in series each with a rated capacity of 3.3 Amp-hours for a total capacity of 24.2 Watt-hours. Using this figure, the batteries were calculated to theoretically last for 5.4 hours. To account for the effect of low temperature at high altitude, this value was de-rated by 20% to achieve an estimated 4.3 hours of usable battery life, which was deemed sufficient for flight.

This power budget was tested during the endurance test, with the PCB successfully recording data for all 4 hours.

4 Printed Circuit Board Design

This section describes the PCB developed for logging sensor data. The PCB was designed using the free open source tool KiCad. KiCAD was selected because it does not require a license, so all group members could download the program and collaborate if necessary. Additionally, several group members already had experience with the program.

The PCB schematic is split into several sub-circuit sections which are shown in Figure 32 in Appendix B.1. The primary subsections are the microcontroller (ATSAMD21), barometer (MS5607), IMU (MPU9250), SD Card, humidity sensor (HDC1080), and temperature sensor (TMP36). The main design approach for each sub-circuit was to find an application note for the component (usually provided in the component datasheet) and then design the circuit around that recommendation. This strategy was used to design a schematic for each sensor. The completed system schematic is shown in Figure 33 in Appendix B.2.

Similar to the schematic, the physical PCB layout is also roughly divided into the previously mentioned sub-circuits. Each component within a sub-section was placed near each other and then all of these sub-sections were connected together. The result of this process is shown in Figure 34 in Appendix B.3 and Figure 35 in Appendix B.4. These figures show the copper layers, which connect the components together.

The PCB was ordered from a Chinese company, AllPCB. AllPCB has very low prices ($50 cheaper compared to OshPark) and quick manufacture times. The boards were received the week of February 18th and assembled through spring brake. The board used mostly surface mount components, so assembly was done using a lead paste and hot air station. During this period initial debugging was conducted and several minor mistakes were identified. These mistakes were fixed or "bodged" and the PCB moved to the software development phase.

The fully fabricated and assembled printed circuit board can be see below in Figure 7. It shows all of the surface mount components, including the sensors, the microcontroller, and the SD card in the top left corner. The heater power circuits, including the transistors

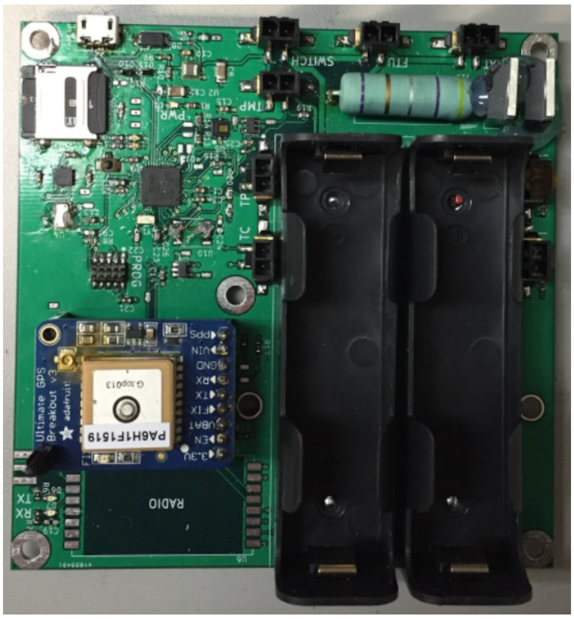


Figure 7: Assembled Payload PCB

and connectors, are shown in the top right corner. The battery connectors are shown in the bottom right corner. The GPS and the unused radio slot are shown in the bottom left corner.

5 Flight Code

The purpose of this section is to describe the structure and function of the flight code used for the payload. The flight software in the payload was written in the Arduino environment, which uses a combination of the C and C++ programming languages. The target device the software was compiled for was the payload microcontroller, an Atmel SAMD21 ARM system. The first point of note is Figure 36 in Appendix D.1, which is a software flow diagram for the flight code. It visually describes the flow of the code through the initialization process, and the main program loop that controls repeating actions. The actual code is presented in Appendix D.2, with the header file in Appendix D.3.

The flight code, running on the microcontroller, interfaces with all of the sensors and the SD card. It periodically collects data, processes it, and then writes it to the SD card so that it can be accessed later.

Some of the sensor code was written with assistance from external libraries. These included code for the humidity sensor [5], code for the IMU [2] and code for the GPS [8]. The barometer code was derived from the specification sheet [6], as was the code for the temperature sensors.

On power-up, the microcontroller runs through an initialization routine. Most of the critical microcontroller functions, like timers, clocks, and interrupts, are initialized behind the scenes in the Arduino framework. After this initialization has completed, the sensor connections are initialized. Several digital sensors were used, so several different microcontroller communication peripherals must be initialized. This includes a SPI port connection, an I^2C port connection, and a UART connection. The ADC is also initialized for the thermistor and battery voltage level, and it is set to a 12-bit resolution.

Each digital sensor has a specialized initialization routine, which is defined by the documentation for the sensor, or sample code found for the sensor. The SD card logging file is also accessed to create a designation that a new set of logging data is being written for the current power cycle.

The program then enters the "loop" phase, where it will repeat the same tasks until it is powered off. The program attempts to execute the following tasks, listed in order of descending priority. If 10ms have passed since the last IMU sampling, it will attempt to collect a data sample from the IMU over SPI. It will log this result and the current time to a IMU data buffer. If 200 ms have passed since the last other sensor sampling, it will attempt to collect data samples from all of the other sensors. This includes reading from the humidity sensor over I^2C, reading from the barometer over SPI, and reading the battery voltage and two temperature values through the ADC. Each of these values is put in its own dedicated buffer. After these sensors have been sampled, the program will attempt to read any available data from the GPS. The program will then toggle the heaters if it is time for them to toggle. The heater pattern is heater 1 on for 15 seconds, both off for 15 seconds, heater 2 on for 15 seconds, both off for 15 seconds. If a complete set of GPS data is found, it will be processed and the useful information will be stored.

Once all of the buffers are full and a GPS sample has been collected, the data is ready to be sent to the SD card. The program runs a moving average filter on the barometer, humidity, battery, and temperature data, and logs the IMU data as normal. The data is accumulated into a C struct, typecast into a buffer of bytes, and then written to the SD card as an array of bytes. This method of writing is significantly faster than storing numbers as ASCII data, though it is much more difficult to parse it out later on the ground. All of the buffers are then cleared. After all of the previous loop steps have been checked or executed, the loop returns to the top.

On the ground, post flight, the binary data format written to the SD card is extracted and converted to a CSV file that can be processed and plotted in MATLAB.

Also of note, our team used the University of Michigan EECS GitLab server for code sharing and revision control. This was a valuable tool to handle a significant quantity of flight code, processing code, and lab code.

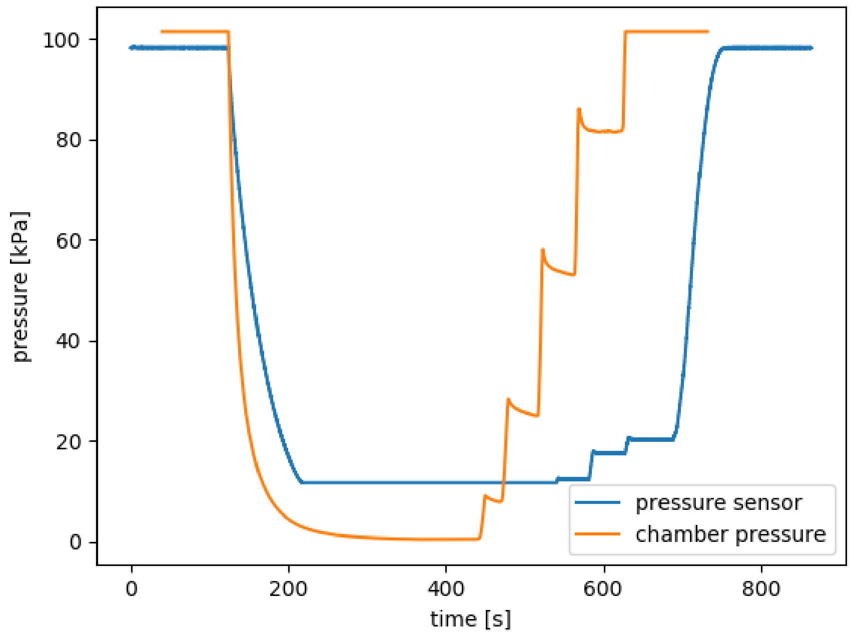


Figure 8: Barometer calibration data.

6 Testing and Verification

6.1 Calibration Tests

Calibration curves for the majority of the sensors were pulled from their respective specification sheets. In order to verify the curves, the sensors were tested in various conditions and the results were compared to known values.

For the two temperature sensors and the humidity sensor, measurements were taken at different conditions and the results were compared to a portable weather station.

To test the pressure sensor, it was placed inside a vacuum chamber which was pumped down below the sensor's minimum pressure rating. The chamber was then pumped up slowly, giving the sensor time to respond. The pressure sensor initially provided in the class bottomed out at 15 kPa, explaining the plateau at the base of the plot, shown in Figure 8. It is also clear that there is a significant lag between the true pressure and the measured pressure. The results of this initial calibration were the primary reason for selection of a different barometer for the payload board.

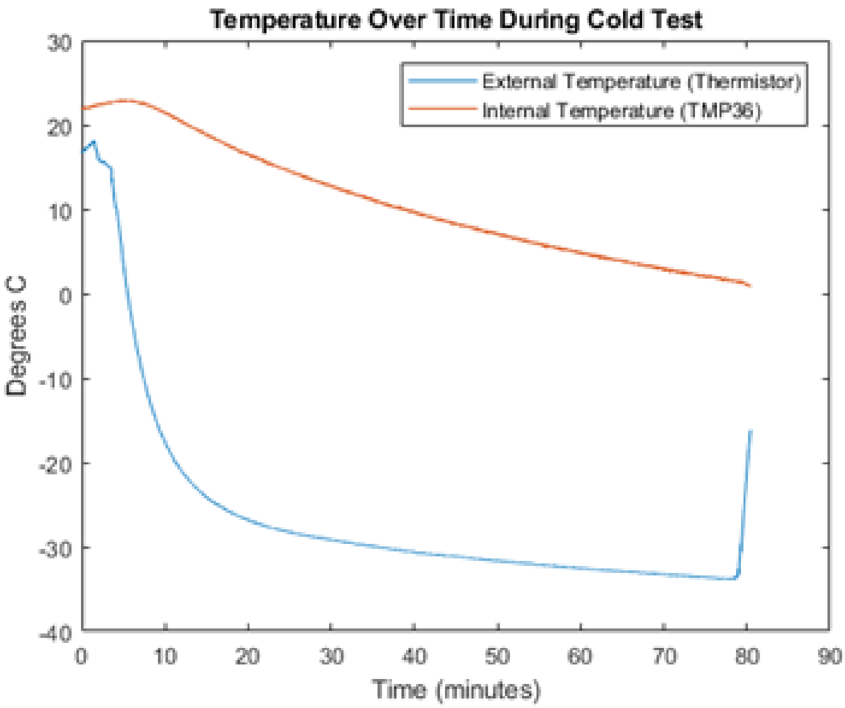


Figure 9: Internal and external temperature during cold test.

6.2 Thermal Test

Thermal testing was required to verify the payload's performance down to -40°C, the lowest expected flight temperature. The payload was fully assembled and placed in a cooler with dry ice for one hour. Data was recorded throughout the course of the test and examined to verify nominal performance. The most important of this data is the temperature data, shown in Figure 9. The external temperature rapidly drops due to the dry ice, while the internal temperature drop is well buffered by the heaters.

All components of the payload were fully operational for the entirety of the cold test.

6.3 Shock Test

Shock testing was used to verify that the payload could survive aerodynamic buffeting in flight and impact with the ground upon landing. Two methods of shock testing were used. First, the payload was thrown directly upwards, simulating a single large shock. Second, the payload was thrown forward with nonzero angular velocity, simulating buffeting from the jetstream.

Several components failed during the shock test. All were repaired and strengthened to ensure that they could withstand flight.

- The GPS unit on the payload board was originally supported by nylon standoffs. Both standoffs sheared during the shock tests. They were replaced by threaded steel fasteners and nuts.
- The batteries on the payload board fell out of their holders. The assembly SOP was updated to include zip-tying the batteries to the holders.
- The Trackuino/APRS antenna coaxial cable sheared at the connector. The cable was replaced and strain relieved. The antenna was also secured to the box to prevent it from mechanically loading the cable.
- Two body ground solder joints on the Trackuino radio module broke. They were re-soldered, and the radio module was hot glued to the Trackuino shield.
- The Arduino Uno fell out of its plastic holder. Threaded steel fasteners were added to hold them together.

6.4 Endurance Test

The payload was fully assembled, and all components were powered on and left to collect data for four consecutive hours. This test is based on the expected deployment time of the payload of two hours (half an hour for preflight operations, and up to ninety minutes for the flight), with a safety factor of two. The primary driver for the test is operation of the Trackuino up until recovery. All payload systems, including sensors and heaters, operated for the full duration of the test. One camera operated for about three hours, and the other camera lasted 76 minutes. Both of these cameras satisfy the minimum requirement of one hour, but only one was expected to last the full flight.

6.5 FTU Test

An FTU test was performed to confirm that it behaved as expected. The FTU box was completely assembled, including the rope to cut, and that rope was clamped off the ground. A timer was started to confirm that the FTU would activate two hours from powering on. The FTU cut the rope at exactly two hours, as expected. A new timer was started to confirm that the FTU would deactivate and cease powering the nichrome wire. After exactly two minutes, the nichrome wire powered off. This successfully mitigates the risk of causing fire or heat related damage if the FTU powers on from the ground. The team also performed a quick test to confirm that the foam used for the FTU box would melt when exposed to the heat of the nichrome instead of burning.

6.6 Ground Station

Significant debugging was required to ensure reliable operation of the ground station. It failed to receive APRS packets from the Trackuinos and MicroTrak for two tests, after which several team members worked to debug it by reading through the radio manual and online documentation for the software used on the ground station. Several problems were identified and corrected:

- The DIN 8 to DB9 cable was faulty and needed to be replaced.
- The DIN 8 side of the cable was plugged into the wrong port in the radio.
- The radio was not configured to transmit to the PC in KISS (Keep It Simple, Stupid) mode.

After these issues were fixed, the ground station was successfully used for the car chase. The knowledge gained from this process was also used to repair the MBuRST ground station on launch day.

6.7 Car Chase

The car chase was successfully completed on March 31, 2018, after two previous failed attempts. Packets received by the ground station during the test are shown in Figure 10. The test started at the SRB parking lot, went around the North Campus Diag, and returned to the SRB parking lot. The numerous points in the parking lot are a result of pretest debugging. Over the course of the test, no packets were dropped, and the chase car occasionally was able to move within line of sight of the tracked car.

7 Launch

7.1 Pre-Launch Operations

The week prior to launch, Go/No-Go slides were created outlining the readiness of the team. These slides included the status of all required tests, the current state of the payload, balloon flight predictions, a mass budget and schedule for the flight day. They were presented at a Go/No-Go meeting two days before flight, along with the the packing checklist and assembly SOP. The day before flight, all items were packed per the checklist. This checklist can be found as part of the Payload Assembly SOP in Appendix C.

Several launch simulations were also conducted, with Athens, MI being selected as the launch site. This resulted in an expected landing location northeast of Britton, MI, as shown in Figure 11. As the bird flies, this is a distance of 76 miles.

Figure 10: Trackuino packets received by ground station during car chase.

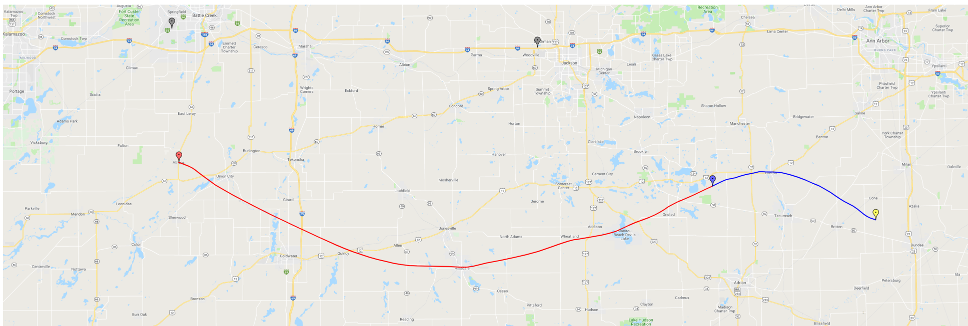


Figure 11: Predicted balloon path from a simulation conducted the night before launch.

7.2 Launch Day

On launch day, teams met an hour and a half before departure to pack up trailer with general launch supplies. These supplies included helium tanks, pressure regulators, balloons, a housing to hold the balloon during filling and various tools. Packing took a total of half an hour, upon which teams awaited the arrival of ENGR 100 students. Once all personnel were present and placed into cars, they began the drive to the launch site at Athens High School in Athens, MI.

Teams arrived at Athens High School at approximately 3 PM and began unpacking payload train supplies, such as the paracord and Velcro needed to build up the payload train. In parallel, assembly began of a balloon housing used to contain the balloon during the fill procedure. The housing was necessary because of high winds on launch day. Next, both teams began organizing the ENGR 100 payloads which had been assigned to respective payload trains. Each payload was secured using an orthogonal wrapping of para-cord. The paracord was tied using a standard ”double” and ”double figure 8 loop” knots with the ends taped to prevent untying. Once the ENGR 100 payloads were secured, tracker payloads were added to the train, including a proprietary MBuRST tracker and MBuRST built Trackuino and MicroTrak. These payloads were secured using a double wrapping of Velcro. At this time balloon fill started and teams began payload assembly. Assembly of the Team Too payload was time sensitive due to the limited battery life of the cameras. Payload train 1 was ready first, and was launched on the first balloon. Once the second payload was complete, and all items on the SOP were checked off, it was integrated into payload train 2 using Velcro and paracord. Finally, the radar reflector, parachute, and FTU were all attached to the payload. A safety line was run from the parachute to the bottom of the payload train in case of a break in the main line. The payload train was completed as the second balloon finished filling, and was launched approximately a half hour after the first balloon.

Once both balloons were launched, the chase began. Team Too had one of the ground stations, which relayed APRS info to aprs.fi. Consistent radio contact with payload train two was maintained for a majority of the chase. Due to the very strong winds, the balloon maintained a lead on the chase team for the entire flight and reached speeds upwards of 100 mph. Payload train 1 was recovered first at around 5:44 PM. Payload train 2 was recovered shortly after at 6:06 PM.

Once the balloons were recovered, everyone returned to the SRB and unpacked the components. MBuRST’s three trackers and all of the ENGR 100 payloads were also returned.

8 Data Collected

The payload successfully collected data during the flight. The data was collected from aprs.fi and the SD card and post-processed, and will be shown below. Section 8.1 shows

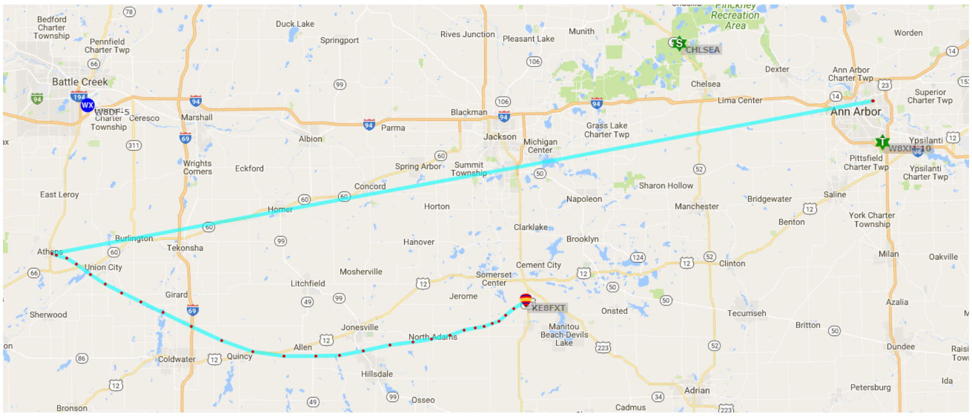


Figure 12: MicroTrak transmissions to APRS during flight.

the path taken by the balloon throughout the flight. Section 8.2 will discuss the data that follows. The plots of the collected data are shown in Appendix A, Figure 16 to Figure 27, because of the quantity of space they occupy. This includes collected temperature data, humidity data, battery voltage data, barometer data, and estimated altitude data. It will also show the data collected from the IMU, and an estimate of Euler angles from the IMU. The data from the GPS is shown in Appendix A as well, though it is mostly not useful. It will also show a set of pictures collected from the cameras in the payload.

8.1 Flight Path

Figures 12, 13, and 14 show APRS tracks from aprs.fi from the trackers onboard the balloon. Several features in these tracks are noteworthy:

- The MicroTrak antenna (Figure 12) was lost partway during the flight, explaining the lack of packets partway through the flight.
- The payload Trackuino (Figure 14) did not acquire GPS lock until roughly ten minutes into the flight. However, that GPS lock was shaky - it is offset from the MBuRST Trackuino (Figure 13). Proper lock was acquired about ten minutes before landing.
- The simulated balloon track was quite similar to the predicted track in Figure 11. The predicted landing spot was northeast of Britton, near the intersection of Day Rd and Far Rd. The actual landing site was southwest of Britton, off Samantha Dr (near Sutton Rd). The straight line distance between the actual and predicted landing locations was about 9 miles.

A three dimensional image of the flight path was also generated in Google Earth using data from the MBuRST and payload Trackuinos, shown in Figure 15. This image clearly shows where the balloon entered the jetstream (much wider spacing between APRS transmissions,

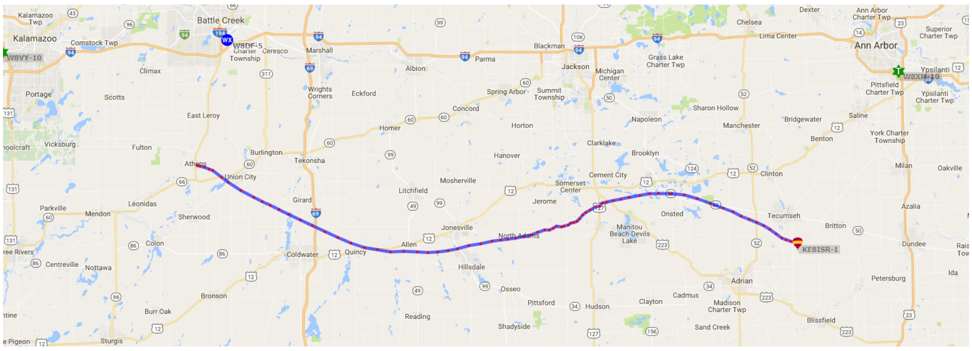


Figure 13: MBuRST Trackuino transmissions to APRS during flight.

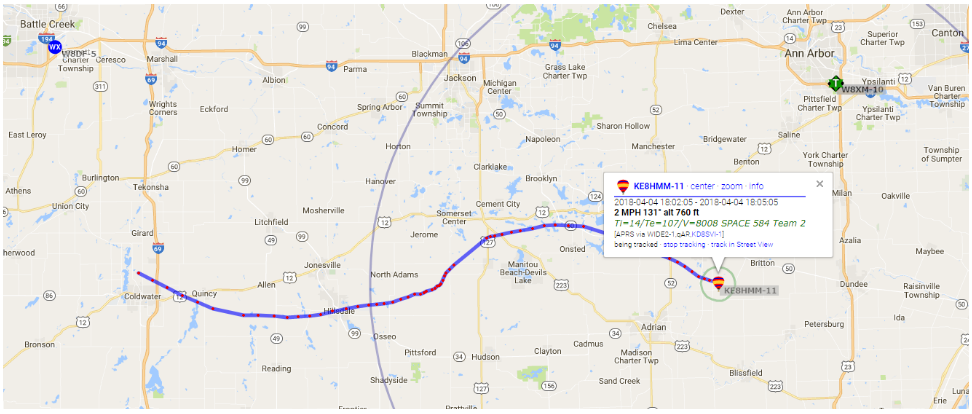


Figure 14: Team Too Trackuino transmissions to APRS during flight.

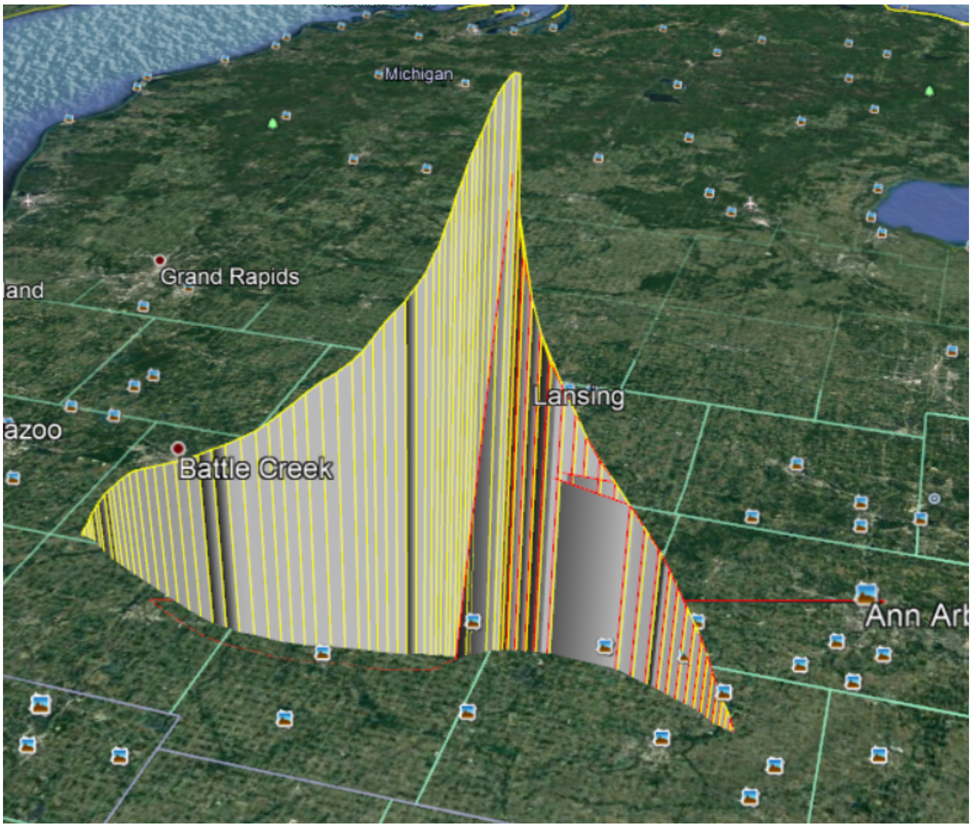


Figure 15: Flight track overlaid on Google Earth. MBuRST Trackuino is in yellow, and Team Too Trackuino is in red.

suggesting a high speed), and the location of balloon burst. In addition, it also shows the differences between the two Trackuinos' GPS locations up until the final descent, when they start to coincide.

8.2 Data Analysis

This section will provide an overview of the interpretation of each set of collected data. The plots below highlight the data in each stage of flight, including the ascent phase, the descent phase, and the landed phase. The transition from ascent to descent phase was determined as the point of absolute minimum pressure. The transition of descent to landed was determined from a break in data logging when the payload lost power for roughly three seconds as it rolled on the ground.

Figure 16 describes the internal and external temperatures logged during the flight through the various stages of the flight. The internal temperature behaves as expected, where it stays roughly constant during the ascent, drops during the descent, and then rises back up after landing. The external temperature drops rapidly during ascent, also drops during descent, and then rises back to the ambient starting temperature. There is an unexpected event where the external temperature rises during the second half of the ascent phase. It is believed that this happens as the self-heating effect from powering the thermistor to generate a voltage begins to dominate the effect of the cooler air as the air density drops, decreasing the heat that can be carried away by convection. Convection also explains the temperature drop during descent: a large airspeed in cool air results in a relatively large amount of heat being carried away from the payload.

Figure 17 describes the drop of the payload battery voltage over time. It is mostly not noteworthy, beyond the observation that the the timing of the heaters can be seen in the voltage dropout from the increased current draw.

Figure 18 describes the collected humidity data. It shows that the humidity drops to zero during ascent, stays at zero for most of the flight, rises quickly during descent as condensation forms, and then drops back down to the starting ground humidity.

Figure 19 shows the pressure during the flight. It shows as expected that the pressure drops during the ascent phase, has a minimum of roughly 18 mbar, and then increases faster back to the starting value during the descent phase. The pressure data has several artifacts in it, which are believed to be a result of the algorithm provided in the sensor documentation. There are many places in the algorithm where a small rounding or typecasting error could compound into a much more significant error.

Figure 20 shows the estimated and measured altitudes during the flight from GPS data and derived barometer data. The barometer data agrees with APRS logs, which showed an apogee of nearly 90 000 ft. The altitude from the barometer is generated using barometric formulas from the US Standard Atmosphere (1976). As can be seen, the altitude from the GPS seems to be mostly inaccurate.

Figure 21 shows the measured acceleration over time on 3-axes. The plots highlight 0 g, -1 g, and 1 g, which are useful resting accelerometer values. The plot shows several interesting events. The first is around 40 minutes into the ascent, where the winds pick up and cause significantly oscillation. This lasts for nearly 5 minutes, and then it calms significantly. This is presumably when the balloon enters the jet stream. Soon after, the values of the accelerations seem to imply that the payload train is nearly sideways, which the video seems to corroborate. The next is at roughly 53 minutes at the point of balloon burst. Following the more relatively calm point just discussed, it then enters a much more aggressive acceleration pattern. Next, the point of descent at around 80 minutes, roughly 3 seconds of data was lost on landing. This unfortunately included the most interesting data at this stage, the accelerations on the actual ground collision. Finally, it is evident that the balloon was found at 105 minutes when a new acceleration event begins.

Figure 22 shows the measured 3-axis angular velocity over time of the payload. It generates nearly the same conclusions as those discussed in the previous section. One portion of extra note is at roughly 45 minutes when the x-axis shows a near constant value for a few minutes, nearly at the same time as the acceleration looks rotated. It does make sense that for there to be a constantly different acceleration, it would have to be induced by some kind of constant rotation from the wind.

Figure 23 shows the measured magnetometer data. This data is mostly useless visually, but can be used in attitude determination algorithms.

Figure 24 shows Euler angles generated from a a Mahony filter, a commonly implemented attitude determination algorithm (code in [1]). Unfortunately, from looking through the data and from collective team experience with this particular type of attitude determination, the sampled data appears to be too noisy to get a good result from the algorithm. From observing the recorded video, the motion also appears to be too fast and oscillatory to get a reasonable result using this type of algorithm without better inertial sensors and a better sensor configuration.

Figure 25 shows the number of available GPS satellites during the flight. Given what seems to be a constant gain and loss of satellites, it seems to make sense that the GPS data is mostly useless.

Figure 26 shows the GPS velocity recording. Most of the data appears to be lost or incorrect. Figure 27 shows the GPS course heading, which gives the same conclusion.

Data was also collected from both of the cameras. Several pictures from the cameras are included in Appendix A. The downward facing camera collected video for the entire flight. The side facing camera only collected data for a part of ascent as the battery died about ten minutes from lift-off (most of the power was expended waiting to launch). Figure 28 shows the side facing camera about 5 minutes from leaving the ground. It is just about to enter the clouds, and snow flakes can be seen in the image. Figure 29 is a second picture from the side facing camera, just before it died about ten minutes into the flight, showing it above the clouds. Figure 30 is an image from the downward facing camera about forty five minutes into flight, near the balloon burst. It shows that the payload train was nearly sideways at this point in the flight. Figure 31 shows an image from the downward facing camera about an hour and twenty minutes into the flight, just before landing. The videos of the flight can be viewed from Google Drive. The downward facing video can be found here, and the side facing video can be found here.

9 Issues Encountered

Various problems were encountered throughout the course of the project. Each issue was addressed and resolved in a timely fashion to ensure a successful launch.

9.1 Communications

There was difficulty with the ground station during the initial attempts to conduct a car chase. Team Too members were able to successfully troubleshoot the ground station and get it working for a second attempt. Further discussion of the solutions that were implemented is provided above in section 6.6.

9.2 PCB

There were three major issues with the PCB, which are described below.

The most critical mistake was with the barometer's communication protocol. The intention was to communicate with the barometer over SPI, however, the barometer pin which controls communication protocol configuration was set to the wrong voltage which put the barometer into I^2C mode. To fix this mistake, the barometer was lifted off the PCB and connected a thin wire underneath the chip to the correct voltage. The chip was then hot glued to the PCB for structural support.

The next mistake was an issue with the battery connection. It was unclear in the documentation which parts of the battery connector were attached to the battery terminals, and which were for mounting. They were swapped incorrectly in the design, and had to be corrected with external wires.

The other major mistake regarding the PCB was the FTU circuit. The FTU circuit was initially built into the payload PCB, however, it was later realized that the FTU needed to be in its own package. To fix this, an independent FTU was implemented, but the board was still left with wasted weight and board space. This mistake did clear up a critical misunderstanding of the payload train, which was important for subsequent labs and launch. Eventually, this turned out to be a positive, as the second heater was able to use the FTU connector for power.

9.3 Launch

The major issue during launch was with ground station logistics. During the first launch date, there were two ground stations, one with the MBuRST team and one with Team Too. Once the first balloon was launched, one of the ground stations was asked to begin pursuit, however, both ground stations needed to stay for the launch of the second balloon. MBuRST was using a proprietary tracker, which could only track their payload on the second balloon, while Team Too needed to stay with the second payload train until launch. As a result, neither ground station left with the first balloon. To fix this in the future, ground station logistics and chase car assignments need to be determined prior to balloon launch.

10 Conclusion

In conclusion, the team was able to successfully design, build, and fly a custom payload on a high altitude balloon. A printed circuit board was designed and integrated into a custom box, subjected to thermal and mechanical shock tests, and was successfully flown and recovered. The payload successfully recorded inertial, pressure, GPS, humidity, and temperature data for the duration of the flight. All trackers worked well after initial troubleshooting and every payload on the train was recovered. Overall, the team learned a great deal from this project and had fun doing so.

References

[1] Open source IMU and AHRS algorithms, x io Technologies. July 31, 2012. http://x-io.co.uk/open-source-imu-and-ahrs-algorithms/

[2] Invensense MPU-9250 SPI Library, Brian Chen. May 16, 2017. https://github.com/brianc118/MPU9250

[3] MPU-9250 Product Specification Revision 1.1, Invensense. June 20, 2016. https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf

[4] HDC1080 Low Power, High Accuracy Digital Humidity Sensor with Temperature Sensor, Texas Instruemnts. January 2016. http://www.ti.com/lit/ds/symlink/hdc1080.pdf.

[5] Arduino Library for ClosedCube HDC1080, ClosedCube. February 14, 2018. https://github.com/closedcube/ClosedCube_HDC1080_Arduino

[6] MS5607-02BA03 Barometric Pressure Sensor, with stainless steel cap, TE Connectivity, June 2017. http://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=MS5607-02BA03&DocType=Data+Sheet&DocLang=English

[7] Low Voltage Temperature Sensors, Texas Instruments. 2015. http://www.analog.com/media/en/technical-documentation/data-sheets/TMP35_36_37.pdf

[8] TinyGPSPlus, Mikal Hart. February 3, 2018. https://github.com/mikalhart/TinyGPSPlus

A Data Collected

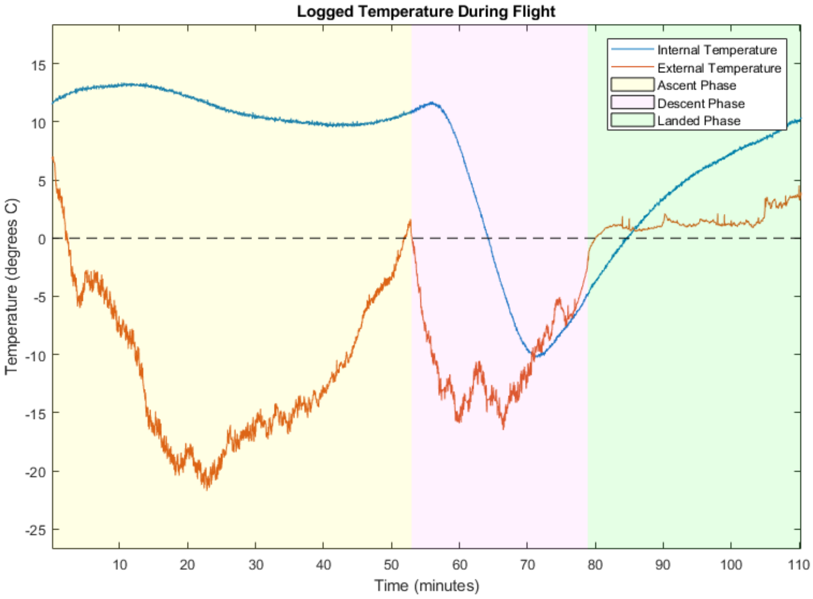


Figure 16: Temperature during flight

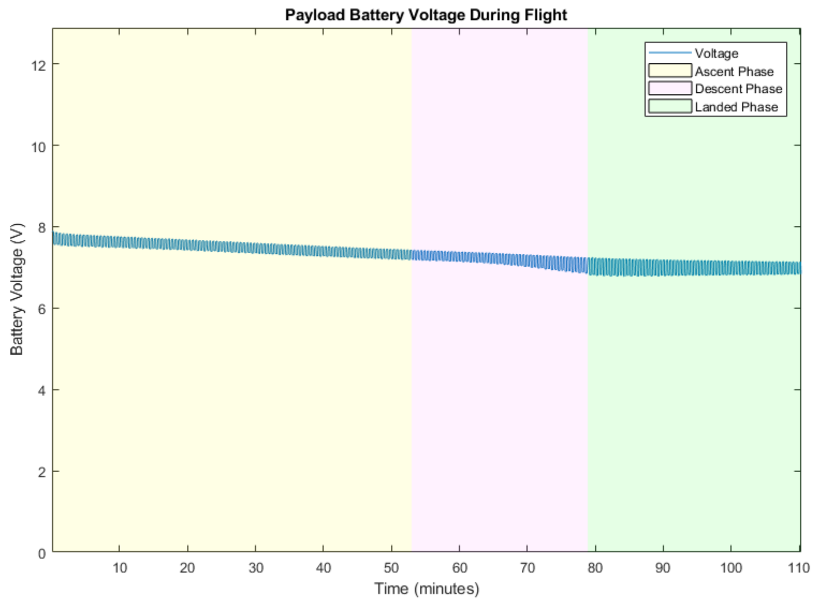


Figure 17: Battery during flight

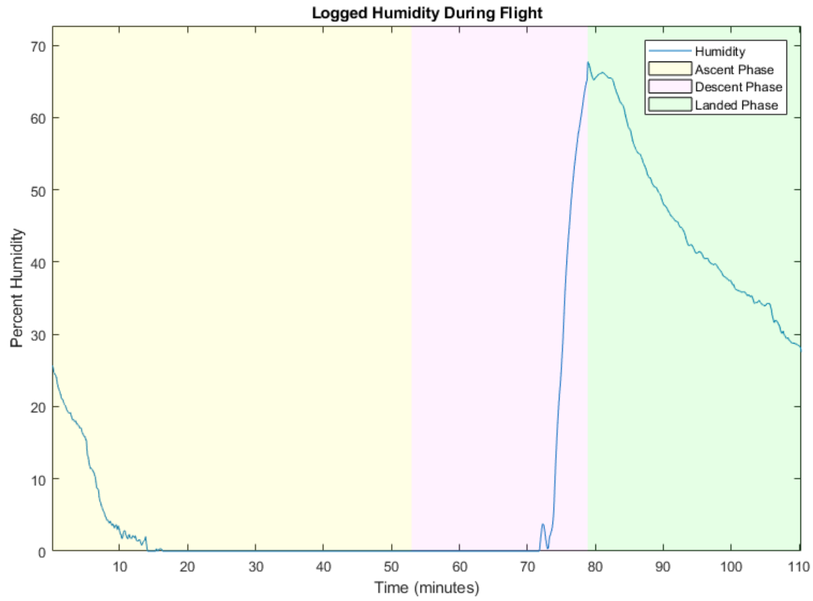


Figure 18: Humidity during flight

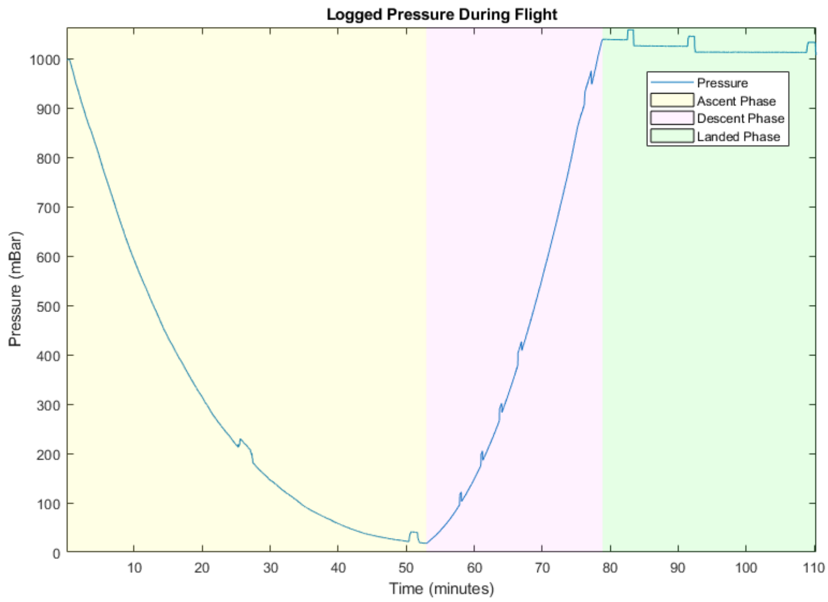


Figure 19: Pressure during flight

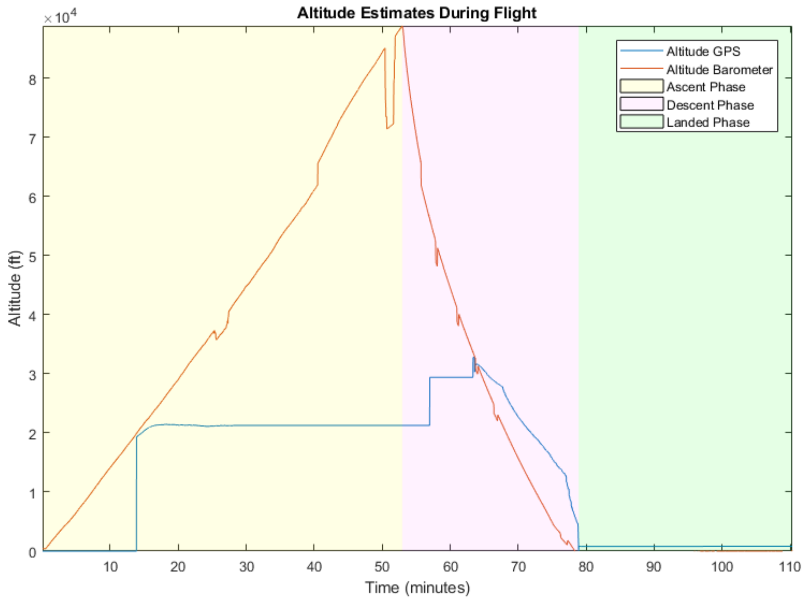


Figure 20: Altitude during flight

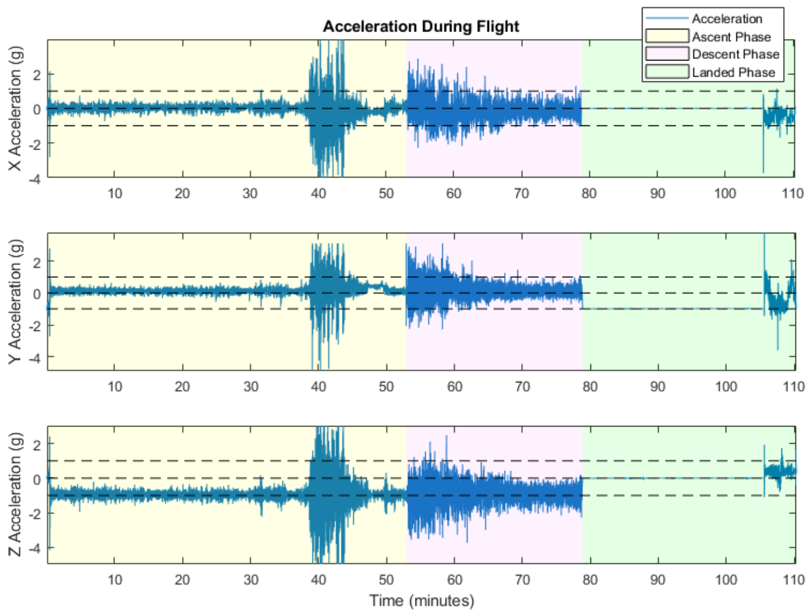


Figure 21: Acceleration during flight

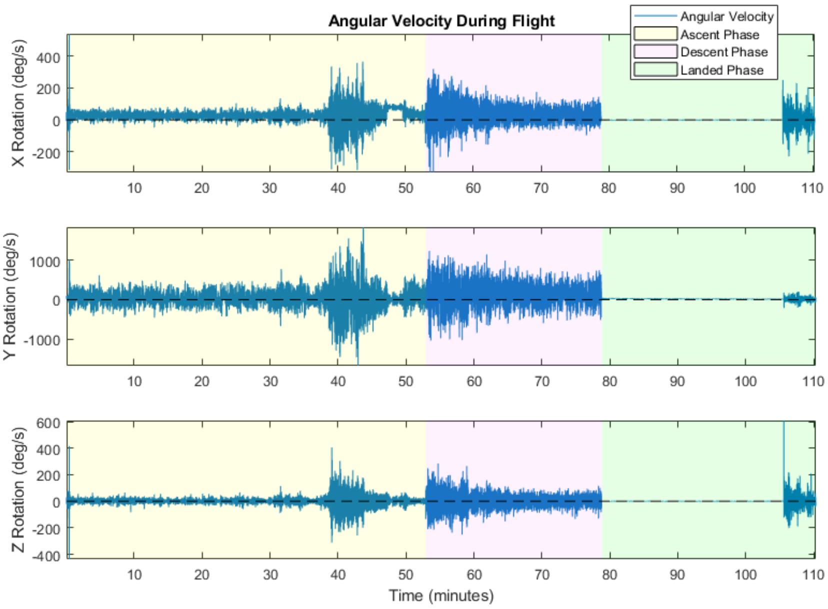


Figure 22: Gyroscope during flight

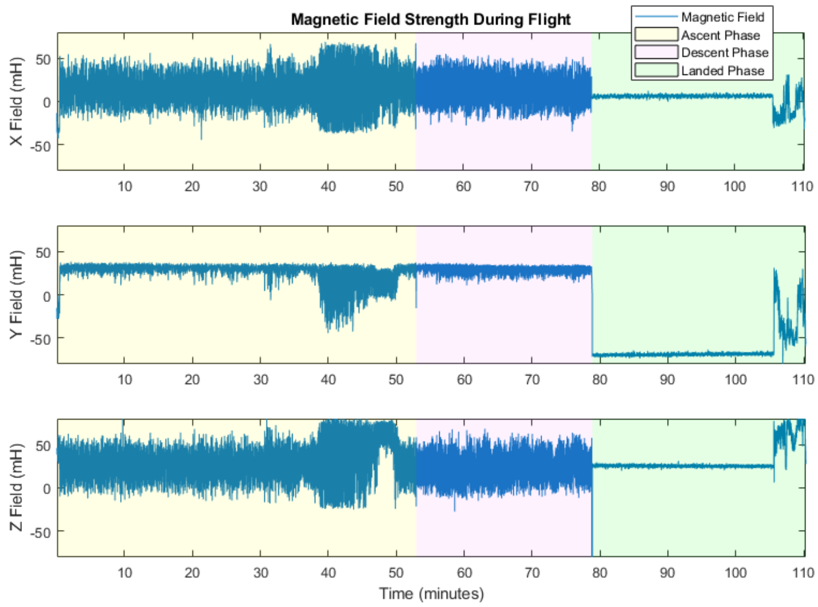


Figure 23: Magnetometer during flight

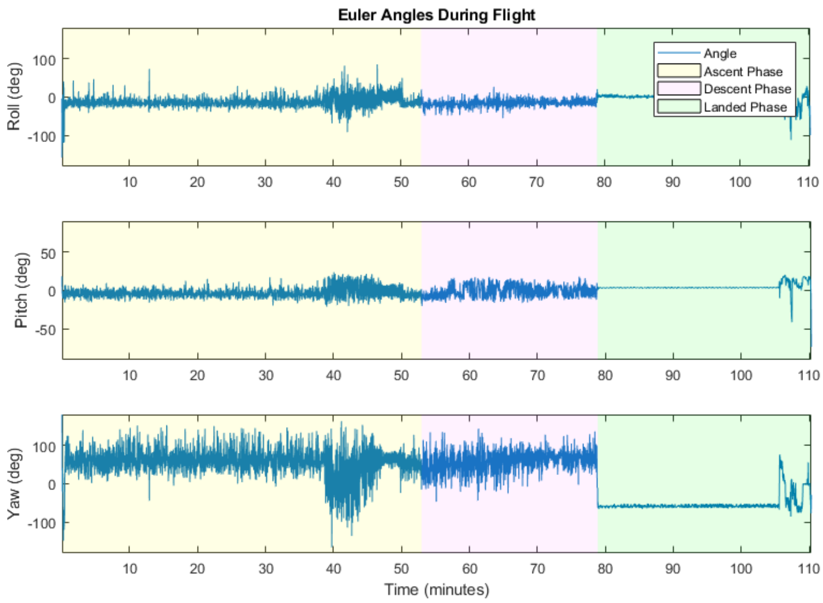


Figure 24: Euler angles during flight

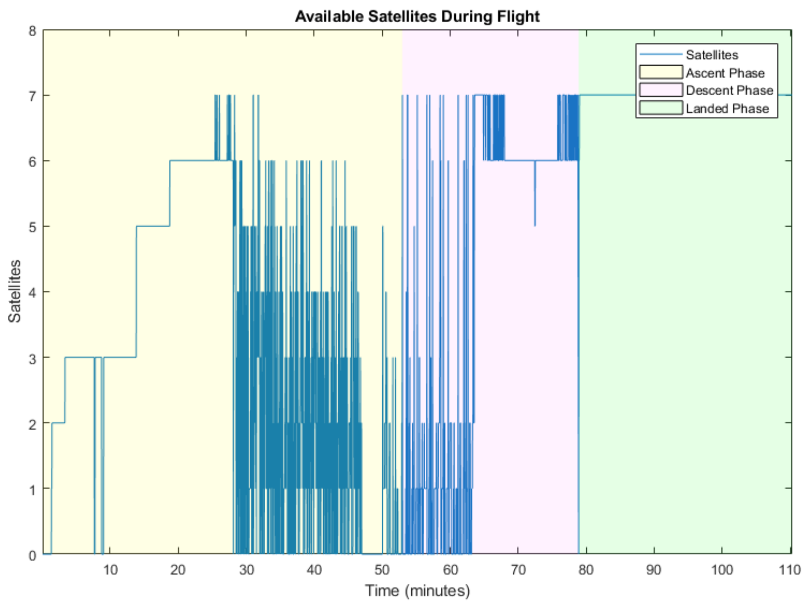


Figure 25: Satellites during flight

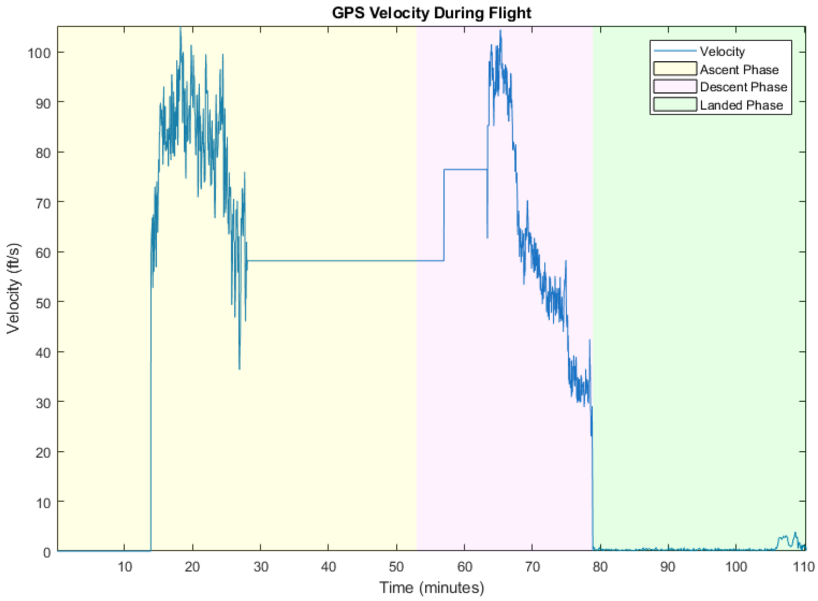


Figure 26: GPS Velocity during flight

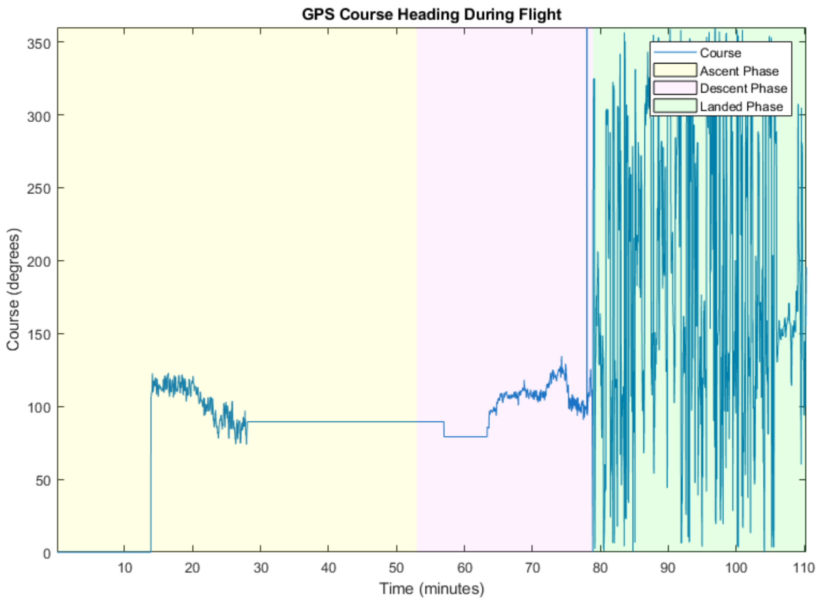


Figure 27: GPS Course during flight

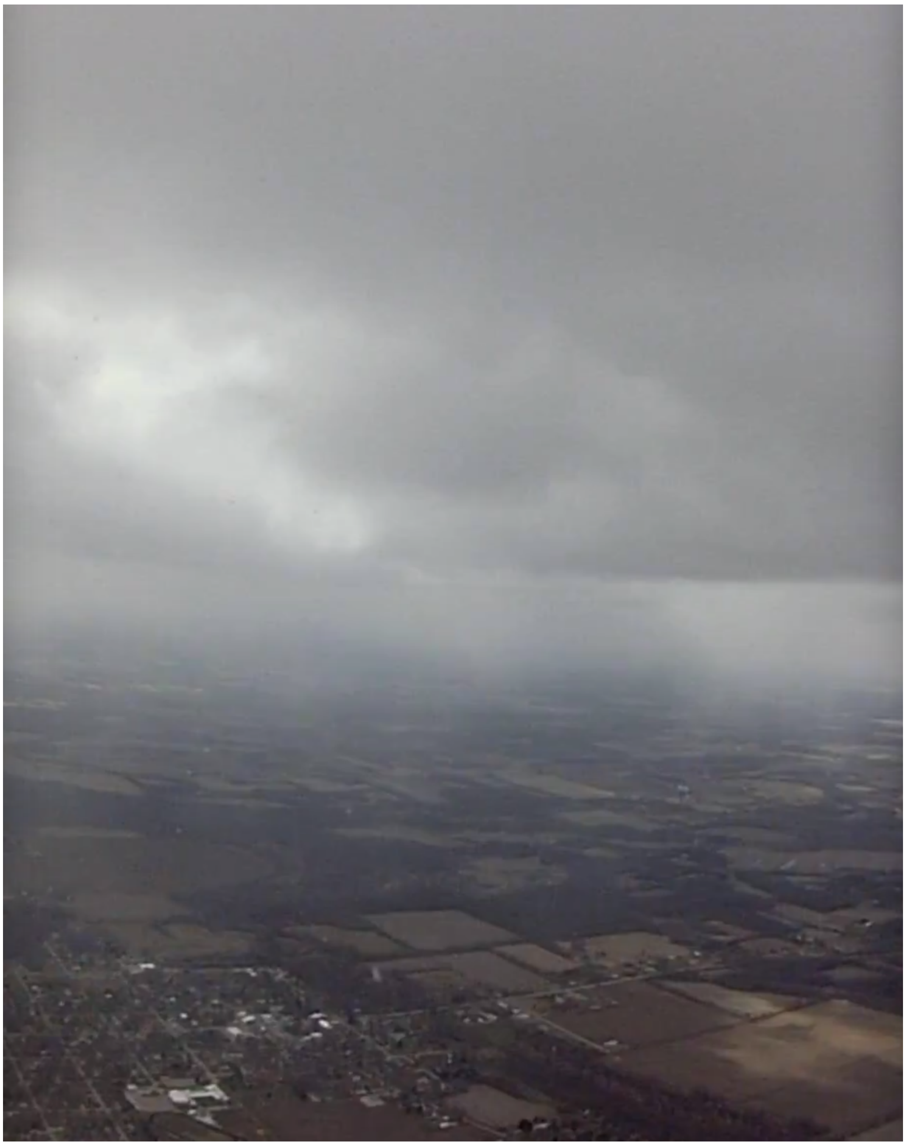


Figure 28: Snow flakes in the side-facing camera, roughly 5 minutes into flight

Figure 29: Side facing camera, roughly 10 minutes into flight

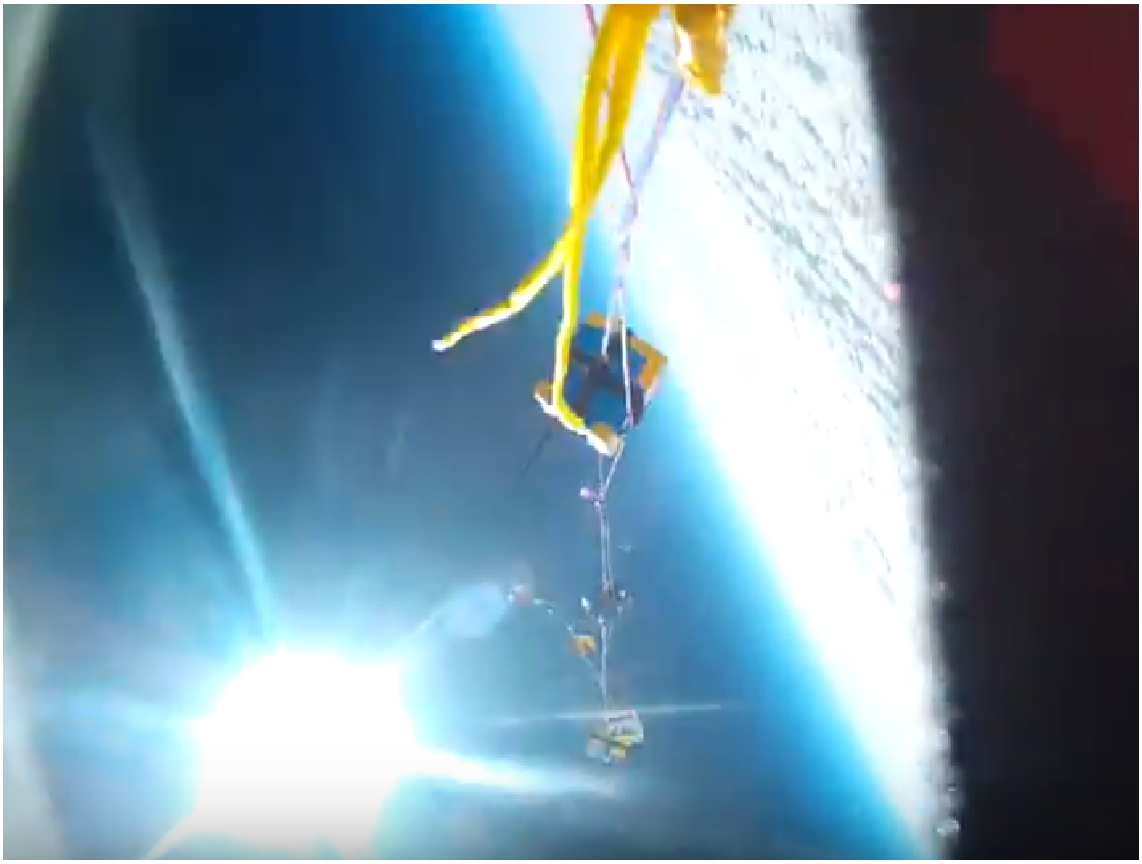


Figure 30: Down-facing camera in high winds, roughly 45 minutes into flight

Figure 31: Down-facing camera just before landing, roughly 1 hour 20 minutes into flight

B Printed Circuit Board

B.1 Hardware Architecture

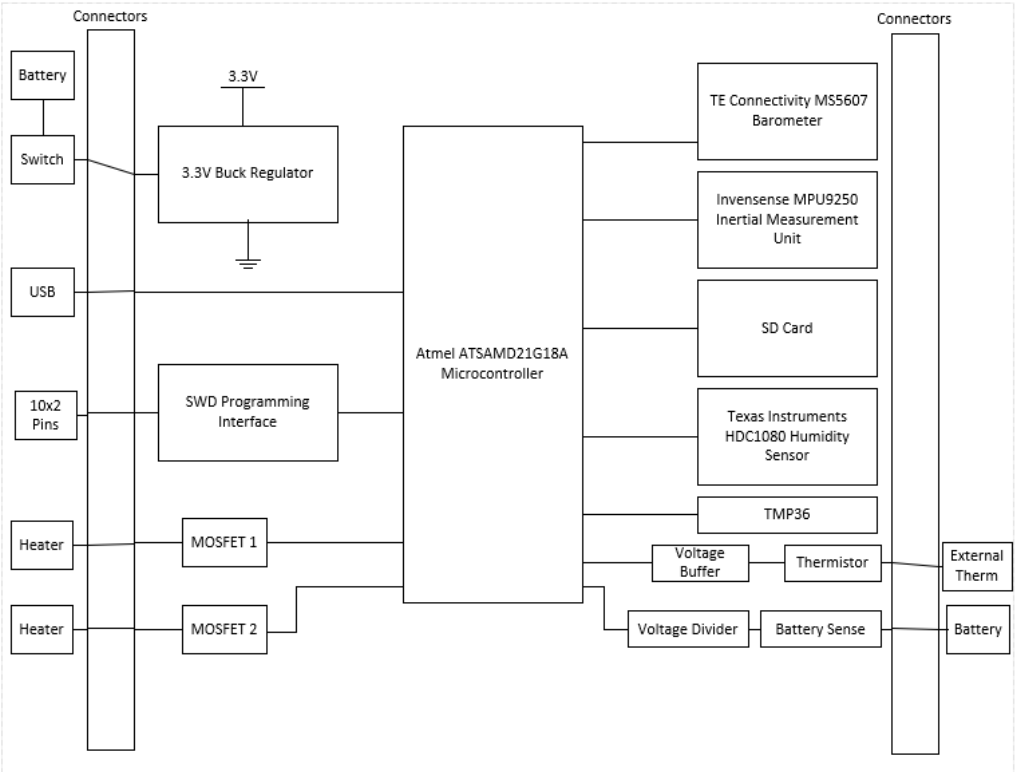


Figure 32: Hardware Architecture Diagram

B.2 Printed Circuit Board Schematic

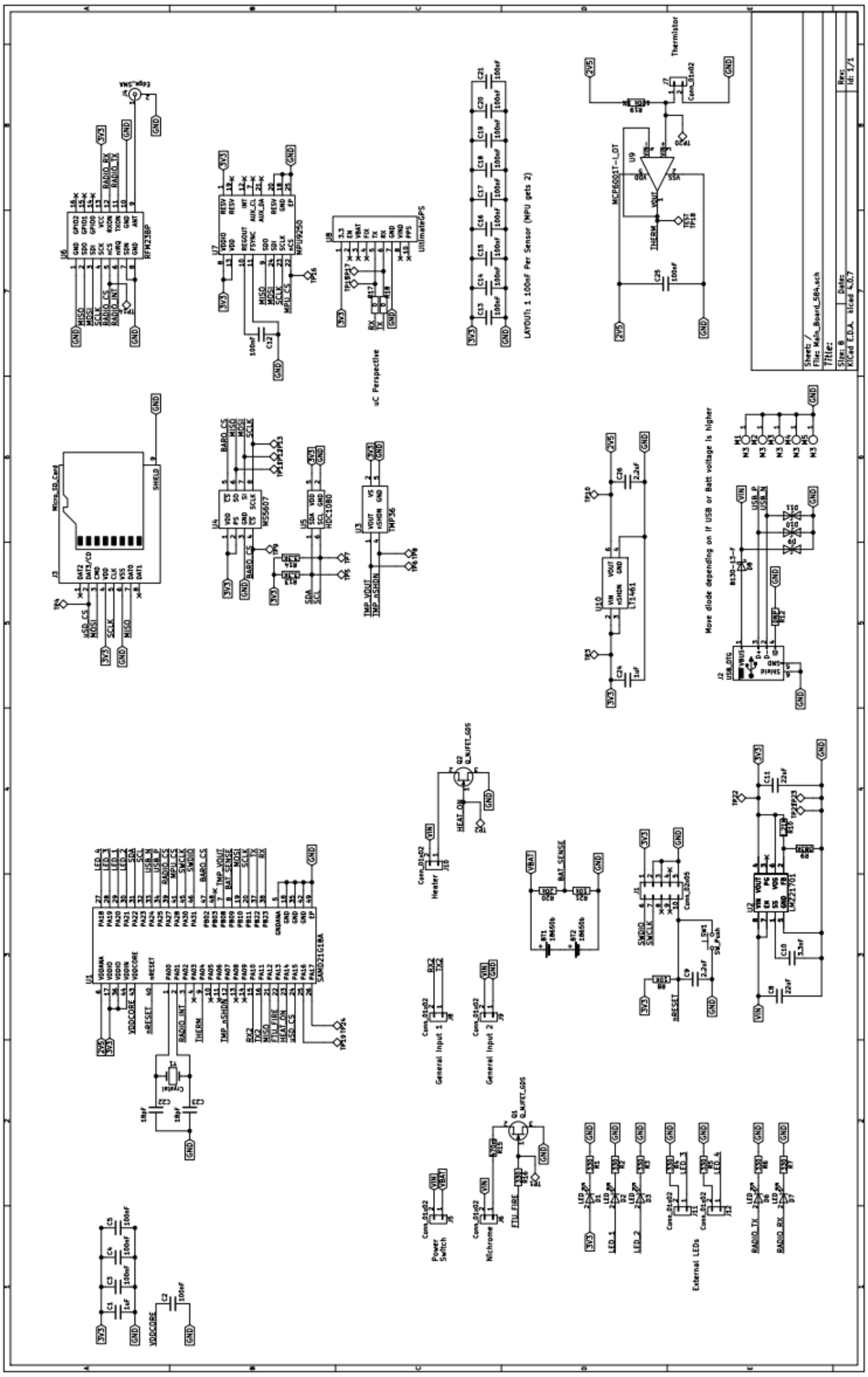


Figure 33: Schematic

B.3 Printed Circuit Board Top Layer

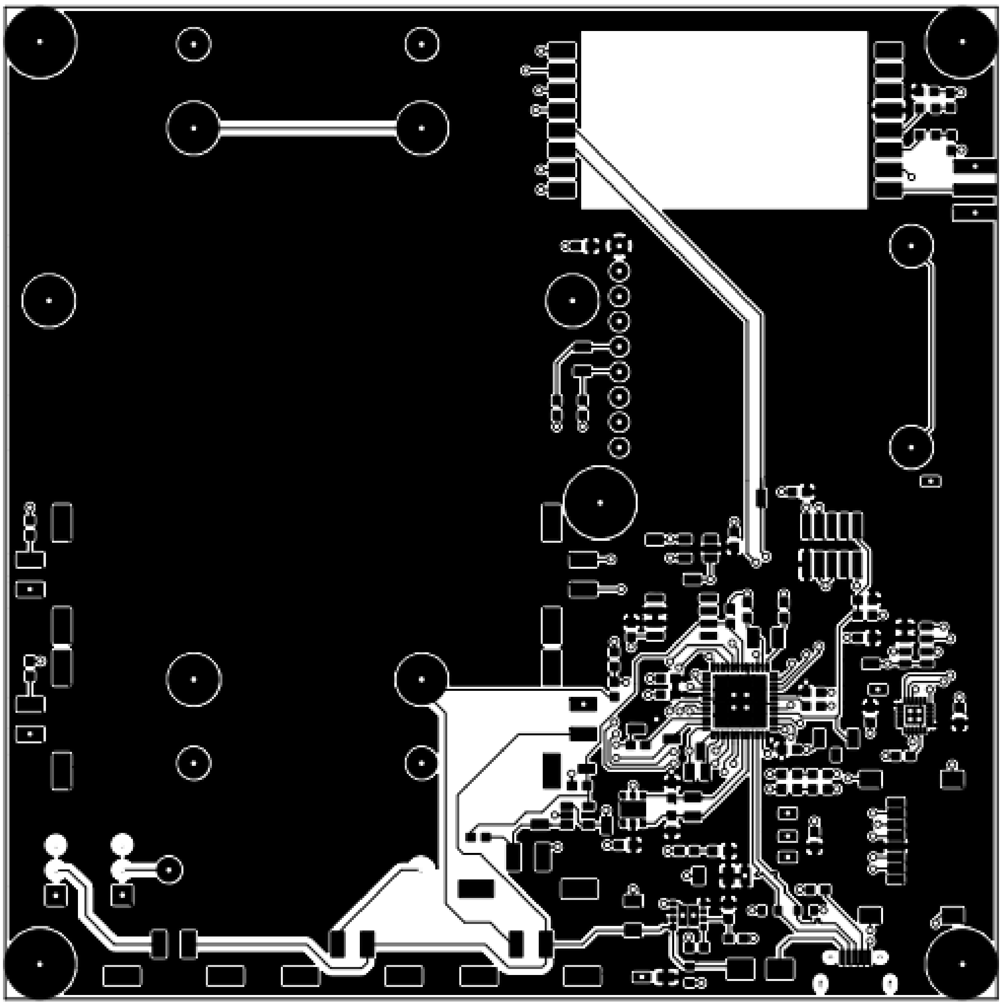


Figure 34: Top Layer

B.4 Printed Circuit Board Bottom Layer

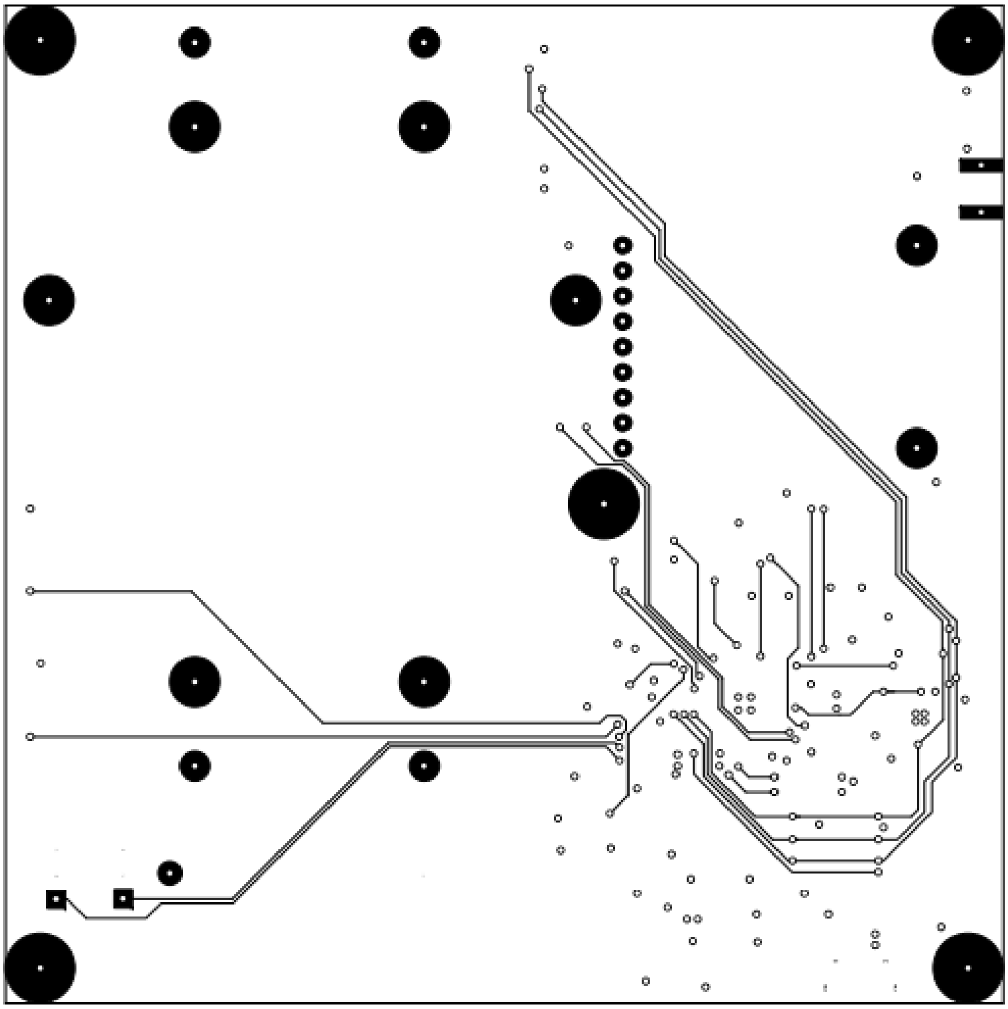


Figure 35: Bottom Layer

C Launch SOP

Day Before

Materials

- White Trackuino battery
- White FTU battery
- Black payload batteries (x2)
- Ground station battery
- microSD card (x3)

Procedure

- Fully charge Trackuino battery
- Fully charge both payload batteries
- Fully charge both cameras
- Fully charge ground station battery
- Wipe all microSD cards
- A66 camera microSD card
- Not A66 camera microSD card
- Payload microSD card
- Install microSD cards in payload board and both cameras

Day Of

Materials

- FTU
  - ☐ White FTU battery
  - ☐ FTU protoboard
  - ☐ FTU box
  - ☐ FTU lid
  - ☐ Nichrome
- Payload
  - ☐ Black payload batteries (x2)

  - ☐ Payload board
  - ☐ Trackuino
  - ☐ White trackuino battery
  - ☐ Cameras (x2)
  - ☐ microSD card (x3)
  - ☐ Payload box/lid
  - ☐ Payload internal support
  - ☐ Heater (x2)
  - ☐ Trackuino GPS antenna
  - ☐ Trackuino APRS antenna with SMA cable
  - ☐ Velcro straps
  - ☐ Duct tape
  - ☐ Small zip ties
  - ☐ Spare velcro
- Tools
  - ☐ Crescent wrench for helium bottle
  - ☐ Allen keys
  - ☐ Screwdriver for electronics
  - ☐ Multimeter
  - ☐ Pliers
  - ☐ Scissors
  - ☐ Duct tape
  - ☐ Electrical tape
  - ☐ Masking tape
  - ☐ Hot glue gun
  - ☐ Hot glue
  - ☐ Scale
  - ☐ Solder
  - ☐ Soldering iron

- ☐ Wire
- ☐ Wire cutters
- ☐ Wire strippers

- Payload support
  - ☐ Payload battery charger
  - ☐ FTU/Trackuino battery charger
- Train
  - ☐ Balloon
  - ☐ Parachute
  - ☐ Radar reflector
  - ☐ Zip ties (for balloon)
  - ☐ 10 paracord lines for connecting components
  - ☐ 20 carabiners/clips
- Balloon filling
  - ☐ 4x bottles (two balloons)
  - ☐ Regulator/fill valve
  - ☐ Dolly
  - ☐ Crescent wrench
  - ☐ Filling hose
  - ☐ Vinyl gloves (x4)
  - ☐ Leather gloves for string (¿=3)
  - ☐ Kite nylon rigging string
  - ☐ Pre-made rigging harnesses, with key rings (¿=6)
  - ☐ Key rings (x24)
  - ☐ Ground tarp
  - ☐ Balloon tarp
  - ☐ "House"
  - ☐ Zip ties
  - ☐ Full roll of duct tape

  - ☐ Safety glasses (x3)
- Ground Station
  - ☐ Kenwood TM-D710 radio display/transceiver
  - ☐ Laptop/charger
  - ☐ Inverter
  - ☐ APRS antenna
  - ☐ GPS antenna
  - ☐ Cabling for radio to laptop (DIN8 to DB9, serial to USB converter, USB cable)

Payload/Trackuino Assembly

- ☐ Place batteries in payload board, taking the utmost care to install them in the correct orientation. This should be verified by a second person.
- ☐ Add zipties around the batteries.
- ☐ Place the payload board in the box. Hot glue may be needed to attach the hook side of the Velcro to the box. Push the board down on the Velcro, and wiggle it around laterally. Gently pull up on the board to verify that it is secure.
- ☐ Turn on cameras and start recording. Install them in the payload box. The one on the side should be installed first.
- ☐ Wrap heaters snugly around cameras.
- ☐ Attach connectors to the board.
  - ☐ One heater to the HEAT connector
  - ☐ One heater to the FTU connector
  - ☐ Thermistor to the TMP connector
  - ☐ Switch connector with two wires to the SWITCH connector
  - ☐ Switch connector with one wire to the TP connector
- ☐ Route wiring around edges of the box, and tape to walls.
- ☐ Attach the Trackuino to the lid, following the Velcro pattern.
- ☐ Install the buzzer in the Trackuino, taking care to match the polarities. Tug gently to ensure that it is secure.
- ☐ Attach the GPS antenna to the red GPS module on the Trackuino.
- ☐ Attach the APRS antenna to the SMA connector on the Trackuino board through an SMA coaxial cable.

- ☐ Attach the white Trackuino battery to the Trackuino.
- ☐ Slide the Trackuino battery into the slot in the payload internal support.
- ☐ Slide the Trackuino GPS antenna into the slot in the top of the payload internal support.
- ☐ Slide the Trackuino APRS antenna into the hole at the bottom of the payload box.
- ☐ Gently slide the internal support into the payload box over the cameras/heaters, being careful to not press any of the camera buttons. Keep the lid close to avoid ripping wires out of the Trackuino.
- ☐ Pull the APRS antenna through the hole until most of the antenna is outside. Tape the SMA coaxial cable to the side of the payload box.
- ☐ Place the lid on the box.
- ☐ Wrap tape around the base of the APRS antenna.
- ☐ Wrap the box with Velcro straps in both directions.
- ☐ Hold the box with the APRS antenna facing downwards until the Trackuino buzzer starts to beep, indicating GPS lock.
- ☐ The payload is now ready for integration with the payload train.

D Flight Code

D.1 Flow Diagram

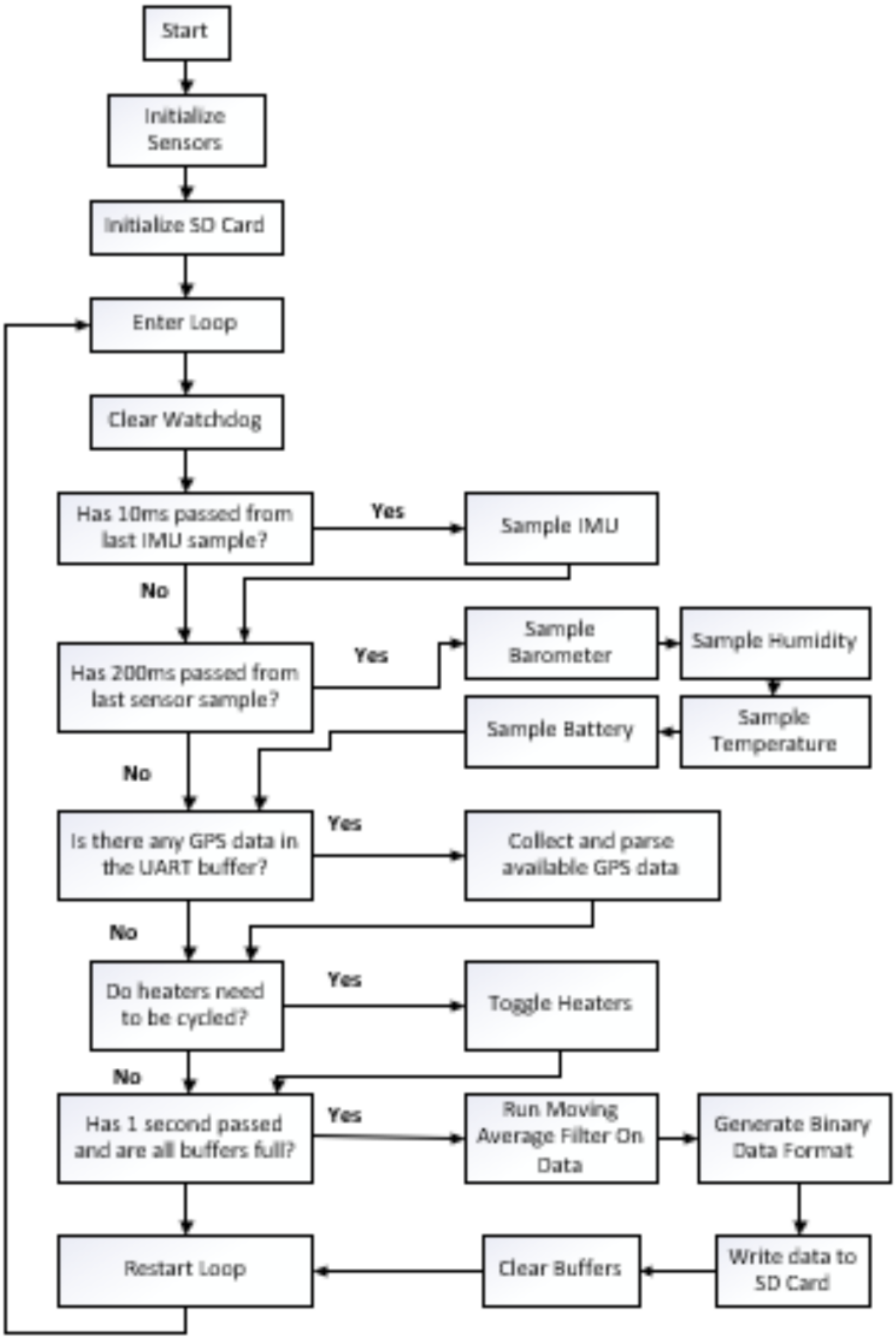


Figure 36: Software Flow Diagram

D.2 Payload Code

```
/*
 * flight_code.ino
 * This file contains the Arduino main program code to handle in-flight tasks
   .
 *
 * Tasks to complete:
 * Sample MPU9250: 10Hz, SPI, Circular Buffer
 * Sample GPS: 1Hz, Serial, Processing Intense Task
 * Sample Humidity: 5Hz, I2C, MAF, Circular Buffer
 * Sample TMP36: 5Hz, ADC, MAF, Circular Buffer
 * Sample Thermistor: 5Hz, ADC, MAF, Circular Buffer
 * Sample Barometer: 5Hz, SPI, Process, MAF, Circular Buffer
 * Sample Battery Voltage: 5Hz ADC, MAF, Circular Buffer
 *
 * Write Data to SD Card: 1Hz, SPI
 * Write or Read data to Radio 1Hz, SPI
 * Write or Read data to SD for reset handling (100mHz write, read on power
   on)
 *
 * Clear Watchdog: Handles crashes. Call whenever possible.
 * Enable/Disable FTU from timer or command (use RTC for timer), 1Hz
 * Enable/Disable Heater from timer or TMP36 feedback (use RTC for timer), 1
   Hz
 * Enable/Disable Status LEDs, Lowest Priority
 */

//
  ___________________________________________________________________________

//Includes
#include <SPI.h> //Built in Arduino SPI library

//The standard Arduino SD library is actually built on this library
//Using the smaller version to reduce overhead and speed up write speeds
//Library ZIP included in git repo
//Install with Sketch>Include Library>Add .ZIP Library
#include <SdFat.h> //Installed SD Library

//File with #define constants
#include "flight_config.h"

#include "src/tinygps/TinyGPS++.h" //Downloaded GPS library
#include "src/mpu9250/MPU9250.h"    //Downloaded IMU library
#include "src/hdc1080/ClosedCube_HDC1080.h" //Downloaded humidity sensor
   library
#include "src/rtc/RTCZero.h" //Arduino official library for real-time clock
//End Includes
//
  ___________________________________________________________________________

```

```
//
    ____________________________________________________________________________
//Macros
#ifdef SERIAL_DEBUG
#define debug(X) SerialUSB.println(X)
#else
#define debug(X) do {} while(0) //compiles to noop or optimized out
#endif
//End Macros
//
    ____________________________________________________________________________

//
    ____________________________________________________________________________
//Global Objects
ClosedCube_HDC1080 hdc1080; //I2C Humidity sensor
MPU9250 mpu(MPU_SPI_CLOCK, MPU_SS_PIN); //SPI IMU
TinyGPSPlus gps; //Serial GPS
RTCZero rtc; //Internal Real-Time Clock

SdFat sd; //SD handler
SdFile log_file;    //Log file handler
SdFile config_file; //Config file handler
//End Global Objects
//
    ____________________________________________________________________________

//
    ____________________________________________________________________________
//Structs
//Stucture for data to log on next SD call
data_to_log data;

//Structure to read and write from on power on or state log
config_log config;
//End Structs
//
    ____________________________________________________________________________

//
    ____________________________________________________________________________
//Global Variables

//Buffer pointers
uint8_t mpu_offset = 0;
uint8_t slow_offset = 0;
```

```

//Buffers to store data
//IMU buffer stored in log data structure
float32_t humid[SLOW_SAMPLE_RATE];
uint32_t tmp[SLOW_SAMPLE_RATE];
uint32_t therm[SLOW_SAMPLE_RATE];
float32_t baro[SLOW_SAMPLE_RATE];
uint32_t batt[SLOW_SAMPLE_RATE];

//Current state of FTU and heater
uint32_t ftu_state = 0;
uint32_t heat_state_1 = 0;
uint32_t heat_state_2 = 0;
//Last stored temp, for feedback heating
float32_t last_temp = 0;

//Assorted timers for spacing function execution time
uint32_t counter;
uint32_t last_call;
uint32_t last_mpu;
uint32_t last_slow;
uint32_t last_gps;
uint32_t this_call;
uint32_t last_heat;
uint32_t last_ftu;
uint32_t last_config;
uint32_t last_led;
uint32_t ftu_start = 0;

//Trigger for logging to the SD card
uint16_t ready = 0x00;

//Status of the SD card after opening
uint8_t sd_status = 0;

//Stores barometer configuration settings
uint16_t baro_prom[8];

//Counter for time left until FTU trigger
uint32_t ftu_ms_remain = TWO_HOURS_MS;


//End Global Variables
//
    ____________________________________________________________________________________


//
    ____________________________________________________________________________________

//Function Declarations
void init_spi();
void init_mpu();
```

```
void init_gps();
void init_humid();
void init_baro();
void init_tmp();
void init_sd();
void init_radio();
void init_config_file();
void init_rtc();
void init_leds();
void init_ftu();
void init_heater();

void init_watchdog();

void clear_watchdog();

//End Function Declarations
//________________________________________________________________________________

//________________________________________________________________________________
//Program Setup
void setup() {
  #ifdef SERIAL_DEBUG
  SerialUSB.begin(115200);
  while(!SerialUSB){}
  #endif
  debug("Here we go");
  debug("Starting init");

  //Assorted initialization functions for each feature
  init_spi();
  init_leds();
  init_ftu();
  init_heater();
  init_mpu();
  init_gps();
  init_humid();
  init_baro();
  init_tmp();
  init_sd();
  init_radio();
  init_config();
  init_rtc();

  init_watchdog();

  debug("Done with init");
  debug("Starting Task Creation");

```

```
  //Set up timers
  last_call = millis();
  last_mpu = last_call;
  last_slow = last_call;
  last_gps = last_call;
  this_call = last_call;
  last_heat = last_call;
  last_ftu = last_call;
  last_config = last_call;
  last_led = last_call;
  analogReadResolution(12);

}
//End Program Setup
//
  _____________________________________________________________________________

//
  _____________________________________________________________________________
//Program Loop
void loop() {
   this_call = millis(); //Gets current time
   clear_watchdog(); //Needs to happen very frequently or program resets

    //Triggers regular IMU logging
    if ((this_call - last_mpu) > MPU_SAMPLE_PERIOD && !(ready&0x01 << 0)){
        debug("MPU");
        debug(this_call);
        read_mpu();
        last_mpu = this_call;
    }

    //Triggers regular logging for non-IMU sensors
    if ((this_call - last_slow) > SLOW_SAMPLE_PERIOD && !(ready&0x01 << 1)){
        debug("SLOW");
        debug(this_call);
        read_humid();
        read_tmp();
        read_therm();
        read_batt();
        read_baro();

        slow_offset++;
        if (slow_offset==SLOW_SAMPLE_RATE ){
            slow_offset--;
            ready|=0b0000000000000010;
            update_humid();
            update_tmp();
            update_therm();
            update_baro();
            update_batt();
```

```
        }
        last_slow = this_call;
    }

    //Attempts to read GPS whenever possible
    //if ((this_call - last_gps) > 1000 && !(ready&0x01 << 2)){
    if (GPS_SERIAL.available()){
        debug("GPS");
        debug(this_call);
        read_gps();


        //last_gps = this_call;
    }

    //Clears wait for GPS if more than a second has passed
    //This is necessary to ensure IMU samples are not too delayed
    if ((this_call - last_gps) > 1000){
        ready|=0b0000000000000100;
        last_gps = this_call;

    }

    if ((this_call - last_led) > 500){
        update_leds();
        last_led = this_call;
    }

    //Check the FTU and heaters
    if ((this_call - last_ftu) > 100){
      //ftu_check();
      heat_check();
      last_ftu = this_call;
    }

    //Update the config file to protect from unexpected reset data loss
    if ((this_call - last_config) > 10000){
      write_config();
      last_config = this_call;
    }

    //Logs data to SD card and resets
    debug(ready);
    if (ready == READY_TO_LOG){
        debug("SD");
        debug(this_call);
        write_sd();
        ready = 0;

        mpu_offset = 0;
        slow_offset = 0;

        memset(humid, 0, sizeof(humid));
```

```
        memset(tmp, 0, sizeof(tmp));
        memset(therm, 0, sizeof(therm));
        memset(batt, 0, sizeof(batt));
        memset(baro, 0, sizeof(baro));
    }

}
//End Program Loop
//
    ______________________________________________________________________________________

//
    ______________________________________________________________________________________

//Program Initialization
void init_spi(){
    /*
    init_spi()
    This function starts the SPI bus and sets chip selects
    */
    SPI.begin();
    pinMode(BARO_SS_PIN, OUTPUT);
    digitalWrite(BARO_SS_PIN, HIGH);
    pinMode(SD_SS_PIN, OUTPUT);
    digitalWrite(SD_SS_PIN, HIGH);
    pinMode(MPU_SS_PIN, OUTPUT);
    digitalWrite(MPU_SS_PIN, HIGH);
    pinMode(RADIO_SS_PIN, OUTPUT);
    digitalWrite(RADIO_SS_PIN, HIGH);

}

void init_mpu(){
    /*
    init_mpu()
    This function loads config information to the IMU and
    runs an internal calibration
    */
    mpu.init(true);
    mpu.set_acc_scale(BITS_FS_4G);
    mpu.set_gyro_scale(BITS_FS_2000DPS);
    mpu.calib_acc();
    mpu.calib_mag();
}

void init_gps(){
    /*
    init_gps()
    This function sets up the UART Serial bus for collecting GPS data
    */
    GPS_SERIAL.begin(9600);
}
```

```

void init_humid(){
    /*
    init_humid()
    This function initializes the humidity sensor
    */
    hdc1080.begin(0x40);
}



void init_baro(){
    /*
    init_baro()
    This function sets up the barometer and loads configuration information
    */
    SPI.beginTransaction(SPISettings(BARO_SPI_CLOCK, MSBFIRST, SPI_MODE0));
    digitalWrite(BARO_SS_PIN,LOW);
    SPI.transfer(BARO_R);
    delay(5);
    digitalWrite(BARO_SS_PIN,HIGH);
    delay(5);

    uint8_t a = 0;
    uint8_t b = 0;
    for (uint8_t i = 0; i < 8; i++){

        digitalWrite(BARO_SS_PIN,LOW);
        SPI.transfer(BARO_PROM_READ | ((0b111&i)<<1));
        a = SPI.transfer(0x00);
        b = SPI.transfer(0x00);
        digitalWrite(BARO_SS_PIN,HIGH);
        baro_prom[i] = (a<<8)|b;
        delay(10);
    }
    SPI.endTransaction();
}

void init_tmp(){
    /*
    init_tmp()
    This function sets up the shutdown pin for the TMP36
    */
    pinMode(TMP_NSHDN, OUTPUT);
}

void init_sd(){
    /*
    init_sd()
    This function sets up the log file on the SD card
    */
    uint32_t buff[4] = {0x88888888, 0x88888888, 0x88888888, 0x88888888};
    sd_status = sd.begin(SD_SS_PIN, SD_SCK_MHZ(SD_SPI_CLOCK));
```

```
    log_file.open(LOGFILE_NAME, O_CREAT | O_APPEND | O_WRITE);
    log_file.write(buff, sizeof(buff));
    log_file.close();
}

void init_radio(){
    //TODO
    /*
    init_radio()
    This function initializes the radio for transceiver functionality
    */
    delay(100);
}


void init_config(){
    /*
    init_config()
    This function sets up the config file on the SD card
    */
    #ifdef CONFIG_POR
    config_file.open(CONFIG_FILE_NAME, O_READ);
    uint32_t len = config_file.available();
    config_file.seekSet(len-sizeof(config_log));
    uint8_t* buff = (uint8_t *) &config;
    for (uint32_t i= 0; i< sizeof( config_log); i++ )
    {
        if ( config_file.available())
        {
            *( buff + i ) = config_file.read();
        }
    }
    config_file.close();
    #endif
}



void init_rtc(){
    //TODO read from config file instead of static read
    /*
    init_rtc()
    This function initializes the built in real-time clock
    */
    rtc.begin();
    rtc.setTime(NOW_HOURS, NOW_MINUTES, NOW_SECONDS);
    rtc.setDate(NOW_DAY, NOW_MONTH, NOW_YEAR);

    //Sets an alarm to trigger after two hours to drive FTU control
    rtc.setAlarmTime(NOW_HOURS+2,NOW_MINUTES,NOW_SECONDS);
    rtc.enableAlarm(rtc.MATCH_HHMMSS);

    rtc.attachInterrupt(panic_ftu);
```

```
}


void init_leds(){
    /*
    init_leds()
    This function sets the pin direction for LED pins
    */
    pinMode(LED1_PIN,OUTPUT);
    pinMode(LED2_PIN,OUTPUT);
    pinMode(LED3_PIN,OUTPUT);
    pinMode(LED4_PIN,OUTPUT);

    digitalWrite(LED1_PIN, LOW);
    digitalWrite(LED2_PIN, LOW);
    digitalWrite(LED3_PIN, LOW);
    digitalWrite(LED4_PIN, LOW);
}

void init_ftu(){
    /*
    init_ftu()
    This function sets pin direction for the FTU FET
    */
    pinMode(FTU_PIN,OUTPUT);
    digitalWrite(FTU_PIN, LOW);
}

void init_heater(){
    /*
    init_heater()
    Sets pin mode for the heater FET
    */
    pinMode(HEAT_PIN,OUTPUT);
    digitalWrite(HEAT_PIN,LOW);
}

void init_watchdog(){
    /*
    init_watchdog()
    This function enables the internal watchdog timer
    A watchdog timer will reset the microcontroller if it is not cleared regularly
    This give the program a hardware mechanism of reseting in the event of a crashes
    This watchdog is set to reset the microcontroller if it is not cleared every 8 seconds
    Code is pulled from an old MASA project
    */
    GCLK->GENDIV.reg = GCLK_GENDIV_ID(2) | GCLK_GENDIV_DIV(4);

    GCLK->GENCTRL.reg = GCLK_GENCTRL_ID(2) | GCLK_GENCTRL_GENEN |
```

```
    GCLK_GENCTRL_SRC_OSCULP32K | GCLK_GENCTRL_DIVSEL;

    while (GCLK->STATUS.bit.SYNCBUSY);

    GCLK->CLKCTRL.reg = GCLK_CLKCTRL_ID_WDT | GCLK_CLKCTRL_CLKEN |
    GCLK_CLKCTRL_GEN_GCLK2;

    WDT->CTRL.reg = 0; // Disable watchdog for config
    while(WDT->STATUS.bit.SYNCBUSY);

    WDT->INTENCLR.bit.EW   = 1;     // Disable early warning interrupt
    WDT->CONFIG.bit.PER    = 0xA;   // Set period (8192ms) for chip reset
    WDT->CTRL.bit.WEN      = 0;     // Disable window mode
    while(WDT->STATUS.bit.SYNCBUSY); // Sync CTRL write

    WDT->CLEAR.reg = WDT_CLEAR_CLEAR_KEY;
    while(WDT->STATUS.bit.SYNCBUSY);

    WDT->CTRL.reg = WDT->CTRL.reg | WDT_CTRL_ENABLE;
    while(WDT->STATUS.bit.SYNCBUSY);

}

//End Program Initialization
//
-------------------------------------------------------------------------------

//
-------------------------------------------------------------------------------
//Looping Functions
void clear_watchdog(){
    /*
    clear_watchdog()
    This function clears the watchdog timer countdown
    It must be called more frequently than the watchdog timeout
    */
    WDT->CLEAR.reg = WDT_CLEAR_CLEAR_KEY;
    while(WDT->STATUS.bit.SYNCBUSY);
}

void read_mpu(){
    /*
    read_mpu()
    This function reads measurements from the MPU9250 IMU.
    The library call stores the measurements in an internal library structure
    Measurements are taken from library structure and moved into logging
    structure
    Called at the IMU sampling frequency

    Measurements are read over the SPI bus

```

```
    The measurements are converted from binary data to analog data in the library
    Analog floating point data is logged into the data logging structure
    */
    mpu.read_all();

    //ax,ay,az (g)
    data.mpu[0][mpu_offset] = mpu.accel_data[0];
    data.mpu[1][mpu_offset] = mpu.accel_data[1];
    data.mpu[2][mpu_offset] = mpu.accel_data[2];

    //gx,gy,gz (deg/s)
    data.mpu[3][mpu_offset] = mpu.gyro_data[0];
    data.mpu[4][mpu_offset] = mpu.gyro_data[1];
    data.mpu[5][mpu_offset] = mpu.gyro_data[2];

    //mx,my,mz (uT)
    data.mpu[6][mpu_offset] = mpu.mag_data[0];
    data.mpu[7][mpu_offset] = mpu.mag_data[1];
    data.mpu[8][mpu_offset] = mpu.mag_data[2];

    //time (ms)
    data.mpu[9][mpu_offset] = this_call;

    //Increments the buffer pointer, checks if all data logged
    mpu_offset++;
    if (mpu_offset==MPU_SAMPLE_RATE){
        mpu_offset--;
        ready|=0b0000000000000001;
    }
}


void read_humid(){
    /*
    read_humid()
    This function reads from the HDC1080 humidity sensor
    It requests measurements over the I2C bus
    Collected measurements are placed in the slow sample buffer
    */
    humid[slow_offset] = hdc1080.readHumidity();
}

void read_tmp(){
    /*
    read_tmp()
    This function reads from the TMP36 on board temperature sensor
    It enables the sensor, waits for it to power up, and then samples from the ADC

    Enabling and disabling the sensor is a way to limit self-heating

    Collected digital measurements are placed in the slow sample buffer
```

```
    */
    digitalWrite(TMP_NSHDN,HIGH);
    delayMicroseconds(150);
    tmp[slow_offset] = analogRead(TMP_ADC_PIN);
    digitalWrite(TMP_NSHDN,LOW);
}

void read_therm(){
    /*
    read_therm()
    This function reads from the thermistor voltage divider

    Collected digital measurements are placed in the slow sample buffer
    */
    therm[slow_offset] = analogRead(THERM_ADC_PIN);
}

void read_batt(){
    /*
    read_batt()
    This function reads from the battery voltage divider

    Collected digital measurements are placed in the slow sample buffer
    */
    batt[slow_offset] = analogRead(BATT_ADC_PIN);
}

void read_baro(){
    /*
    read_baro()
    This function reads measurements from the MS5607 barometer.
    The library stores the measurements in the slow sample buffer

    Measurements are read over the SPI bus

    The measurements are converted from binary data to analog data
    Analog floating point data is logged into the slow buffer.

    Formula for conversion is found in the datasheet
    http://www.te.com/commerce/DocumentDelivery/DDEController?Action=srchrtrv&DocNm=MS5607-02BA03&DocType=Data+Sheet&DocLang=English
    */

    //Collect the raw digital pressure value
    uint8_t a = 0;
    uint8_t b = 0;
    uint8_t c = 0;
    SPI.beginTransaction(SPISettings(BARO_SPI_CLOCK, MSBFIRST, SPI_MODE0));
    digitalWrite(BARO_SS_PIN,LOW);
    SPI.transfer(BARO_CONVERT_D1);
    digitalWrite(BARO_SS_PIN,HIGH);
    delay(10);
    digitalWrite(BARO_SS_PIN,LOW);
```

```
    SPI.transfer(0x00);
    a = SPI.transfer(0x00);
    b = SPI.transfer(0x00);
    c = SPI.transfer(0x00);
    digitalWrite(BARO_SS_PIN,HIGH);
    uint32_t read_pressure = (a<<16)|(b<<8)|c;

    //Calculate the raw digital internal temperature value
    digitalWrite(BARO_SS_PIN,LOW);
    SPI.transfer(BARO_CONVERT_D2);
    digitalWrite(BARO_SS_PIN,HIGH);
    delay(10);
    digitalWrite(BARO_SS_PIN,LOW);
    SPI.transfer(0x00);
    a = SPI.transfer(0x00);
    b = SPI.transfer(0x00);
    c = SPI.transfer(0x00);
    digitalWrite(BARO_SS_PIN,HIGH);

    uint32_t read_temp = (a<<16)|(b<<8)|c;

    //Runs the magic formula in the datasheet
    //Beware if reimplementing later, parentheses are extremely important
   here
    int32_t dT = read_temp - (baro_prom[5] * pow(2,8));

    int32_t temp = 2000 + dT * (baro_prom[6]/pow(2,23));

    float64_t temp_c = temp/100.0;

    int64_t off = baro_prom[2] * pow(2,17) + (baro_prom[4]*dT)/pow(2,6);

    int64_t sens = baro_prom[1] * pow(2,16) + (baro_prom[3] * dT)/pow(2,7);

    int64_t p = (read_pressure * (sens/pow(2,21)) - off)/pow(2,15);

    float64_t p_mbar = p/100.0;

    baro[slow_offset] = (float32_t) p_mbar;
}

void update_humid(){
    /*
    update_humid()

    This function averages the collected humidity data
    Runs a moving average filter on the humidity slow buffer
    Puts the average into the logging structure
    */
    float32_t avg = 0;

    for (uint8_t i = 0; i < SLOW_SAMPLE_RATE; i++){
        avg+=humid[i];
```

```
    }
    avg/=(float32_t)SLOW_SAMPLE_RATE;

    data.humid = avg;

}


void update_tmp(){
    /*
    update_tmp()

    This function averages the collected TMP36 temperature data
    Runs a moving average filter on the TMP36 slow buffer

    Converts the average into degrees C

    Puts the average into the logging structure
    */
    float32_t avg = 0;

    for (uint8_t i = 0; i < SLOW_SAMPLE_RATE; i++){
        avg+=tmp[i];
    }
    avg=avg/((float32_t)SLOW_SAMPLE_RATE);
    avg = (avg * VDDANA / ((float32_t)MAX_RES)) * 100.0 - 50.0;
    data.tmp = avg;
    last_temp = avg;
}


void update_therm(){
    /*
    update_therm()

    This function averages the collected thermistor temperature data
    Runs a moving average filter on the thermistor temperature slow buffer

    Converts the average into degrees C

    Puts the average into the logging structure
    */
    float32_t avg = 0;

    for (uint8_t i = 0; i < SLOW_SAMPLE_RATE; i++){
        avg+=therm[i];
    }
    avg=avg/((float32_t)SLOW_SAMPLE_RATE);
    float32_t volts = (avg * VDDANA / ((float32_t)MAX_RES));

    float32_t res = (volts * R_1)/(VDDANA-volts);

    float32_t t = BETA / log(res / R_INF) - 273.15;
```

```
    last_temp = t;

    data.therm = t;

}

void update_batt(){
    /*
    update_batt()

    This function averages the collected battery data
    Runs a moving average filter on the humidity slow buffer

    Converts the average into volts

    Puts the average into the logging structure
    */
    float32_t avg = 0;

    for (uint8_t i = 0; i < SLOW_SAMPLE_RATE; i++){
        avg+=batt[i];
    }
    avg=avg/((float32_t)SLOW_SAMPLE_RATE);
    float32_t volts = (avg * VDDANA / ((float32_t)MAX_RES));
    float32_t b = volts * (10000 + 20000)/(10000);
    data.batt = b;

}

void update_baro(){
    /*
    update_baro()

    This function averages the collected barometer pressure data
    Runs a moving average filter on the barometer slow buffer
    Puts the average into the logging structure
    */
    float32_t avg = 0;

    for (uint8_t i = 0; i < SLOW_SAMPLE_RATE; i++){
        avg+=baro[i];
    }
    avg=avg/((float)SLOW_SAMPLE_RATE);

    data.baro = avg;

}

void read_gps(){
    /*
    read_gps()

```

```
    This function attempts to poll the GPS for current data

    It reads until data is found. If a full message is found, it parses the
   NMEA
    data and puts it into an internal library structure.

    If data is valid, it is added to the datalogging structure
    */


    //uint32_t poll_start = millis();
    uint8_t disp = 0;

    while(GPS_SERIAL.available())
        disp = gps.encode(GPS_SERIAL.read());

    if (disp){
        ready|=0b0000000000000100;
        last_gps = millis();
        if (gps.satellites.isValid()){
            data.gps[0] = gps.satellites.value();
        }else data.gps[0] = 0;

        if (gps.location.isValid()){
            data.gps[1] = gps.location.lat();

            data.gps[2] = gps.location.lng();
        }else {
            data.gps[1] = 0;
            data.gps[2] = 0;
        }

        if (gps.altitude.isValid()){
            data.gps[3] = gps.altitude.feet();
        }else data.gps[3] = 0;

        if (gps.speed.isValid()){
            data.gps[4] = gps.speed.mph();
        }else data.gps[4] = 0;

        if (gps.course.isValid()){
            data.gps[5] = gps.course.deg();
        }else data.gps[5] = 0;
    }
}



void write_sd(){
    /*
    write_sd()

    This function writes the datalogging structure to the SD card
```

```
    */
    data.sof = 0xAAAAAAAA;
    data.eof = 0xCCCCCCCC;
    data.length = sizeof(data) - sizeof(data.sof) - sizeof(data.eof);
    data.time = this_call;
    data.ftu = ftu_state;
    data.heat = heat_state_1;
    uint8_t* buff = (uint8_t *) &data;

    uint32_t crc  = crc32c(0, buff+sizeof(data.sof), sizeof(data) - sizeof(
   data.eof) - sizeof(data.crc) - sizeof(data.sof));

    data.crc = crc;

    log_file.open(LOGFILE_NAME, O_APPEND | O_WRITE );
    log_file.write( buff, sizeof( data ));

    log_file.close();

    memset(buff, 0, sizeof(buff));
}

void write_config(){
    /*
    write_config()

    This function write to the SD card using the config data structure
    */
   #ifdef CONFIG_POR
    config_file.open(CONFIG_FILE_NAME, O_CREAT | O_WRITE | O_APPEND);
    config.sof = 0xFEFEFEFE;
    config.eof = 0x8A8A8A8A;
    config.length = sizeof(config);
    config.time = this_call;
    config.rtc_start_hour = 12;
    config.rtc_start_min = 0;
    config.rtc_start_sec = 0;

    config.rtc_hour = rtc.getHours();
    config.rtc_start_min = rtc.getMinutes();
    config.rtc_start_sec = rtc.getSeconds();

    config.time_to_ftu = TWO_HOURS_MS - this_call;
    config.ftu = ftu_state;
    config.heat = heat_state_1;

    uint8_t* buff = (uint8_t *) &config;
    uint32_t crc  = crc32c(0, buff+sizeof(config.sof), sizeof(config) -
   sizeof(config.eof) - sizeof(config.crc) - sizeof(config.sof));
    config.crc = crc;
    config_file.write( buff, sizeof( config ));

    config_file.close();
```

```
    memset(buff, 0, sizeof(config));
    #endif
}

//https://stackoverflow.com/questions/27939882/fast-crc-algorithm
uint32_t crc32c(uint32_t crc, const unsigned char *buf, size_t len){
    /*
    crc32c()

    This fuction calculates a CRC32 value from the given buffer

    A CRC32 value is a 32-bit checksum value that will be somewhat unique to
    a set of data
    This can be used to ensure that a data packet is valid later

    */
    int k;

    crc = ~crc;
    while (len--) {
        crc ^= *buf++;
        for (k = 0; k < 8; k++)
            crc = crc & 1 ? (crc >> 1) ^ POLY : crc >> 1;
    }
    return ~crc;
}

void update_leds(){
    digitalWrite(LED1_PIN, HIGH); //Write Power LED High

    digitalWrite(LED2_PIN,!digitalRead(LED2_PIN));

    digitalWrite(LED3_PIN, HIGH); //Write Power LED High

    digitalWrite(LED4_PIN,!digitalRead(LED4_PIN));
}

//End Looping Functions
//
    ______________________________________________________________________________


//
    ______________________________________________________________________________

//Program State Monitors
void panic_ftu(){
    /*
    panic_ftu()

    This function is the interrupt assigned to the real-time clock alarm
    After two hours have passed, it will trigger the FTU
    */
```

```
    ftu_state = 1;
}

void ftu_check(){
    /*
    ftu_check()

    This function checks the time remaining from a millis counter, and also
    updates the current state of the FTU trigger pin
    */
    if (this_call > ftu_ms_remain){
      if (ftu_start == 0){
        ftu_start = this_call;
      }
      if (this_call - ftu_start > ONE_MIN_MS){
        ftu_state = 0;
      }else{
        ftu_state = 1;
      }
    }
    digitalWrite(FTU_PIN, ftu_state);
}


void heat_check(){
    /*
    heat_check()

    This function controls the heater

    If in FEEDBACK_HEAT mode, it will try to drive the internal temperature to TEMP_SETPOINT
    If not in FEEDBACK_HEAT, it will be on for HEAT_TIME_ON ms and off for HEAT_TIME_OFF ms

    */
  static uint8_t heater_state = 0;
  if (this_call - last_heat > HEAT_PERIOD){
    heater_state = (heater_state+1)%4;
    switch (heater_state){
      case 0:
        heat_state_1 = 0;
        heat_state_2 = 0;
        break;
      case 1:
        heat_state_1 = 1;
        heat_state_2 = 0;
        break;
      case 2:
        heat_state_1 = 0;
        heat_state_2 = 0;
        break;
      case 3:
```

```
        heat_state_1 = 0;
        heat_state_2 = 1;
        break;
      default:
        /* this should be impossible */
        break;
    }
    last_heat = this_call;
  }
  digitalWrite(HEAT_PIN, heat_state_1);
  digitalWrite(FTU_PIN, heat_state_2);
}

//End Program State Monitors
//
  ------------------------------------------------------------------------------------------------
```

D.3 Header File

```
/*
flight_config.h
This file contains constants for the flight program
*/

#ifndef __CONFIG__
#define __CONFIG__

//Compilation modifiers ------------------------
//enables debug() macro
//#define SERIAL_DEBUG

//enable reset config file
#define CONFIG_POR

//toggles feedback heating or time based heating
//#define FEEDBACK_HEAT
//---------------------------------------------

//Pin Constants --------------------------------

//LEDs
//1 and 2 are on board, 3 and 4 are external
#define LED1_PIN            6
#define LED2_PIN            7
#define LED3_PIN            12
#define LED4_PIN            10

//SPI Chip select pins
#define BARO_SS_PIN         A5
#define SD_SS_PIN           5
#define MPU_SS_PIN          27
#define RADIO_SS_PIN        26
```

```

//Analog read pins
#define TMP_ADC_PIN          A1
#define THERM_ADC_PIN        A3
#define BATT_ADC_PIN         A2

//Misc pins
#define FTU_PIN              38
#define HEAT_PIN             2
#define TMP_NSHDN            9
#define RADIO_INT            A0
//-----------------------------------------------------------

//Generic RTC starting time if no config -------
#define NOW_HOURS            12
#define NOW_MINUTES          00
#define NOW_SECONDS          00
#define NOW_DAY              24
#define NOW_MONTH            02
#define NOW_YEAR             18
//-------------------------------------------------------------

//Serial Objects ---------------------------------------------
#define GPS_SERIAL           Serial
#define TRACKUINO_SERIAL     Serial1
//-------------------------------------------------------------

//Max SPI Clock Speeds ---------------------------------------
#define MPU_SPI_CLOCK        24000000
#define SD_SPI_CLOCK         24000000
#define BARO_SPI_CLOCK       20000000
#define RADIO_SPI_CLOCK      24000000
//-------------------------------------------------------------

//Sensor Sample Rates ----------------------------------------
#define MPU_SAMPLE_RATE      100   //Hz
#define MPU_SAMPLE_PERIOD    10   //ms

#define HUMID_SAMPLE_RATE    5
#define TMP_SAMPLE_RATE      5
#define THERM_SAMPLE_RATE    5
#define BARO_SAMPLE_RATE     5
#define BATT_SAMPLE_RATE     5

#define SLOW_SAMPLE_RATE     5
#define SLOW_SAMPLE_PERIOD   200 //ms
//-------------------------------------------------------------

//Keep typing convention constant
#define float32_t            float
#define float64_t            double


```

```
//Barometer Control Constants ---------------------
#define BARO_R                  0x1E

#define BARO_CONVERT_D1         0x48
#define BARO_CONVERT_D2         0x58

#define BARO_ADC_READ           0x00
#define BARO_PROM_READ          0b10100000
//-------------------------------------------------

//Logging and config constants -------------------
//Sets of flags that trigger log
#define READY_TO_LOG            0b0000000000000111
//Logging file name
#define LOGFILE_NAME            "blog0.dat"
//Config file name
#define CONFIG_FILE_NAME        "config.dat"
//-------------------------------------------------

//Analog read sensor constants -------------------
//Analog power rail voltage
#define VDDANA                  3.3F
//Twelve bit ADC resolution
#define MAX_RES                 4095
//Thermistor Constants
#define R_THERM_NOM             10000.0F
#define R_1                     100000.0F
#define BETA                    3950.0F
#define TEMP_NOM                298.15F
#define R_INF                   0.017632269789291F
//-------------------------------------------------

//CRC determination constant ---------------------

/* CRC-32C (iSCSI) polynomial in reversed bit order. */
#define POLY                    0x82f63b78

/* CRC-32 (Ethernet, ZIP, etc.) polynomial in reversed bit order. */
/* #define POLY 0xedb88320 */

//-------------------------------------------------

//FTU and Heater Constants -----------------------
#define TWO_HOURS_MS            7.2e+6 //ms
#define FIVE_MIN_MS             3e5    //ms
#define ONE_MIN_MS              60000  //ms

#define TEMP_SETPOINT           25.0F //deg C
#define HEAT_TIME_ON            30000 //ms
#define HEAT_TIME_OFF           30000 //ms
#define HEAT_PERIOD             15000 //ms
//-------------------------------------------------

```

```
//Structures ------------------------------------
//Structure to organize data to write to SD card
typedef struct data_to_log{
  uint32_t sof;
  uint32_t length;

  uint32_t time;

  //ax,ay,az(g),gx,gy,gz(deg/s),mx,my,mz(uT),time(ms)
  float32_t mpu[10][MPU_SAMPLE_RATE];
  float32_t humid;
  float32_t tmp;
  float32_t therm;
  float32_t baro;
  float32_t batt;
  //sat,lat,long,alt(feet),vel(mph),cse(deg)
  float32_t gps[6];

  uint32_t ftu;
  uint32_t heat;

  uint32_t crc;
  uint32_t eof;
};

//Structure to organize data to write to reboot log file
typedef struct config_log{
  uint32_t sof;
  uint32_t length;

  uint32_t time;


  uint32_t rtc_start_hour;
  uint32_t rtc_start_min;
  uint32_t rtc_start_sec;

  uint32_t rtc_hour;
  uint32_t rtc_min;
  uint32_t rtc_sec;

  uint32_t time_to_ftu;

  uint32_t ftu;
  uint32_t heat;

  uint32_t crc;
  uint32_t eof;
};
#endif
//------------------------------------------------
```

D.4 FTU Code

```
#include "flight_config.h"

uint32_t ftu_state = 0;
uint32_t this_call;
uint32_t last_call;
uint32_t last_ftu;
uint32_t last_config;

uint32_t ftu_start = 0;

uint32_t ftu_ms_remain = TWO_HOURS_MS; //Two Hours ms

//uint32_t ftu_ms_remain = 10000; //Two Hours ms

void init_ftu();

void setup() {
  // put your setup code here, to run once:
    init_ftu();
    last_call = millis();
    last_ftu = last_call;
}

void loop() {
  // put your main code here, to run repeatedly:
    this_call = millis(); //Gets current time
    if ((this_call - last_ftu) > 100){
      last_ftu = this_call;
      ftu_check();
    }
}


void init_ftu(){
    /*
    init_ftu()
    This function sets pin direction for the FTU FET
    */
    pinMode(FTU_PIN,OUTPUT);
    digitalWrite(FTU_PIN, LOW);
}

void ftu_check(){
    /*
    ftu_check()

    This function checks the time remaining from a millis counter, and also
    updates the current state of the FTU trigger pin
    */
    if (this_call > ftu_ms_remain){
      if (ftu_start == 0){
```

```
        ftu_start = this_call;
      }
      if (this_call - ftu_start > 2*ONE_MIN_MS){
        ftu_state = 0;
      }else{
        ftu_state = 1;
      }
    }

    digitalWrite(FTU_PIN, ftu_state);
}
```