11-2019

Design and Performance of a Communications System for a Low-Cost High Altitude Balloon Platform for Troposphere and Stratosphere Research

Noemí Miguélez Gómez

Design and performance of a communications system for a low-cost high altitude balloon platform for troposphere and stratosphere research

A Graduate Thesis
Submitted to Embry-Riddle Aeronautical University
by

Noemí Miguélez Gómez

In partial fulfillment of the requirements for the
Master of Science in
Electrical and Computer Engineering

Daytona Beach, November 2019

"You will not fail. You will just find 10,000 ways it won't work."
-Thomas Edison [edited].

EMBRY-RIDDLE
Aeronautical University

Design and performance of a communications system for a low-cost high altitude balloon platform for troposphere and stratosphere research

by

Noemi Miguelez Gomez

This thesis was prepared under the direction of the candidate's Thesis Committee Chair, Dr. Aroh Barjatya, and has been approved by the members of the thesis committee. It was submitted to the Department of Electrical, Computer, Software, and Systems Engineering in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering.

Aroh Barjatya, Ph.D.
Committee Chair

Eduardo Rojas, Ph.D.
Committee Member

Tianyu Yang, Ph.D.
Committee Member

Jianhua Liu, Ph.D.
Graduate Program Coordinator

Timothy A. Wilson, Sc.D.
Chair, Electrical, Computer, Software, and Systems Engineering

Maj Mirmirani, Ph.D.
Dean, College of Engineering

Christopher Grant, Ph.D.
Vice Provost of Academic Support

11/25/19
Date

Abstract

AFOSR Multidisciplinary University Research Initiative (MURI), "Integrated Measurement and Modeling Characterization of Stratospheric Turbulence", is a 5-year effort to resolve significant operational issues concerning hypersonic vehicle aerothermodynamics, boundary layer stability, and aero-optical propagation. Insitu turbulence measurements along with modeling will quantify spatiotemporal statistics and the dependence of stratospheric turbulence on underlying meteorology to a degree not previously possible. Data from high altitude balloons sampling up to kHz is required to characterize turbulence to the inner-scale, or smaller, over altitudes from 20 km to 35+ km.

This thesis presents the development of a standard balloon bus, based on reliable COTS components, that includes radios operating in Ham/ISM frequencies with high-gain ground station antennas to achieve high telemetry rates that potentially enable sub-cm scale sampling. It also presents the development of controlled descent systems based on reliable COTS components that allow high resolution unperturbed measurements during the descent of the balloon payloads. Both single and double balloon configurations for a controlled descent are investigated while maintaining a suitable cost for mass production of the system. We are also investigating configurations for multiple ground station to allow the use of Single Payload Multiple Ground Stations strategies to facilitate low error rate high volume data downlinking and closely-timed launches. The performance of using some retransmission techniques to download the data over altitudes from 20 to 35+km when the balloon is out of the altitude range of interest (below 20 km) is analyzed; thus, being able to reduce the percentage of packet losses even during slow descent rates, reaching long slant ranges.

This thesis is designed and implemented using Arduino IDE and MATLAB for software development and testing, circuit design with National Instrument's Multisim and Ultiboard, transceivers configuration with proprietary software, extensive components and system testing, 3D printing, temperature calibrations using a TestEquity temperature chamber, and actual high-altitude balloon launches for final performance analysis.

Acknowledgments

First of all, I would like to express my gratitude to my advisor Dr. Barjatya for being always available to answer my questions and helping me to go through this thesis. His help and guidelines while developing this project were essential to conclude this thesis, but he also believed in my abilities and gave me the opportunity to be part of this project funded by AFOSR.

Thank you to Susan Adams, for always being patience with me and for doing everything that she could to help us to obtain the hardware required for this project.

Thank you to the Office of Undergraduate Research, for the SPARK Grant that will allow me to present this work in AGU 2019.

Thank you to all my lab mates from the last two years, that helped me during this project, sharing an innumerable amount of hours in the Space and Atmospheric Instrumentation Lab (SAIL): Nick Purvis, Liam Gunter, Christopher Swinford, Peter Douglass, Julio Guardado and Kyle Hrenyo.

I would like to specially thank my family, because even though they are in a different continent, they are always close to me, encouraging me to keep moving forward. I will never be grateful enough to thank my mother for all her efforts along these years. Never. And thanks to Yoli, my sister, because without her rigor and comprehension I would not be an engineer. "Pol, keep following the steps of your mother. She always was my role model..."

And finally, because I think that she deserves all my gratitude and unconditional love, thank you Ann. I still think that you are the person who better understands what it is like to fight for something you really want to achieve. But none of this could ever be possible without you. You are my strength and part of this work is because of you too...

Glossary

A
ASeg: Air Segment
APRS: Automatic Packet Reporting System
AGU: American Geophysical Union

C
COTS: Components-Off-The-Shelf
CDU: Controlled Descent Unit

F
FAA: Federal Aviation Administration
FCC: Federal Communications Commission
FTU: Flight Termination Unit

G
GS: Ground Station
GSeg: Ground Segment
GUI: Graphical User Interface

H
HAB: High Altitude Balloon

P
PCB: Printed Circuit Board

I
IARU: International Amateur Radio Union
ISM: Industrial, Scientific and Medical

U
UHF: Ultra High Frequency
UTLS: Upper Troposphere, Lower Stratosphere

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High-altitude balloons (HABs) are manned or unmanned balloons, usually filled with helium or hydrogen, that are released into the stratosphere. They have been used for climate and meteorological research for more than 100 years, allowing near-continuous measurements from the Earth's surface into the stratosphere. HABs typically burst around 30 km and the instrument payload descends under a parachute, unless other controlled descent techniques are considered.

The most common application or balloon type are the weather balloons; however, high-altitude flight operations provide a platform for applications such as telecommunications, surveillance and intelligence, real-time monitoring for regions susceptible to natural disasters, and scientific research among others. They have even been considered for space tourism. In this section, some example of HAB systems are presented, including information about their main application, performance and specifications parameters. Some information about how those systems could not be used for the scope of this project is analysed in next sections. From this section, conclusions about why a HAB system design with different capabilities from the ones available is required for the successful of this project can be extracted.

1.1 Weather Balloon Systems

A weather or sounding balloon is a type of high-altitude balloon that carries instruments to send back information on atmospheric pressure, temperature, humidity and wind speed by means of a small, expendable measuring device called a radiosonde. These systems are basically designed to get data beginning at three meters above the Earth's surface.

Twice a day, every day of the year, these systems are released simultaneously from more than 800 locations worldwide, including 92 launched from US territories by the NOAA National Weather Service (NWS)[1]. During their 2-hour duration flights, the weather balloons are being tracked to be able to calculate wind speed and direction with high precision, among other meteorological data that is sent to the ground station. One of the radiosonde models used by NWS is the Vaisala RS92-SGP [2], which downloads the data at 2.4 kbps in the 403 MHz frequency band.

Figure 1.1: Weather balloon, top; parachute, middle, radiosonde instrument, bottom (National Weather Service).

Figure 1.1 presents an example of one of those systems launches. When the balloon bursts, the system descends only under a parachute at 40 ms^{-1} at the beginning and achieves descent rates of less than 10 ms^{-1} by the end of the flight.

1.2 Telecommunications Systems

An example of HAB systems used in telecommunications applications is the Loon project. Loon LLC is an Alphabet Inc. subsidiary working on providing Internet access to rural and remote areas. The company uses HAB systems placed in the stratosphere at an altitude of 18 to 25 km to create an aerial wireless network with up to 4G-LTE speeds [3].

The balloons are maneuvered by adjusting their altitude in the stratosphere to float to a wind layer after identifying the wind layer with the desired speed and direction using wind data from the National Oceanic and Atmospheric Administration (NOAA). The balloons also adopted figure-eight patterns instead of simple circles to stay in a specific area over longer periods of time, which indeed proved the more effective way to deliver a reliable and consistent LTE connection over time. Figure 1.2 presents one of the balloons that Loon LLC uses during their internet access campaigns.

Their communications systems have been working at unlicensed 2.4 and 5 GHz frequency bands. Google also experimented with laser communication technology to interconnect balloons at high altitude and achieved a data rate of 155 Mbps over a distance of 100 km[4].

Figure 1.2: A Loon balloon used for the internet access campaign.

1.3 Transport Systems

Due to the limitations in terms of downloaded data and validation of the results, HAB are often used just as transport platforms, so other complex systems can reach stratospheric altitudes. There are a few private companies, such as Zero2Infinity [5], using HAB systems transport platforms for "elevation services", as they called them. Their applications are divided in platforms for payload testing, satellite subsystems validation, marketing, drop tests, weather data or remote sensing. They even consider high altitude balloon platforms for "human payloads" [6].

Their stratospheric transportation service uses high altitude balloons to bring the equipment/payload to up to 22 km. Their flight cycle includes ascent rates between 4-5 ms^{-1}, up to 24h floating at a constant altitude between 18 and 22 km, and a descent using a parachute. The flight endurance depends on the total payload mass: for payloads between 2.5 and 10 kg, the maximum flight time is 10h. In those flights, the data is saved on board and the payload is usually recovered. Figure 1.3 presents an example of the balloons used for these systems.

Figure 1.3: Elevate - Zero2Infinity HAB stratospheric transportation systems.

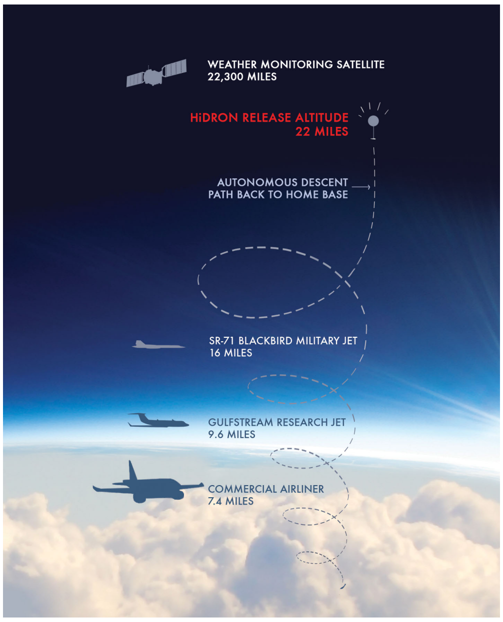


Figure 1.4: Stratodynamics Flight Height Graphic.

An example of a completely external system that takes advantage of HAB transport platforms is the HiDRON [7]. The HiDRON is an unmanned glider designed by Stratodynamics to collect high-altitude atmospheric data autonomously.

The glider is designed to be lifted by a high altitude balloon up to an altitude of 35 km, where it is released and starts descending and collecting data. Despite the harsh environments, the HiDRON is able to transmit data at 256 kbps to the ground station during a four-hour controlled descent up to a range of 100 km to a data relay network. This system requires a flight path pre-programmed to work as expected. This subsystem trajectory can be seen in Figure 1.4.

1.4 Academic Research Systems

The low cost of the equipment for high-altitude balloon launches, makes them a hands on project; where several organizations even assist and commercialize the development of their payloads. One such example is High Altitude Science [8] that provides HAB kits and instruments at a relatively affordable cost, from launch setup materials to communications systems. Even if there is no science instrument on board a HAB, a communication link is required to at least be able to track it. Under certain regulations, their payloads can use ISM and amateur radio frequencies for the data transmission, assisting the flight path tracking, and the data downloading from the on-board sensors. The data rate required from those sensors depends on the balloon application and desired measurements resolution.

There are global education programs and companies that provide students an opportunity to design and compete to launch experiments into space using high-altitude balloons; they can engage in activities to design and develop the on-board experiments and they expand the usage of these profitable systems.

Figure 1.5: Idoodlelearning - Cubes in Space Program, 2016.

Idoodlelearning inc. [9] is a global education company that provide free high-altitude balloon and rocket launches to students participating in their program 'Cubes in Space' with the collaboration of NASA. The students have to design an experiment that fits into a 4 cm cube that has to be launched into space (or near space environment) and perform different analysis, e.g. materials, sensors accuracy, battery cells experiments. Figure 1.5 presents the deployment of this system for the program of 2016.

NASA has a collaborative High Altitude Student Platform (HASP)[10] that uses HAB systems to provide students with flight/launch opportunities for their research payloads. The HASP flight program is supported by the NASA Balloon Program Office and the Louisiana Space Consortium. Currently, HASP flies once a year in September from the Columbia Scientific Balloon Facility (CSBF) base in Fort Sumner, New Mexico.

HASP carries all the payloads to altitudes of 36 km at an ascent rate of 5 ms^{-1}, for durations of up to 20 hours. After that, the platform descends at rates higher than 15 ms^{-1}. Figure 1.6 presents an example of one of the HASPs.

Figure 1.6: NASA High-Altitude Student Platform launch.

1.5 AFOSR - MURI Project

The design of hypersonic vehicles needs to account for the effects of ambient atmospheric turbulence and particles in the middle stratosphere. The lack of statistically significant turbulence measurements at those altitudes makes it hard to design the aerodynamics of aircraft that can consistently fly at hypersonic speeds (above Mach 5 or 3,800 mph) for a long time. Furthermore, availability of such data will enable constraining and parameterizing of detailed modelling.

The AFOSR funded Multidisciplinary University Research Initiative (MURI) "Integrated Measurement and Modeling Characterization of Stratospheric Turbulence" [11] is a 5-year project consisting of a consortium of three universities - University of Colorado Boulder, Embry-Riddle Daytona Beach, and University of Minnesota- working on HAB platforms for common goals. The HAB platforms will be used for hypersonic boundary layer modeling, aero-optical propagation assessments, and linkages from meteorology to stratospheric turbulence statistics, yielding the following expected outcomes addressing US Air Force capabilities:

- Quantify the roles of atmospheric turbulence and particle concentrations on laminar-turbulent transition for hypersonic flight conditions.
- Rigorously connect the atmospheric turbulence state to the disturbance forcing amplitude of relevant boundary layer instability mechanisms.
- Understand how atmospheric particles interact with a hypersonic flow field and promote instability growth and transition to turbulence.
- Quantify the impacts of stratospheric turbulence spatio-temporal statistics and larger-scale coherent refractive index fluctuations on long-distance aero-optical propagation.
- Provide a "strawman" stratospheric turbulence forecasting scheme accounting for variable environments and energy inputs from meteorology at lower altitudes.

To cover the previous capabilities, the following research points shall be addressed:

- Spatio-temporal statistics of small-scale turbulence measurements in the middle and upper stratosphere, and to what extent are they dictated by larger-scale motions, such as primarily gravity waves that arise from meteorological sources at lower altitudes.
- Distributions of particles in the stratosphere, and their dependence on underlying meteorology.
- Relative roles of particles and pre-existing atmospheric turbulence for the laminar-turbulent transition at hypersonic speeds in the middle and upper stratosphere.
- Effects of particles, temperature sheets, and small-scale turbulence in the middle and upper stratosphere on long-range optical propagation and how can these effects be accurately represented in computational simulations.

1.6 Thesis Outline

The MURI HAB system design, implementation, and testing constitutes the scope of this thesis, from payload subsystem components to controlled descent units for single and double balloon configurations. The development work is enumerated in several chapters and appendices, showing the progress made in the different stages of the design and the different approaches analysed:

- In the next Chapter 2, the state of the art of HAB regulations and policies, controlled ascent and descent systems, payload tracking and data downloading techniques is presented.
- Chapter 3 presents the hardware and software design of both ground station and payload systems. From early design stages with first considered transceiver and on-board microcontroller models to double and single balloon controlled descent unit designs. It includes the PCB design for the final stages, when needed, and a summary of main changes and conclusions considered when updating the design.
- Then, Chapter 4 presents the main results obtained from the final designs. The results will demonstrate that the project requirements are met and will present the system behaviour in real scenarios.
- Chapter 5 details the final design costs and the available facilities that were used for the development of this thesis.
- The main conclusions of the thesis efforts are discussed in Chapter 6.
- The future approaches to improve the final designs and the integration of the ERAU part of the AFOSR-MURI project are presented in Chapter 7.
- Finally, a series of appendices incorporate information about modules configuration, ground station setup, sensors calibration, PCB designs, and software used for both ground and air segments.

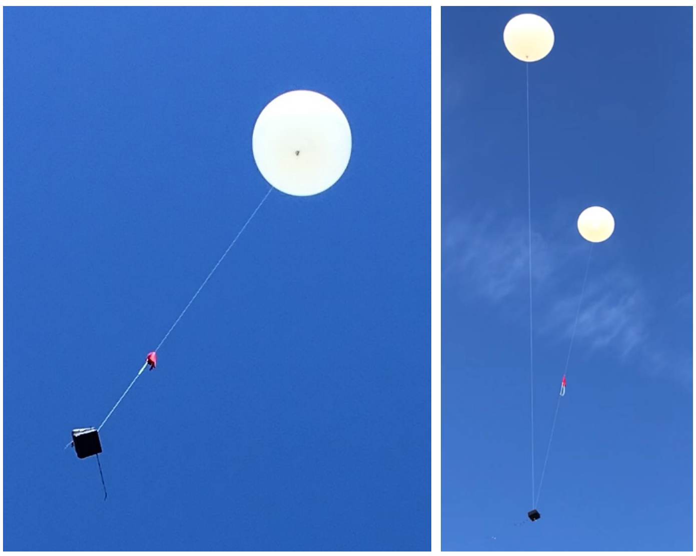


Figure 1.7: ERAU HAB Systems - Single and Double Balloon Configurations.

Chapter 2

State of the art

The state of the art of this project is a brief introduction of high-altitude balloon system performances and applications from both the ground and the air segments. It covers regulations considered when developing these HAB systems, approaches used for single and double balloon configuration launches to achieve a controlled ascent and descent, performance parameters of interest -achieved altitude, resolution of the measurements-, and the payload tracking techniques.

2.1 HAB Regulations and Policies

HAB launches are subjected to governing laws and regulations of the country to ensure the safety of pilots and the communications regulations.

The following FAA and FCC laws and regulations shall be considered and always checked for possible updates. The following list presents a summary of the most important ones to apply to the HAB design and launches:

- **Federal Aviation Administration (FAA) - Part 101 [12]**:
  - No person may operate an unmanned free balloon at any altitude where there are clouds or obscuring phenomena of more than five-tenths coverage.
  - No person may operate an unmanned free balloon at any altitude below 60,000 feet (18 km) standard pressure altitude where the horizontal visibility is less than five miles.
  - No person may operate between sunrise and sunset an unmanned free balloon with a suspension device more than 50 feet (15 m) along, without this device being visible for at least one mile.
  - The balloon shall be equipped with at least two payload cut-down systems or devices that operate independently of each other.
  - The balloon envelope shall be equipped with a radar reflective device(s) or material that will present an echo to surface radar operating in the 200 MHz to 2700 MHz frequency range.
  - Any individual payload must weight less than 6 pounds (2.7 kg).

- Total payload of two or more packages carried by one balloon must be less than 12 pounds (5.4 kg) total.
- The balloon cannot use a rope or other device for suspension of the payload that requires an impact force of more than 50 pounds (22.7 kg) to separate the suspended payload from the balloon.
- No person operating any balloon may allow an object to be dropped therefrom, if such action creates a hazard to other persons or their property.
- The local FAA ATC must be notified of the estimated date and time of launching, amended as necessary to remain within plus or minus 30 minutes, as well as the launching site and forecast trajectory.
- Each person operating an unmanned free balloon shall forward any balloon position reports requested by ATC.
- One hour before the descent, the person operating the balloon shall forward to the nearest FAA ATC facility the altitude and forecast trajectory.
- If a balloon position report is not recorded for any two-hour period of flight, the person operating the balloon shall immediately notify the nearest FAA ATC facility, providing the last recorded position and any revicion of the forecast trajectory.

• **Federal Communications Commission (FCC) - 22.925 [13]**:

- Cellular telephones installed in or carried aboard must not be operated while are airborne. The violation of this rule could result in suspension of service and/or a fine.

2.2 Controlled Ascent and Descents

High-altitude balloon experiments are a key point for vertical profile measurements in the upper troposphere and lower stratosphere (UTLS). Traditional meteorological methods employed by national weather services start with ascent at approximately 5 ms^{-1} up to the altitude of balloon burst, when it starts descending at high speed (40-60 ms^{-1}) for about 20 km, until the parachute reduces the descent rate to less than 40 ms^{-1}. Considering that the parachute works as expected, the payload impacts the surface at up to 15 ms^{-1} [14]. It has been demonstrated that ascending weather balloons can perturb the UTLS measurements; and at the aforementioned descent rates, the vertical resolution and accuracy of the measurements are critically reduced. Consequently, the use of controlled descent techniques has been investigated in this thesis.

There are different designs to control the start of the descent, commonly known as Flight Termination Units (FTU) or Controlled Descent Units (CDU). Custom packages located above the parachute that separates/cuts the payloads from the balloon before its burst altitude are considered FTU. In this case, the payload descents with a single parachute at the aforementioned high speeds.

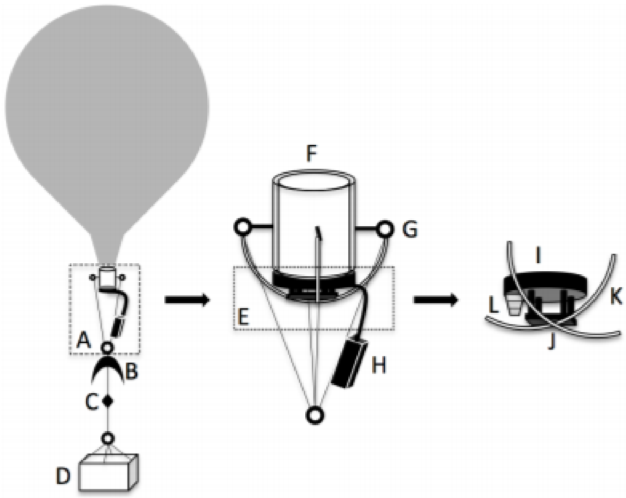


Figure 2.1: Single balloon method of controlled descent the balloon flight consisting of (A) the automatic balloon valve and pressure sensor assemblies (B) a parachute (C) a 52 m string unwinder and (D) the instrument payload. The valve and pressure sensor assemblies include (E) a valve cap assembly (F) a PVC pipe segment (G) four screw-in eyelets and (H) a pressure sensor, logic board and batteries. The pipe cap assembly includes (I) a pipe cap (J) a hot wire string cutter (K) two cap anchoring strings and (L) a helium fill port.

For a slow descent, CDU designs are considered for double/single balloon configurations. In those designs, at least one balloon will descend with the payload, enabling descent rates of 2-4 ms^{-1} to obtain high-resolution measurements during that part of the flight.

A. Kräuchi et al.[14] presented two different approaches, used by NOAA for the past decade, for achieving controlled slow descent: single-balloon scheme with a vent mechanism for the lift gas and double-balloon scheme wherein one balloon is released and descent occurs under one balloon.

For the single balloon mechanism, a valve system attached to the neck of the balloon is activated at a desired pressure. The valve system consists of a PVC pipe, a pipe cap, two anchoring strings and a hot nichrome wire. The strings will retain the pipe cap until a certain pressure is reached and the nichrome wire will burn them. Once the cap falls away, the helium flows out of the balloon through the pipe. The balloon keeps ascending until it reaches a neutral buoyancy and then begins the descend as more helium is released. An sketch of this design can be seen in Figure 2.1.

Payloads up to 5 kg were able to successfully flown with this CDU, achieving descent rates of approximately 5.4±0.4 ms^{-1} at 30-25 km to 3.1±0.3 ms^{-1} below 14 km. The difference in those rates is based on the air pressure at the valve opening and the temperature of the internal gas at different altitude ranges. Between 2008 and 2016, NOAA launched 250 balloons with this CDU design, achieving successful controlled descent in 75% of them, reaching a maximum altitude of 30 km.

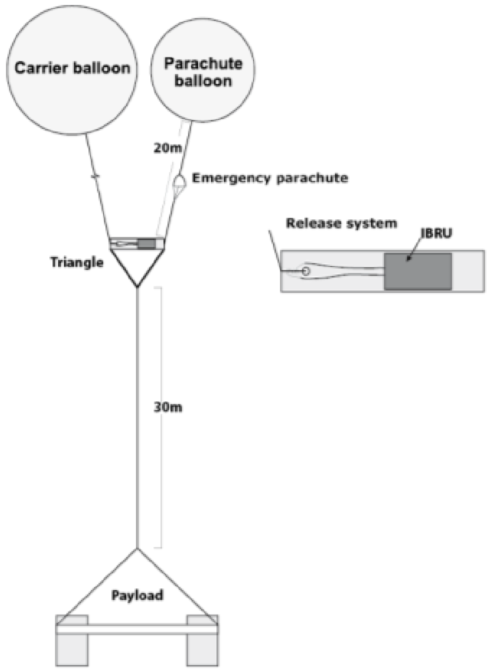


Figure 2.2: Double balloon method of controlled descent with carrier and parachute balloon connected to the payload via the triangle that includes an Intelligent Balloon Release Unit (IBRU) release mechanism.

The double balloon configuration technique presented in A. Krauchi et al [14] can be seen in Figure 2.2. As it can be seen, this technique uses a carrier balloon to lift the payload and a second balloon that acts like a parachute to allow a slow descent. The payload is connected to a triangular frame, where each balloon is connected to one vertex. The frame contains another hot wire mechanism to cut the string of the carrier balloon at a certain altitude, periodically measured by a GPS receiver. In this case, the carrier balloon is inflated with enough gas to lift the payload at 5 ms^{-1}, while the other balloon is only inflated with enough gas to maintain a 5 ms^{-1} descent rate once the other balloon is released.

The double balloon mechanism presents reduced pendulum motions when compared with the single balloon mechanism, which is important for the quality of the measurements. This mechanism also improves the stability of the descent rates.

In Vignelles et al.[15], the data of 95 launches over 3 years achieving a mean altitude of 30.5±4.2 km is presented. The main goal of those launches was to measure the spatial and temporal variability of aerosols in the troposphere and stratosphere. The minimum altitude achieved was 14.4 km and the maximum was 36 km, with only two balloons crossing 35 km. During these launches, only the ascent part of the flight was considered, since a CDU was not included in the system, and the payload was descending under a parachute. The data of 18% of the launches was declared invalid, due to perturbations in the measurements. The source of some of those perturbations are caused by the balloon system crossing the area of measurements before the specific sensors.

2.3 Payload Tracking Systems

It is important to be able to track a balloon trajectory due to regulations, but there are other important reasons to do that:

- A balloon tracking system allows to communicate with the payload and receive telemetry back or send commands to it even at high slant ranges from the ground station.
- An accurate balloon tracking system provides a possibility to recover the payload when it lands, with low uncertainty of its final location.

The different available techniques to track the payload are based on GNSS/GPS technology to transmit the position of that payload. The main difference between those techniques is how to get the information to the ground station to be able to track the system: the coordinates can be sent using an on-board transceiver that transmits the payload position to a satellite network, amateur Automatic Packet Reporting System (APRS) stations, cellphone towers or custom ground stations working at the frequency band of the transmitter.

Considering that FCC regulations don't allow the use of cell phones during the flight, only satellite and amateur tracking techniques are going to be analysed in this section:

- **Satellite Balloon Tracking.**
  Satellite trackers are designed to rely on a network of satellite in orbit to receive their position signal. Once the correct coordinates are obtained, the tracker beams the packets to a communication satellite to relay the position to various ground stations using Internet connections. However, there are a few things to keep in mind when using a satellite tracker:
  - The antenna of the payload shall be always pointed at sky. If not, the satellite in orbit will possibly not receive the position signal. Many payloads have been lost for this reason.
  - Satellite trackers require a subscription fee.
  - The position is only updated once every 5 or 10 minutes, so the accuracy of the measurements based on position is low, because only flight path predictors cannot provide the required level of accuracy.
  - Satellite trackers do not use specialized GPS receivers and there are typically stop updating position above 18 km. Once the balloon starts descending, below 18 km, the tracking is resumed.
- **Amateur Balloon Tracking.**
  A portion of the ISM spectrum is reserved for amateurs and can be used to send your balloon position to your ground station. In this case, there are different options too:
  - **1.- Automatic Packet Reporting System (APRS).**
    Thousands of stations are listening your balloon transmissions, performed by modules similar to the one presented in Figure 2.3 from Stratotrack.

Figure 2.3: StratoTrack APRS Transmitter [16].

Once they hear your packet, they automatically push it to the internet to display on a map. The system can rely on data backup and there is no need to download the data during the flight if the payload is recovered. These are the main things to consider about these systems:

- ∗ To legally use an APRS tracker, the FCC does require that you have an amateur radio license.
- ∗ Most APRS trackers are designed for tracking vehicles; therefore, their GPS receivers have the same issue of not working above certain a altitude (18 km in this case) as satellite trackers do.
- ∗ The cost of a APRS tracker can vary from $200 to $600.
- ∗ If the payload lands in a rural area, far from an amateur radio station that can receive the tracker's signal, the payload coordinates are never received. That is why these systems are usually used as supplements to satellite trackers.

– **2.- ISM - Communications System.**
A completely dedicated and independent from other sources communications system is used where the balloon sends its GPS coordinates and the telemetry of interest to a ground station that is tracking only the signals from that particular balloon during its flight, and saving all the data of interest. This is the approach presented in this thesis since it is completely modular and customizable; therefore, it can be adapted to possible project changes. Moreover, it is not dependable of the availability of external signals or monitoring systems.

2.4 Data Downloading

HAB platforms are usually used as on-board data loggers, due to the fact of not having a dedicated ground station to download the data to, and their maximum slant ranges capabilities.

During the 95 launches presented in D. Vignelles [15], in order to avoid measurement disturbances, the data recorded on-board was only transmitted 0.35/1 seconds. During that transmission time, the data was not saved, resulting in only 0.65/1 seconds of measurements. For an average ascent rate of 5 ms^{-1}, 1.7 m every 5 m was not recorded. For the purpose of this project, high spatial and temporal resolutions are required and, therefore, this approach should be improved if it needs to be used for the MURI HAB launches. The total time that the communications of the payload are stopped can be reduced using data rates as high as possible.

In A. Shagger and N. Amilia [17], a communications subsystem independent of the GPS tracking system (APRS) was designed by the University Saints Malaysia with a maximum range of 50 km, but most of the data was stored on-board to decrease the data transfer to the ground. Considering that the payload is not always recovered, this approach could easily translate in low measurements resolution or even invalidation of the launch data.

Another example of HAB communications systems is the one designed to test a CubeSat payload in terms of functionality in H. Kimm et al. [18]. This system only transmitted data from a movement sensor at 1 Hz, with a system based on APRS. The communications link was maintained for the whole launch duration and the payload was recovered, but the resolution of the measurements was low. Moreover, the on-board subsystems were COTS CubeSat components, which make the balloon system very expensive to mass-produce for this project analysis purposes.

SparkFun provides several components to be used for HAB platforms, including examples of complete HAB systems and flight analysis [19]. In one of them, a 1W transmitter was included in the payload to download scientific data to the ground working at the 900MHz ISM band. The system reached 15 miles (24 km) of slant range before losing the transmission link, due to the type of antennas and the tracking system used. Since slow ascent and descent rates can be translated in HAB systems flying far away from the ground station, a slant range of only 15 miles is not enough to be compliant with the communications link requirements for this project.

Future HAB communications systems are moving towards heavy systems of up to 1 ton to be able to work as satellite or WiFi signals relays for fixed or mobile services in stratospheric altitudes [20]. Those systems will be able to provide high-data rates, but at a high cost and difficulty, which it is out of the scope of this project, since their mass-production is not affordable. Figure 2.4 presents an example of those platforms.

In summary, high-data rates HAB communications systems have not been exploited since they can be used as data loggers and they were not economically affordable. Even with high data rates downlinks, the maximum slant range achieved did not allow slow ascent and descent rates. A new communications systems needs to be designed for this project, since payload recovering will not always be possible and downlink rates of at least 80 kbps shall be considered for high resolution measurements.

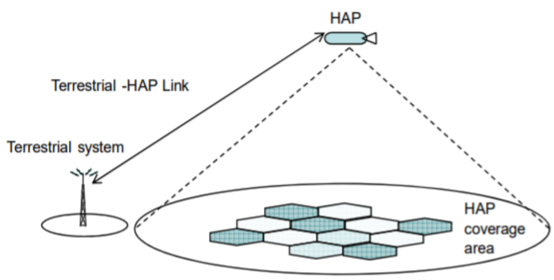


Figure 2.4: High-Altitude Balloon Platform - Terrestrial System [20].

Chapter 3

Design and Implementation

Considering the information presented in the previous chapters, chapter 3 presents the design considerations and implementation of both the ground and the air segments of the project. First, a summary of the project requirements and objectives is presented, followed by the design constraints and considerations. Then, the ground station design is explained, including the Graphical User Interface (GUI) used to control and monitor the system. Finally, a detailed description of the main stages of the payload and controlled descent unit designs is presented.

3.1 Project Requirements and Objectives

The MURI High-altitude balloons will carry high data rate sampling instruments on-board to allow sub-cm scale measurements. During their flights, real-time data is transmitted to a ground station that is tracking the payload as well as storing the received data for future analysis. The data transmission is required as retrieving of balloons launched from certain locations is impossible; for example in Florida where most of them end up in the ocean or in alligator swamps. Furthermore, the sub-cm scale spatial sampling required by the instruments necessitates high data rate communications over long range with a communications link with as low percentage of losses or data errors as possible.

Taking into account that some of the sensors on-board will probably only record valid data during the descent, a controlled descent mechanism must be considered. Moreover, it makes possible to use the data at the altitude range of interest twice, for the sensors that can obtain valid data during the ascent too.

In view of all previous research and the objectives of the project, the MURI project in ERAU has set the following requirements for the balloon bus:

- Achieve capability for consistent high altitude (+30 km) launches.
- Achieve undisturbed environment for turbulence measurements, i.e. slow descent.
- Achieve cheap high-data rate telemetry for centimeter scale turbulence measurements (~100 kbps).

- Ability to 'mass produce' balloon payloads with optimum trade-off between low cost and capability to allow more launches for the same cost.
- System design for simultaneous multi-point balloon launches and measurements, or multiple follow-on launches for temporal measurements.

3.2 Design Constraints and Considerations

In this section, the main design constraints and considerations are discussed. First, a summary of the size, weight, power and cost requirements and considerations is presented. Then, a preliminary link budget is discussed considering expected maximum working slant range for the communications link.

3.2.1 Size, Weight, Power and Cost (SWaP-C)

The SWaP-C considerations for this project were basically based in FAA/FCC regulations and the requirements of the project. As it will be seen, they do not present exact numbers, but an approximation of which limits or goals we should or should not achieve/reach.

- **Size and Weight**
  Considering the FAA regulations, the maximum weight for any individual payload is 6 pounds (2.7 kg), and the total weight that a balloon can carry is 12 pounds (5.4 kg). However, considering that cost is important for mass-production purposes, the payload shall be as light as possible to be able to reduce the cost in the type of balloon used for the launches and the amount of helium to lift the payload at the desired ascent rate. While that could be also translated into a specific size required to cover all the hardware, the use of light styrofoam boxes eliminates size restrictions as long as the payload is compliant with the other constraints and regulations.

- **Power**
  In terms of power, it had to be considered that the power system shall be designed to be able to power the whole payload subsystem for at least a 5 hour launch. This value accounts for slow ascent and descent rates and an average altitude of 33 km.

- **Cost**
  Taking into account that one of the project requirements is to mass-produce the payloads to be able to launch several of them to take turbulence and other measurements, cost is an important specification to consider when designing the whole system. HAB academic launches costs typically are between $1,000 - $1,500 per launch, depending on the main on-board experiment. ERAU is considering a price ceiling of $1250 per launch to make some of the design decisions that will be seen in the next sections.

3.2.2 Communications Link

A preliminary link analysis with worst case scenarios assumptions was used to determine the possible transceivers to be considered for the communications system design. The minimum required specifications to achieve long ranges with low percentage of data losses were specified when analysing this link budget. The results of this analysis were taken into account during the design process and the selection of some of the parts and components.

First of all, the frequency allocation was considered, based on the available transceivers and the cost and performance of each one of them. In order to choose the proper transceiver, a table of available transceivers and their characteristics was linked to the link budget sheet used for the calculations. Based on that analysis, the 900-928 MHz frequency band was selected due to the following advantages and specifications:

- 900-928 MHz frequency band is one of the Industrial, Scientific and Medical (ISM) radio bands and no license is required to operate it.
- 900-928 MHz frequency band is part of region 2, which includes the Americas, and the regulations applied are suitable for this project, such as maximum Effective Isotropic Radiated Power (EIRP) allowed of 4 W (i.e. power output of 1 W and up to 6 dBi of antenna gain).
- The number of available transceiver modules suitable for our design requirements in the 900-928 MHz band is higher than in other ISM bands -433MHz, 2.4GHz- and the specifications are better for this project: maximum transmitted power and configurable data rate, and cost.
- SAIL, one of the facilities used for this project, already owned a 900MHz-17dBi Yagi antenna that was available to be used in the project.

Considering the frequency selected, the transceivers list was reduced and the best ones were selected to develop the payload design presented in next sections. For the link budget analysis, the free space path losses, the atmospheric attenuation, the receiver temperature and the antenna efficiencies were taken into account to estimate the link margin for a FSK modulation, adding approximations of expected losses from hardware, atmosphere or environment interferences.

Table 3.1 present the main parameters considered when computing the link margin of the communications link. There is not a specific valid link margin value, but recommendations from IARU/AMSAT and local radio amateurs suggests that the link margin should be approximately 8-10 dB on top of the SNR value in order to be certain that the communication link will work. The SNR margin depends on the bit error rate considered, taking into account the receiver sensitivity, which varies depending on the data rate used. In this case, the maximum configurable data rate (250 kbps) is considered as the worst case scenario, even though during the final system integration this parameter could change. With those considerations, the margin is approximately 8 dB for a maximum considered slant range of 140 km.

Table 3.1: Link Budget

Parameter	Symbol	Units	Value
Center Frequency	f	[MHz]	915
Transmitter Power	P_{tx}	[dBW]	0
Transmitter Antenna Gain	G_r	[dB]	5
Antenna/Transmitter Loss	L_t	[dB]	-1.33
Equivalent Isotropic Radiated Power	EIRP	[dBW]	3.67
Propagation Path Lenght [Max.]	S	[km]	140
Free Space Loss	FSPL	[dB]	-134.60
Atmospheric Loss	L_a	[dB]	-1
Polarization Loss	L_p	[dB]	-3
Receiver Antenna Gain	G_r	[dB]	17
Receiver Loss	L_t	[dB]	-1.5
Antenna Misalignment Losses	-	[dB]	-1.78
System Noise Temperature	T_{sys}	[K]	1000
Power Flux Density	-	[dB(W/m^2)]	-110.25
Data Rate	R	[bps]	250000
Eb/No	Eb/No	[dB]	21.27
Required Eb/No [BER 1e-5]	Eb/No$_{req}$	[Eb/No]	13.3
Margin	-	[dB]	7.97

The following link equation was considered to compute the link margin:

$$\frac{E_{\rm b}}{N_{\rm o}} = \frac{EIRP\ FSPL\ L_{\rm p}\ L_{\rm t}\ L_{\rm a}\ G_{\rm r}}{kT_{\rm sys}R}, \tag{3.1}$$

where k is the Boltzmann constant.

It is important to consider that when doing this link budget, some parameters are approximated, since one cannot exactly predict the environment interferences at the working frequency band.

3.2.3 Feasibility of Existing HAB Systems

Considering the project requirements and the design constraints and considerations from the previous sections, this section presents a feasibility analysis of the HAB systems presented in Chapter 1.

Weather balloons are a good example of multi-point measurements systems, being tracked and downloading data in real time, but they are not taking into account the down-leg of the flight. The amount of data that they need to download does not require high data rates to be able to ensure high resolution measurements, and they do not get data during the descent part of their flights. However, the capability of mass-producing them to be able to launch two of those systems per day makes them a good system design example for this project.

Even considering that one of the project requirements is to obtain a high data rate communications link, the cost of the Loon LLC system and the working altitude range exclude them from being considered in this project payload design.

In terms of transport systems, Zero2Infinity's system maximum working altitude is 22 km, which makes this system not compliant with our requirements. While HiDRON is compliant with the altitude requirements, it requires a flight path pre-programmed and even though it would be a good option for payload recovery and high data rate downlink, it is still a premature idea that will increase the cost and development time of this project. The impact in terms of cost to develop platforms like those is out of the requirements and capabilities of this project.

Academic research systems, such as HASP and Idoodlelearning, present an affordable low cost for amateur groups and students. However, this project will require multi-point measurements that cannot be ensured with this type of projects, where the experiments are just exposed at a certain altitude for a certain amount of time and they cannot be launched from anywhere. The high descent rates make the systems not suitable for undisturbed measurements while descending.

Table 3.2 presents a summary of the feasibility of the previously presented systems, considering the requirements for this project. It can be concluded that a completely new payload compliant with all the requirements needs to be designed.

Table 3.2: Existing HAB Systems - AFOSR MURI Project Feasibility

Characteristic	**Radiosonde**	**ZeroToInfinity /HiDRON**	**HASP /Idoodle**	**Loon**
Cost	X	-/-	X/X	-
Altitude Range	X	-/X	X/X	-
Data Rate	-	X/X	X/X	X
Launch Locations	X	-/-	-/-	-
Descent Rate	-	-/X	-/-	-

3.3 Ground Station

To track the payload and retrieve as much data as possible, a ground station that combines high-gain antennas, a calibrated and configurable rotor controller and an easy-to-deploy modular design is considered. This section presents this ground station design, based on the Yaesu G-5500 rotor system [21].

3.3.1 System Overview

The Yaesu G-5500 is a rotor system that has both azimuth and elevation (Az/El) controls. The azimuth of the rotor has a turning range of 0°-450°. The elevation of the rotor has a rotation range of 0°-180°. This rotor system is used by many universities and amateur radio operators to point antennas for different uses, from HAB to satellite projects. Yaesu offers a computer interface for their rotor, however it requires RS-232 connection, and the adapter to be able to use it, manufactured by the same company, is more expensive than the rotor itself - approximately $850-. A USB computer interface that has increased functionality was built, considering a maximum cost of $200. This computer interface was designed in this project scope, consisting on a microcontroller board based on the ATmega2560 -Elegoo Mega-, as the main rotor box controller. The microcontroller is in charge of getting the actual antenna pointing and being able to change it by considering actual and desired Az/El parameters. A printed circuit board (PCB) shield was produced to do the signal conditioning required to communicate with the rotor controller box. More information is presented in the next sections and in Appendix A.

One of the distinctive points of this ground station is that it is completely modular. A modular design enables the possibility to transport the ground station and to easily do launches in the field, being able to have a functional ground station in approximately 45 minutes. If a permanent ground station is not a feasible option for a specific team due to space availability or permission, a modular ground station is the best option to be considered.

The ERAU ground station consists of 5 modules: antenna module, rotor module, mast, tripod, and base plates.

- **Antenna**: it should be a high-gain antenna to enable long range communications, as well as directive to avoid as much environment interferences as possible. It also should present enough H-V beam-width to be able to afford pointing errors without substantial signal power losses.

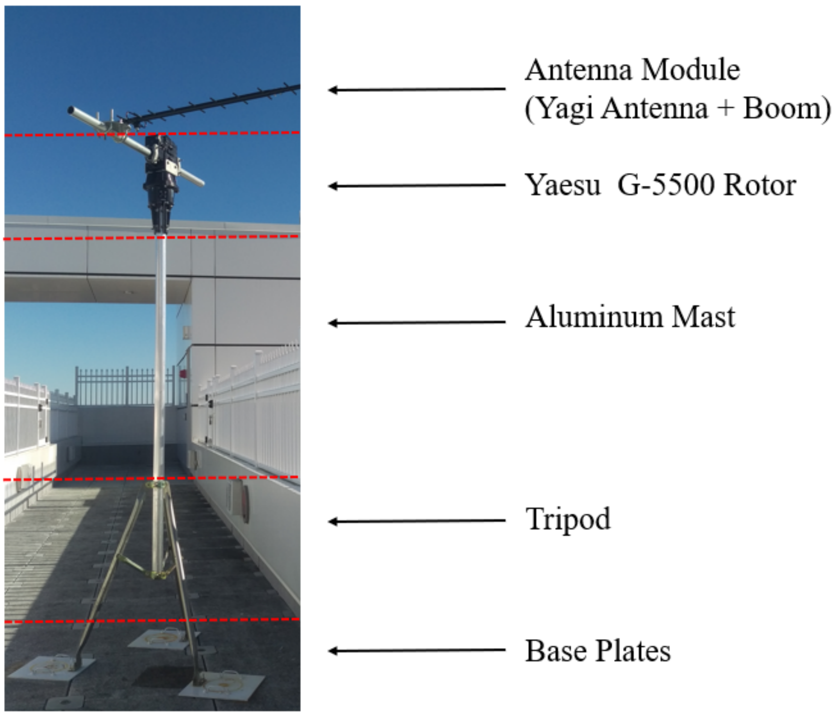


Figure 3.1: ERAU Ground Station Modules

- **Rotor**: the rotor module includes the Yaesu G5500 rotor and controller box, as well as the designed shield to control the rotor controller box automatically. As aforementioned, the algorithm and PCB design to be able to analyze the actual position of the rotor and move it properly to point towards the payload was developed in this thesis scope.
- **Mast**: it shall provide enough altitude to the rotor to be able to avoid interferences due to multipath with the ground and the building structures, and enough line of sight with the HAB payload at the beginning of the launch.
- **Tripod**: the whole rotor and mast structure must be as secured as possible to the ground to avoid north misalignments and pointing offsets during the flight. A 3-legged tripod attached to heavy base plates is used for that purpose.
- **Base Plates**: the whole rotor, mast and tripod structure shall be stabilize in the ground using base plates and, possibly, adding some weights on top of them.

Figure 3.1 presents a mobile ERAU ground station setup, with the different modules differentiated. More information about the ground station modules, their production and configuration, as well as the overall ground station setup can be seen in Appendix A.

3.3.2 Rotor Box Controller

The main part of the ground station design is the pointing control system. This section presents the design of the automatic rotor box controller. Considering that the rest of the modules are hardware parts commercially available or produced in ERAU, only the pointing control system design and implementation is presented in this section, while all the other modules information can be found on Appendices A and C.

The rotor box controller is based on the Yaesu GS-232 interface and is divided in two parts: the microcontroller and the PCB design.

3.3.2.1 PCB Design

The PCB design is based on the actual design of the rotor box controller provided by the company itself. It includes 4 NPN transistors to isolate the G-5500 from the microcontroller control signals used for both azimuth and elevation directions, an operational amplifier to improve low-voltage characteristics when working with the low voltage readings coming from the low Az/El ranges, and a set of 10KΩ and 1pF resistors and capacitors for signal conditioning purposes. This design includes a 5 pin molex connector where a GNSS sensor can be connected in case real-time ground station position tracking is required (i.e. the ground station position is continuously changing).

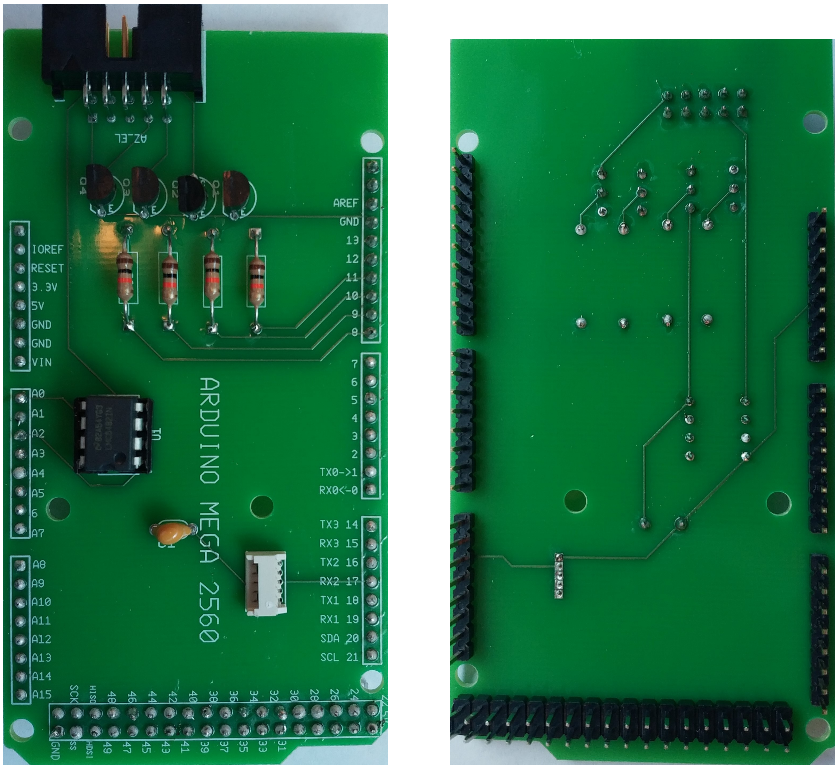


Figure 3.2: Rotor Box Controller - PCB Shield.

As it can be seen in the Figure 3.2, to connect the shield to the rotor controller, a 10-pin female connector is included in the PCB. A cable with the 10-pin male connector to the PCB in one side and a 8-pin male connector matching the rotor box connection is required for the external control connection with the rotor controller box. Figure 3.3 and Table 3.3 present the rotor box controller connections.

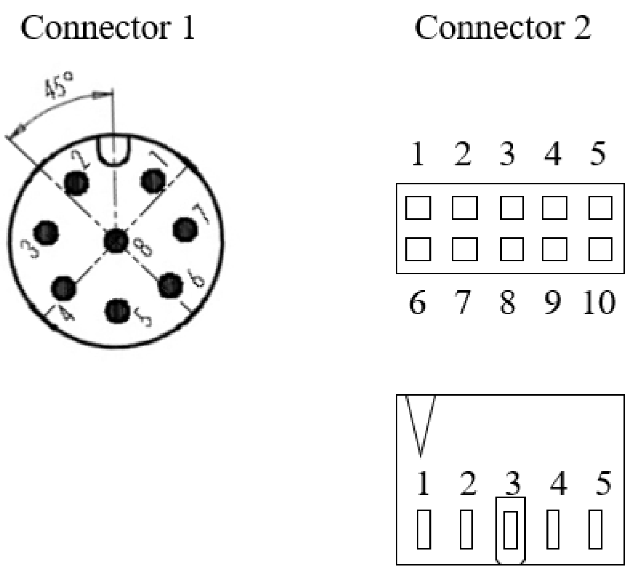


Figure 3.3: Rotor Box Controller - Connections between Shield and Rotor Box

Table 3.3: Controller-Rotor Box Connections

Conn. 1 Pin#	Conn. 2 Pin#	Name/Description
1	-	-
2	2	El Analog Reading
3	7	Az Analog Reading
4	4	Az-LEFT
5	5	Az-RIGHT
6	10	El-DOWN
7	9	El-UP
8	1	Ground

Figure 3.4 presents the aforementioned main components connections considered to automatically control the rotor.

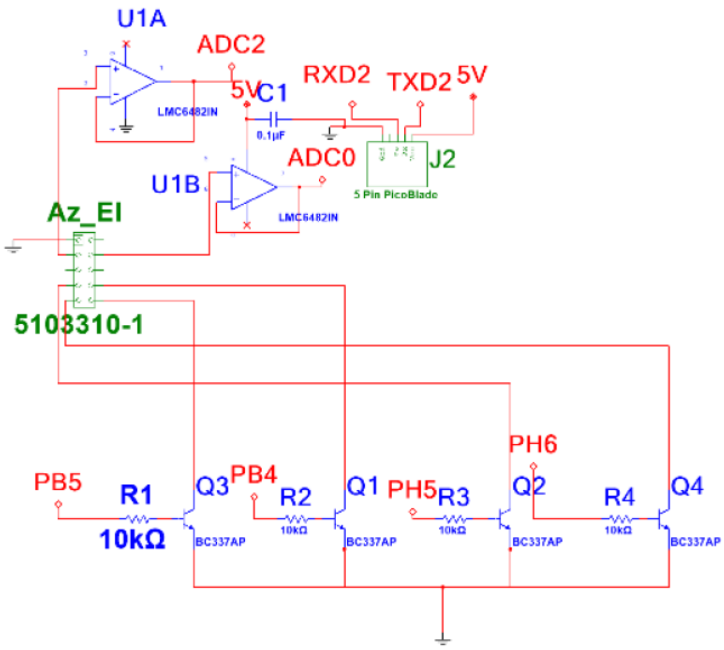


Figure 3.4: Rotor Controller - Main Schematic PCB Design

3.3.2.2 Microcontroller

The microcontroller board considered for the rotor controller design is an Elegoo Mega2560, based on the ATmega2560. This microcontroller includes more than 50 GPIO pins, some of them used to control the rotor Az/El movements while reading the actual position of the rotor. Moreover, it has 4 serial-UART independent communication ports, which can be used to communicate with the ground station user interface, as well as to get the actual position of the ground station from the GNSS receiver connected to a second UART without interruptions between both communications. The GNSS provided coordinates can be indispensable when launching from the field or when the ground station is mobile -used on top of a vehicle, while driving in the balloon direction-.

Figures 3.5 and 3.6 present a complete pointing control system, including a GNSS receiver and the connection to the rotor controller box.

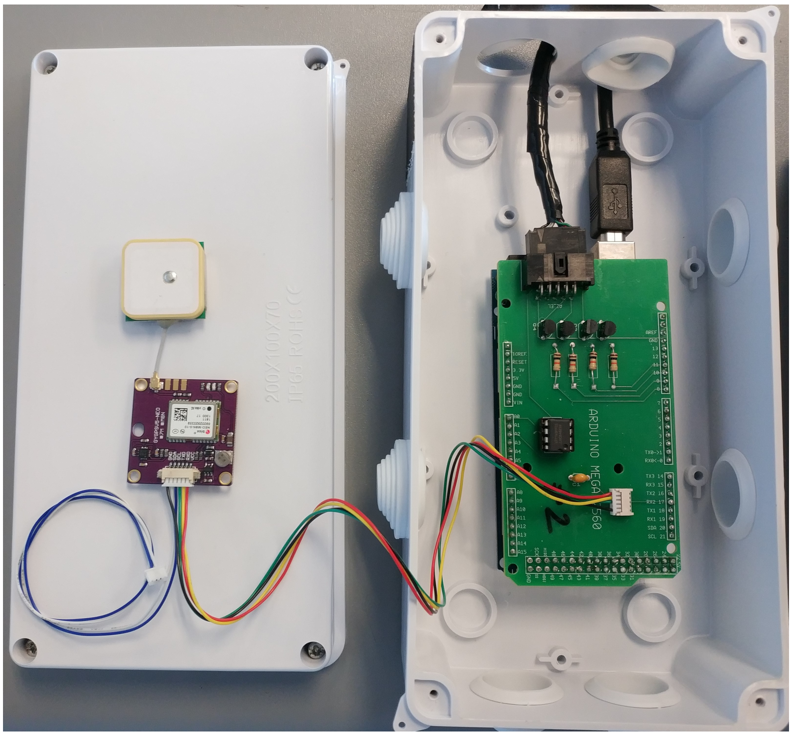


Figure 3.5: Rotor Controller - Real-Time GS Position

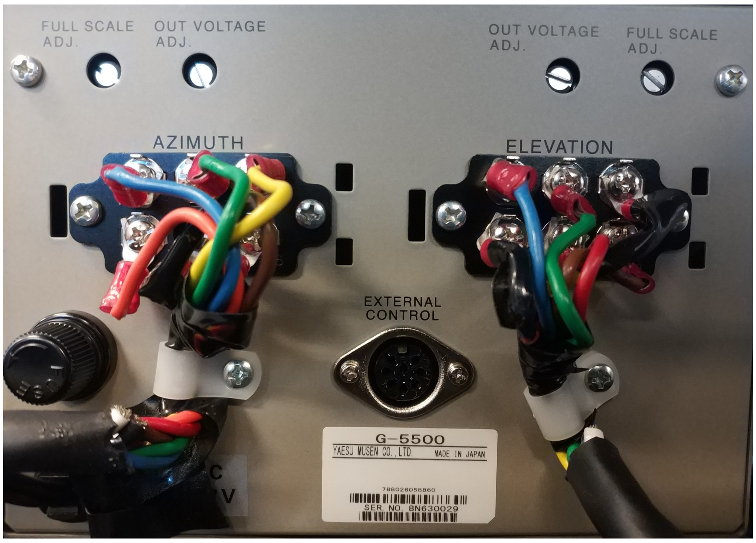


Figure 3.6: Rotor Controller - Box Calibration Adjustments

Two algorithms are used to complete the GS rotor software: (1) to calibrate the rotor signal levels and the gauges that can be seen in Figure 3.6, and (2) to control the rotor movements. The G-5500 rotor control box has a 8 pin DIN external control connection (see Figure 3.6) that controls the different movements by connecting them to the proper pins of the microcontroller shield (see Table 3.3). There are two pins that supply a DC voltage from 2 to 4.5 V corresponding to actual Az/El rotor position. The microcontroller will read them as analog readings that need to be converted using a rotor calibration procedure. Calibration information can be found in Appendix A, including hardware and software procedures.

For the flight code, the microcontroller will enable the proper azimuth and elevation signals (Up, Down, Left, Right, Off, presented in Table 3.4) based on the actual rotor position read from the analog pins and the desired position to point to. The connection of the ground pin to the respective control pin of the external control rotor connector on the G-5500 is accomplished by supplying a 5V DC signal -supplied by the microcontroller- to the proper NPN transistor. The transistor acts as a switch for each pin and/or movement, as it can be seen in Table 3.3. Only when the Az/El positions are at a certain margin (deadzone) from the expected position, the microcontroller will stop enabling the rotor movement. The "deadzone" is a buffer to prevent chattering of the rotor, since it cannot be continuously moving. Due to the movement limitation of the rotor and its duty-cycle, a 2° deadzone was chosen. Figure 3.7 shows the block diagram of the control logic for elevation movements. Azimuth movements are based on the same logic.

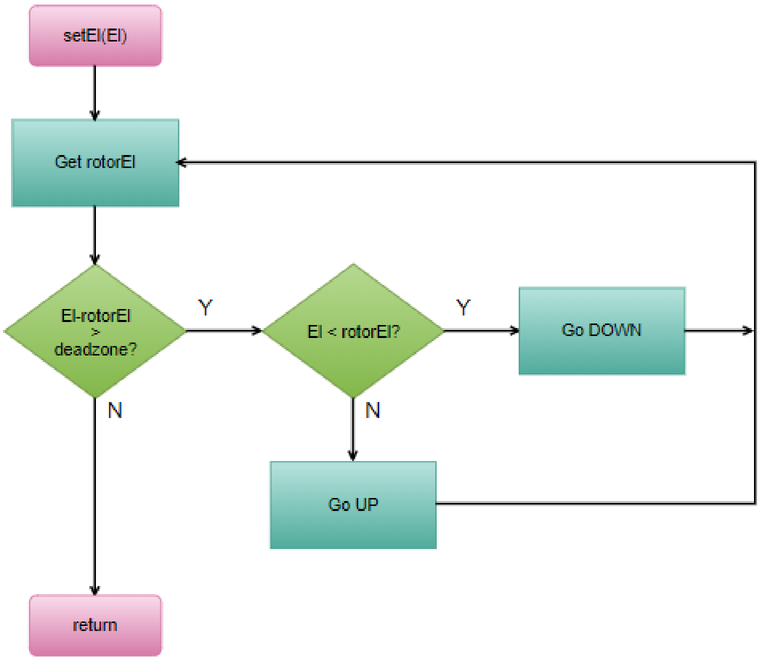


Figure 3.7: Rotor Controller - Control Logic

Table 3.4: Rotor Box Controller Cases

Case	**Pins ON**	**Pins OFF**
'off'	-	UP, DOWN, LEFT, RIGHT
'up'	UP	DOWN, LEFT, RIGHT
'down'	DOWN	UP, LEFT, RIGHT
'right'	RIGHT	LEFT, UP, DOWN
'left;	LEFT	RIGHT, UP, DOWN
'AZ off'	(UP, DOWN)	LEFT, RIGHT
'EL off'	(LEFT, RIGHT)	UP, DOWN

Table 3.5 presents the commands used in the control logic to get or set the ground station pointing parameters:

Table 3.5: Rotor Box Controller Commands

Commands	**Results**
setAzXXX	Set Azimuth to XXX
setElXXX	Set Elevation to XXX
AzElXXXYYY	Set Azimuth to XXX and Elevation to YYY
getAz	Return Azimuth Pointing Direction
getEl	Return Elevation Pointing Direction
getLoc	Retur LLA coordinates with the following format: '%lat, %lon, %alt'
intCal	Initiates the rotor calibration

In Appendix C, it can be seen how these commands are used by the MATLAB GUI implementation to control the rotor and track the payload.

Using those commands, the payload coordinates are obtained and used with the GS ones to compute and change the antenna pointing. As aforementioned, using a GNSS receiver, the GS coordinates can be computed by reading and decoding the proper NMEA messages. The same GNSS receiver is used for the payload design.

Figure 3.8 presents the permanent ground station control design used for the ERAU ground station. The GNSS sensor is not included, since it is a permanent ground station whose coordinates are static and known. More information about the ground station tracking system and GUI can be seen in next sections and Appendices B and C.

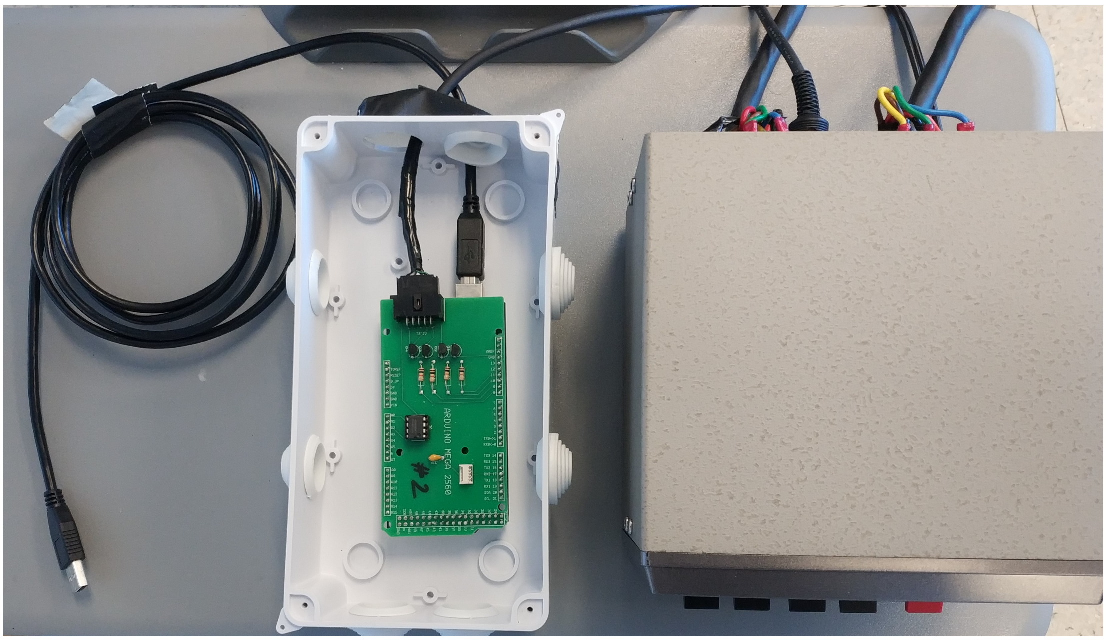


Figure 3.8: Rotor Controller - Arduino Shield and Rotor Controller Box

3.3.3 GS Graphical User Interface

MATLAB is a powerful tool with many toolboxes that makes it ideal for a ground station GUI. The Mapping Toolbox, the Aerospace Toolbox, and the App Designer all have functionality that makes a simple to use but powerful app to track the HAB and control the pointing of the ground station.

The ground station GUI designed and implemented for this project includes:

- Different modes to cover ground station pointing calibration and checks, real-time flight tracking and past flight data reproduction.
- Ground station control including different antenna tracking modes, with optional payload tracking using flight path predictions instead of position data from the payload on-board GNSS receiver.
- Pointing accuracy tuning during the launch (Az/El offsets).
- Predicted and real-time received sensors data plots, and 3D-2D maps with predicted path and real-time received payload position for tracking purposes.
- Payload tracking modes selection, from real-time sensors or using previously predicted flight path data in case of GNSS receiver failure.
- Percentage of data losses specification, in order to control the antenna pointing offset and to detect other possible communication problems.

Figure 3.9 presents an example of the GUI reproducing data from a past launch.

Acceleration and temperature data are plotted based on time and altitude, respectively. GNSS data is presented, as well as the GUI computed ascent/descent rates.

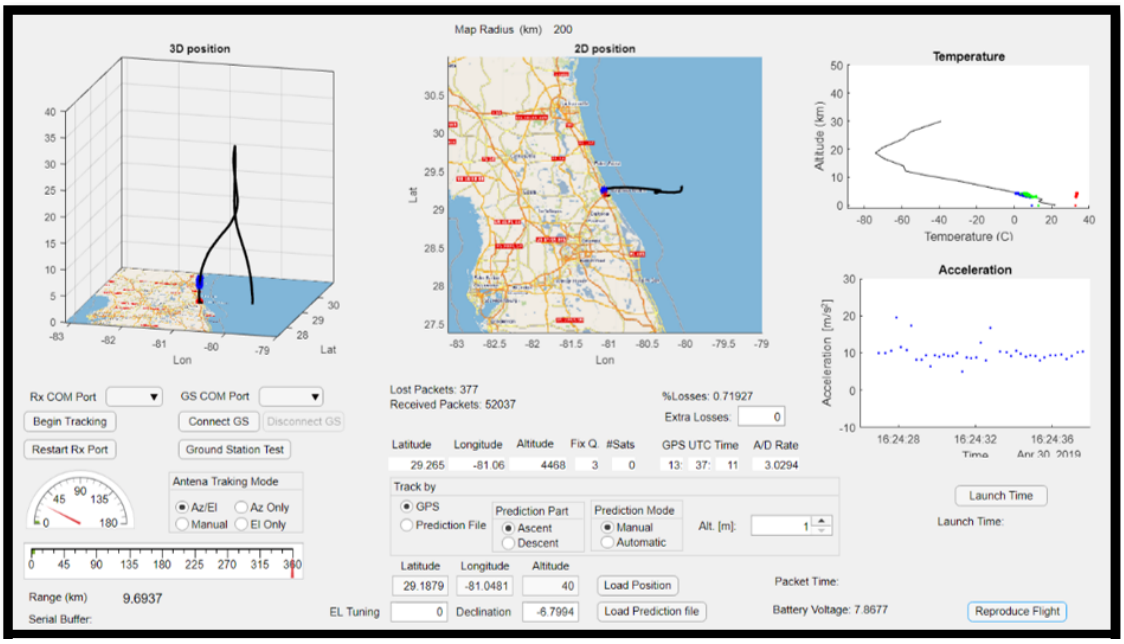


Figure 3.9: GUI - Reproduced Flight Data.

The communication link parameters are computed and presented in terms of received packets, lost packets and total percentage of losses during the launch. The gauges show the actual rotor pointing in case the user has no view of the ground station (i.e. using ERAU permanent GS from inside a building).

3.3.3.1 Modes of Use

The GS GUI of this project has three independent modes of use:

- **HAB Launch:** mode used to track a HAB payload in real-time while plotting the on-board sensors data to check the launch performance. For this mode, both the GS rotor controller and the GS transceiver need to be connected to the MATLAB interface using two different serial communication ports. The data is automatically plotted once a hard-coded amount of data is received and decoded. Not all the data from the packets received is plotted. All the GS tracking modes are available, and in case of unexpected ascent or descent rates, those differences can be afforded by uploading another flight prediction file computed with the proper rates. The last prediction file loaded to the GUI will be the one considered.

- **Ground Station Check:** mode used to check the GS pointing error and the pointing during the predicted flight path to confirm that the antenna will not be pointing to any structures around. For this mode, a prediction file and the communication with the GS rotor controller are required. Google maps can be used to find a land feature (tower, building, etc) within line of sight and determine its exact LLA coordinates. The prediction file will include the coordinates of those land features. Once the GS check mode starts, the prediction file line can be manually selected. Based on the GS LLA coordinates, the GUI will compute the Az/El for the antenna to point to the land feature selected. The rotor is then pointed to that Az/El. By editing the “Declination” field, an azimuth correction can be applied so that the antenna points exactly at the land feature.

- **Flight Reproduction:** mode in which the data of a past launch can be reproduced. In this mode, the same binary file that the GUI recorded during a past flight can be used to reproduce the whole data again. The speed of reproduction can be specified by adjusting the percentage of data samples plotted from the whole flight (i.e 1 out of 200 samples).

3.3.3.2 Predicted Sensors Data

There are available online tools that can predict atmosphere parameters based on altitude, such as temperature, pressure and wind speed. The implemented GS GUI can take a previously created file with those parameters to use them as predictions for the payload’s on-board sensors. This can be useful to monitor how well the sensors are performing during the flight, and to analyse the sensors accuracy during the post-processing of the recorded data.

For most of the HAB flights presented in this thesis, only temperature data profiles are considered. Only during the first flights, pressure data was also considered.

Due to the calibration difficulties of the pressure sensors and the cost of the ones that work in the altitude range of interest, it was decided to stop using them. More information about prediction data files can be found in Appendix C.

3.3.3.3 Tracking Modes

The GUI has a serial connection with the GS rotor controller previously presented. The GUI will use the GS and the payload actual position to obtain the azimuth and elevation coordinates that the antenna should point to at that moment. Considering the antenna tracking mode selected in that instant, the GUI will send the proper command to the rotor:

- **Az/El:** the GUI sends a command with the azimuth and elevation coordinates to point to.
- **Az Only:** the GUI sends a command with only the azimuth coordinates to point to.
- **El Only:** the GUI sends a command with only the elevation coordinates to point to.
- **Manual:** the GUI does not send a command to the rotor controller. The rotor is moved manually.

The GS position can be a single input when starting the GUI or it can be decoded to be updated in real-time during the flight, if the GS is continuously moving. The payload position can be obtained from the on-board GNSS sensor or from a previously made file with the flight path prediction:

- **GNSS Tracking:** when a data packet with information from the GNSS information is received, the GUI decodes the payload's latitude, longitude and altitude (LLA) coordinates and plots them on the user view, converts them to azimuth and elevation coordinates taking into account the actual position of the ground station antenna, and sends the proper command to the GS rotor controller, considering the selected antenna tracking mode at that moment.
- **Prediction File Tracking:** the GUI will consider the prediction file payload's coordinates to compute the antenna pointing coordinates. This can be done manually, by specifying the part of the flight and the altitude of interest, or automatically, in which case the GUI will use the launch time to compute the actual time of the flight and it will use the coordinates closest to that time. In prediction file mode, the GUI will check and send new coordinates to the rotor after a certain amount of time, controlled by a previously programmed timer.

For the prediction file mode, the GUI will consider a previously loaded .csv file with latitude, longitude, altitude and time of the flight parameters. In this case, an online available tool is used to create these files. More information about how to create them can be found in Appendix C.

3.4 Payload

The HAB payload is the part of this project that was being updated the most during the process of this thesis. The next sections will present the main design stages that were considered when developing the payload to meet the project requirements. In order to set the base of each design, the main considered subsystems composing this HAB payload are summarized below.

3.4.1 System Overview

The designed HAB payload consists of the following parts or subsystems (see Figure 3.10) :

- **Payload Controller:** in charge of controlling and performing all the tasks from the payload and, if needed, the controlled descent system.
- **Communications:** module consisting of the transceiver and the antenna used to send the data to the GS.
- **Position Tracking:** GNSS receiver used to get the payload coordinates during the launch.
- **Scientific Data:** data gathered from the on-board sensors that does not have another purpose inside the payload.
- **Data Backup:** SD card module to save each data packet of interest created during the launch, in case a payload recovery is possible.
- **Controlled Descent:** payload module or independent system in charge of ensuring a slow/controlled descent of the payload back to the ground, non considering balloon bursts.
- **Power System:** battery or batteries used to power the payload and the controlled descent system. It should supply enough power to support at least a 5 hour launch (2 hours for the controlled descent system if it is external to the payload).

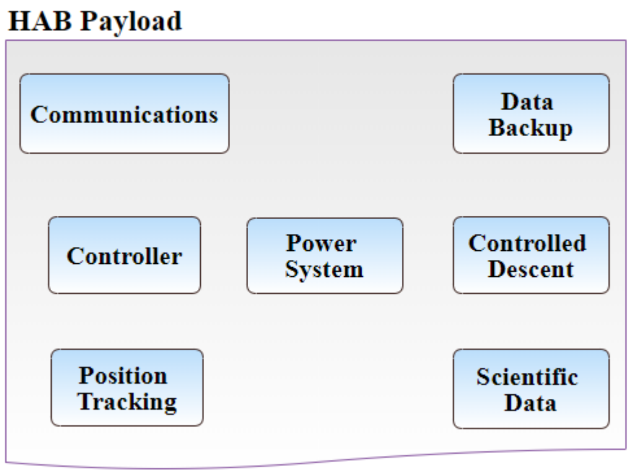


Figure 3.10: HAB Payload - System Overview Block Diagram.

3.5 Design Stage 1

3.5.1 Design Overview

From the first stage of the payload design, the following key parts should be considered:

- ISM 900-928 MHz band chosen for the communication link between HAB and GS using DNT900 transceivers.
- Dipole antenna - linear polarization in the payload.
- Redundancy in payload position tracking: multiple GNSS receivers used to determine which model would work above 18 km -address the COCOM limits[22]-.
- Controlled descent based on a double-balloon configuration and a cutting-thread system included within the payload.
- Data of interest collected: internal and external temperatures, pressure data, and acceleration and angular velocity of the payload.
- ATmega2560-based payload controller.
- SD card module included for data backup.
- Power budget and first launch analysis made.

Figure 3.11 presents the payload block diagram.

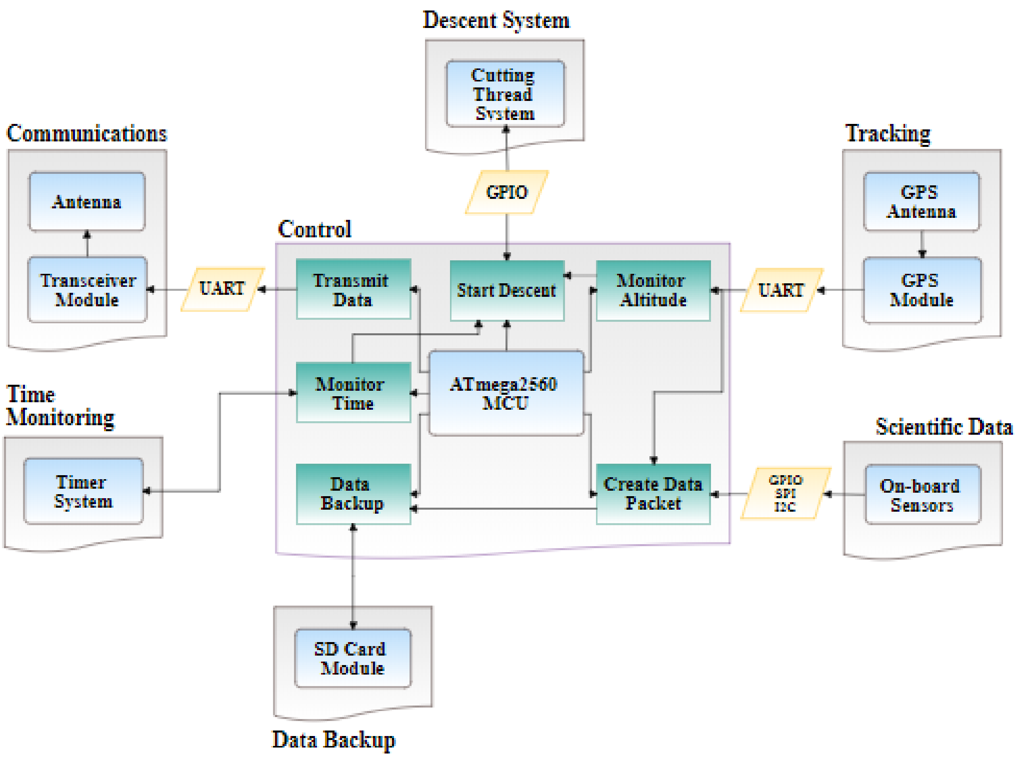


Figure 3.11: Design 1 Block Diagram - Arduino Mega and Internal Cutting System

3.5.2 Payload Controller - ATmega2560

The Elegoo Mega2560[23] is a board based on the ATmega2560 chip. It was selected due to the availability of more than one serial port for GNSS receivers and transceiver communication, as well as for its relation of performance vs cost.

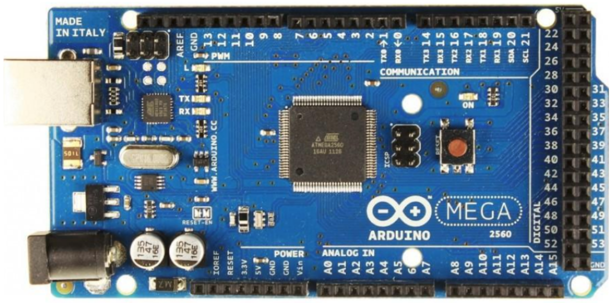


Figure 3.12: ATmega2560-based microcontroller board.

Figure 3.12 presents a sample of the selected microcontroller, which specifications can be found in Table 3.6:

Table 3.6: ATmega2560 board specifications.

Parameter/Specification	Value
Operating Voltage	5V
Clock Speed	16 MHz
Digital I/O pins	54
Analog Inputs	16
UART/Serials	4
Flash Memory	256 KB
EEPROM	4 KB

In this payload design, the microcontroller is getting data from the sensors all the time, while checking if there is data from the GPS available. Based on the available data, a new data packet is created and sent to the radio buffer as well as saved in the SD card for backup purposes. To save time, the data is actually written to the SD card once/twice per hour. All the packets have a size of 100 bytes, with a 2 bytes packet ID to differentiate the packets with GNSS content and identify them during post-processing. A packet counter ID to identify lost packets or packets with errors is added too, as well as a time stamp created by the microcontroller. The packet content can be changed in terms of type of data and order. The length of the packet should be 100 bytes and the ground station has to be changed according to the expected order of the data. If not, the transceiver configuration needs to be changed accordingly.

The controller uses the position of the payload to check if a certain altitude has been achieved and to activate the controlled descent system, if required. More information about the code implementation used in this design can be seen on Appendix H.

3.5.3 Payload Position - uBlox NEO M8N and Trimble Copernicus II

In this design stage, the COCOM limits for GPS technologies were analysed. The COCOM limits refers to a limit placed on GPS tracking devices that disables tracking when the device calculates that it is moving faster than 1,900 km/h at an altitude higher than 18 km in order to prevent the use of GPS in intercontinental ballistic missile-like applications. Even though the speed is not a problem that needs to be addressed in this project, there are several GNSS receivers whose maximum working altitude is below 18 km due to these limits.

The GNSS receiver models tested for the payload position tracking were the uBlox NEO M8N[24] and the Trimble Copernicus II[25] models. Both GNSS receivers are versatile modules that provide high sensitivity, customizable configurations and an altitude operational limit of 50 km, which makes them suitable to be used in this project. The main specifications to consider when using the receivers that are presented in Figure 3.13 and configuring them can be found in Table 3.7.

uBlox Center and Trimble Studio are available evaluation software to easily configure these GNSS devices. Once the desired configuration is saved, the modules include an extra battery to be able to maintain the same configuration for long periods of time. More information about how to properly configure this receiver for the HAB launches can be found on Appendix B.

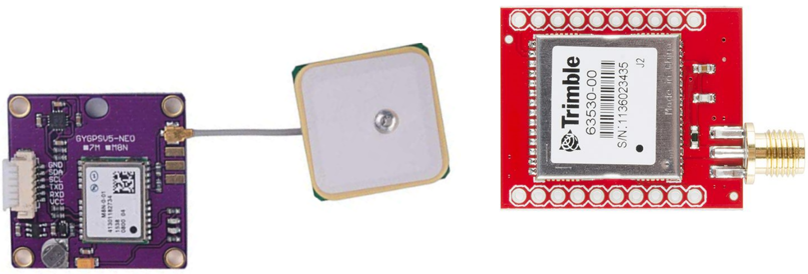


Figure 3.13: Design 1 - (L) uBlox NEO M8N and (R) Trimble Copernicus II GNSS Receivers.

Table 3.7: uBlox NEO M8N and Trimble GNSS receivers specifications.

Parameter/Specification	**NEO M8N**	**Copernicus II**
Horizontal Accuracy	2.5 m 50%	2.5 m 50%
Vertical Accuracy	5 m 90%	5 m 90%
Maximum Navigation Rate	5 Hz[1]	1 Hz
Configurable Constellations	GPS, GLONASS, Galileo, Beidou	
Power Supply	3.3 V	3.3 V
Max. Supply Current	20 mA	40 mA

[1] 10 Hz if only 1 constellation is considered.

3.5.4 DNT900 Transceiver

The 900 MHz transceiver considered in this design was the DNT900 from muRata [26]. This transceiver module is a low-cost, high-power solution for wireless data communications in the 900 MHz ISM band. The package selected of this transceiver for both the payload and the GS was the development board that can be seen in Figure 3.14.

The development board includes all the required pins to communicate and perfectly test the module, which makes easier its validation during the payload tests. Among other things, the development board has LED indicators of signal strength and RX/TX indicators. Using these indicators it could be checked if the board was sending ACK signals or not, a key point for this communication link, considering the high-gain antennas used in the GS segment. Table 3.8 summarizes the most important specifications of this transceiver module.

Once configured, to maintain a communications link between GS and payload, only 3 pins from the board are required: GND, RX and TX. Even though the communication link used in this project scope is only used in one direction, from the payload to the GS, it can be possible to send data from the GS to the payload. To do that, specific RF conditioning is required so the GS is compliant with the band regulations when transmitting data.

Figure 3.14: DNT900 (L) Development Board, (R) Transceiver Module

Table 3.8: DNT900 transceiver specifications.

Parameter/Specification	**Value**
Operating Frequency Range	902.75-927.25 MHz
Modulation	FSK, FHSS
RF Data Tx Rates	38.4, 115.2, 200 and 500 kbps
Sensitivity @200kbps	-98 dBm
Max. RF Output Power @200kbps	1 W
Antenna Connector	u.fl
Network Topology	P2P, P2M, Peer-to-Peer, Tree-Routing
Power Supply Range	3.3-5.5 V
Peak Tx Mode Current	1.2 A

3.5.5 Data Backup:

Since the microcontroller used in this design did not include a built-in SD card slot, the external SD module with SPI communication presented in Figure 3.15 was selected.

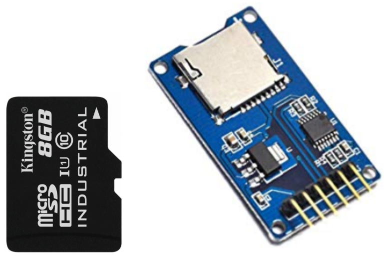


Figure 3.15: (L) Industrial Range SD Card, (R) SD Card Module.

The SD card used was a Kingston of 8GB with an industrial temperature range[27]. 8GB of capacity were chosen because they were enough for our data link requirements, while the industrial temperature range was selected to assure a complete data backup even if the internal temperature of the payload was colder than expected.

3.5.6 On-board Sensors

3.5.6.1 Temperature Sensors

During the launches, the internal temperature was recorded for monitoring, while the external one was used for science purposes to determine accurately the temperature at different altitudes of the stratosphere and the path followed by the payload. To measure them, the thermistor model PR103J2[28] was selected, a NTC 10KΩ with a resistance @25°C of 3892KΩ, with a temperature working range between -55 and 80°C and a maximum accuracy of 0.1 °C. A thermistor resistance changes with temperature changes. Based on that, a voltage divider presented in Figure 3.16 was created in order to be able to measure the resistance through those thermistors at the temperature range of interest. To do that, the ADC specifications of the payload's microcontroller need to be considered to accommodate the temperature range to the voltage range of the ADC. More information about how the voltage divider was designed and calibrated can be found on Appendix D.

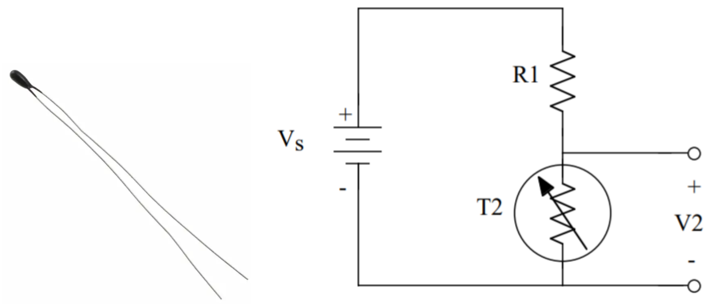


Figure 3.16: Design 1 - (L) PR103J2 thermistor and (R) voltage divider circuit.

3.5.6.2 Acceleration and Angular Velocity.

The movement of the payload was recorded and used to identify possible balloon bursts and to analyze the performance of the controlled descent system. The selected sensor to record acceleration and angular velocity was the LSM9DS1 [29], a single chip that includes an accelerometer, a gyroscope and a magnetometer -nine degrees of freedom (9DoF)-. Each sensor can be configured with a different range and two different communication systems (I2C, SPI) can be used to obtain the data. Section 5 presents how the data obtained during the launch was analysed. Appendix G presents a system to understand the sensor readings and movements.

3.5.6.3 Pressure.

Pressure data at different altitudes in the stratosphere is scientific data of interest for this project. For this design, a Honeywell ASDXACX015PAAA5[30] pressure sensor was selected with a maximum pressure rating of 30PSI (206.84kPa) and an accuracy of 2%. The sensor was calibrated in a vacuum chamber, showing some limitations for low pressure ranges. Due to that, it was decided to add an operational amplifier to amplify that range of measurements, always taking into account the limitations of the ADC of the microcontroller. Using an available online tool, the temperature and pressure profiles for the launch date and time were predicted. That data was used during the flight to analyse if the sensors were working as expected. More information about these predictions can be seen in Appendix C.

3.5.7 Controlled Descent - Internal Cutting System

To lift the payload during the launch, the balloons are attached to the top face of the payload. Considering that the payload box is made out of styrofoam, 3D printed support pieces are used to avoid breaking the whole lid.

In this double-balloon configuration, the controlled descent is achieved by cutting one of these balloons once a certain altitude is reached. To do that, the support pieces of the balloons are separated, being the permanent balloon on the center of the payload, and the balloon that is going to be cut in an outer position for payload stability, as it can be seen in Figure 3.17.

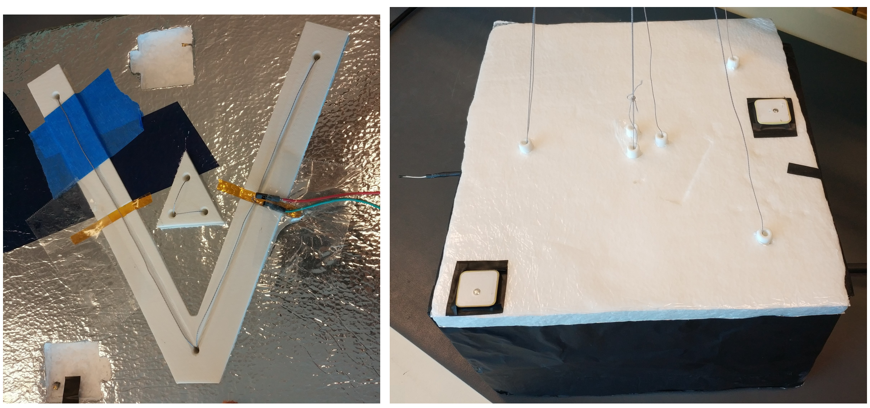


Figure 3.17: Design 1 - Controlled Descent System Sample.

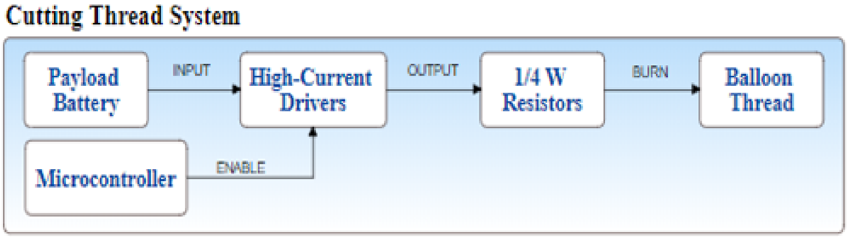


Figure 3.18: Design 1 - Cutting System Logic.

To cut the balloon threads, a cutting system mechanism is implemented, based on a SN745510NE[31] H-driver connected to the battery that once is enabled it outputs enough current -approximately 1-1.5A- to extremely heat a nichrome wire or a 10Ω low power rate resistor -0.25W- connected to the balloon thread. This system is enabled by the payload controller after a certain altitude is achieved. Figure 3.18 presents a block diagram of this cutting system.

Once the balloon is cut, the payload starts descending with the balloon left. To be able to approximate the ascent and descent rates, a predictor for HAB is used[32]. In this predictor, the type of balloon, the mass of the payload and the expected ascent rate can be specified as inputs. The descent needs to be approximated by the amount of payload weight the permanent balloon is not able to lift, which would be as small as possible.

3.5.8 Power Budget

Table 3.9 presents the power budget of this design. Considering the heat produced by the power dissipation of the transceiver in transmitting mode 100% of the time, the internal temperature of the payload is not considered a factor of negative impact in the performance of the battery used. For worst case scenarios, a 90% efficiency is considered, with a result of 5.36 hours of capacity for the payload.

Table 3.9: Design 1 - Power Budget.

Component	**Current Consumption**	**Voltage Supply**	**%Use/Hour**
Transceiver	900	3.3	100
Microcrontroller	20	5	100
GNSS Receiver	20	3.3	100
9DoF	4.6	3.3	100
Cutting System	1500	7.4	0.041
H-Bridges	25	5	100
SD Card Module	100	5	0.7
Pressure Sensor	2.5	5	100
Total Consumption/Hour	-	-	1040.807 mAh
Total Battery Capacity	-	-	6200 mAh
Total Capacity in Hours	-	-	5.96

Figure 3.19: Design 1 - Fluoreon 7.4V 6200mAh.

The total battery capacity is specified by the battery selected for the system. A rechargeable Fluoreon 7.4V 6200mAh battery[33] was chosen for the payload design.

3.5.9 Design Performance Results

- The controlled descent design was not working. Based on the launches analysis, the cutting system was being activated when expected, but the two balloons lines were entangled.
- Even configured in transparent mode, the DNT900 transceiver of the GS was sending acknowledgment signals to the payload, making the communications system not compliant with EIRP regulations for that band.
- The DNT900 transceiver was discontinued, so it needed to be changed.
- The GNSS receivers configuration was fully checked and the payload was able to be followed during the whole launch duration without errors. Both models were working at altitudes above 18 km (approximately 33 km).
- The pressure sensor data was too noisy for low pressure ranges due to its limitations. Due to the cost of pressure sensors presenting high accuracy at those ranges it was decided to stop working with that data.
- The external temperature data showed a profile similar to the predicted one. The accuracy below -50°C was too low.
- The achieved throughput was about 60 kbps, so the code efficiency needed to be improved, considering that the radio was configured at maximum capacity.
- Some sensors were disconnected during the flights due to the movements of the payload because the connections were not secured.
- The ground tests confirmed that the payload was able to work for 5 hours with the chosen battery, and the longest launch with this design had a duration of almost 6 hours, confirming the power budget results.

Figures 3.20 and 3.21 present examples of payloads of this design stage.

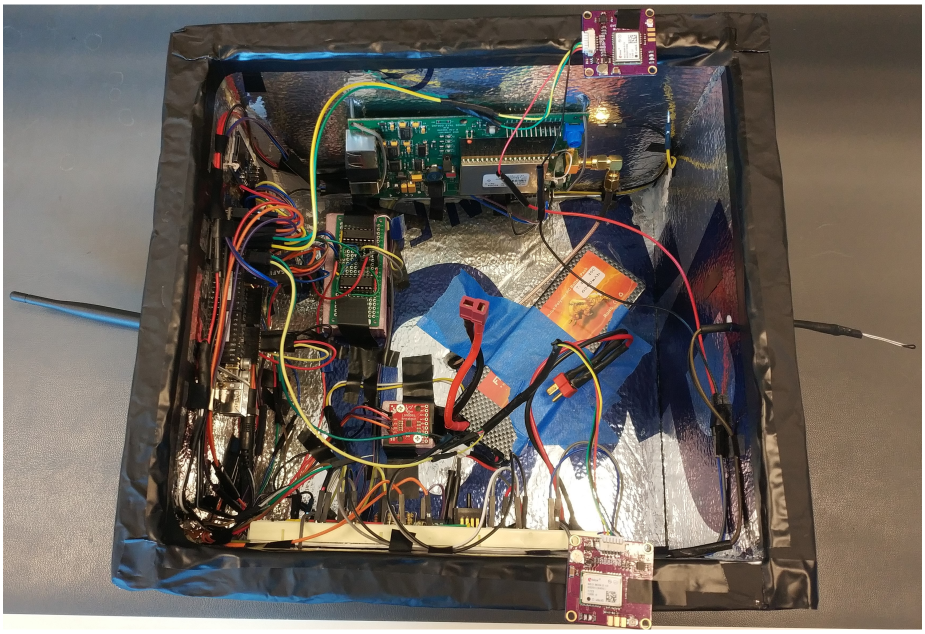


Figure 3.20: Design 1 - Final payload design sample.

Figure 3.21: Design 1 - Double-balloon launch.

3.6 Design Stage 2

3.6.1 Design Overview

The following items are the main updates and upgrades of this design stage.

- The 3D printed support pieces for the controlled descent system were modified. The outer V shape was maintained but in this case was used for the permanent balloon, while the center piece was changed to a diamond shape to get more support and be easier to connect both resistors to the same line. Another design tested was connecting 3D support pieces at the bottom and top faces of the payload (see Figure 3.27).

- Due to the broadcast problems detected when using the DNT900 transceiver and the fact that it was being discontinued, it was changed for the next best module for our design requirements: the XBee PRO SX from Digi [34].

- Considering that the payload position tracking performance was validated and perfectly working for both GNSS receivers in previous launches, a single GNSS receiver was used for next designs.

- Considering the cost difference and the maximum rate of the sensors, uBlox NEO M8N was selected as the GNSS receiver for the MURI HAB payload designs.

- To avoid components getting disconnected due to payload movements during the launches, a microcontroller shield to solder all the main payload components was included in the next design were every connection and sensor was soldered and secured. (see Figure 3.26).

- The external temperature circuit design was changed to achieve a lower temperature range -between -20 and -65 °C-, considering the Z curves of the thermistor and the resistance value at room temperature (R_{25}). For more information, see Appendix D.

3.6.2 XBee PRO SX Transceiver

The transceiver model used for this high-altitude balloon project is the XBee PRO SX. This transceiver is able to transmit up to 1W [30 dBm] of power at a throughput up to 120 kbps using GFSK modulation and FHSS spreading technology [34].

XBee modules are a product from DIGI, which provides a free, multi-platform application to configure their XBee/RF solutions: XCTU[35]. XBee PRO SX is configured with XCTU with a transparent protocol, to be able to control the exact amount of data sent at any moment, and to avoid the used of headers or extra unnecessary information. Moreover, the payload module is configured with a point to multipoint/broadcasting mode to be able to track the same payload with different ground stations sharing the same network.

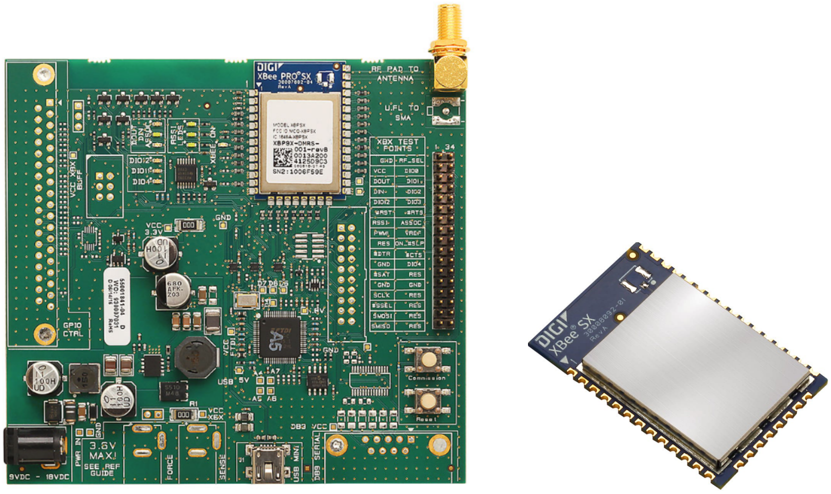


Figure 3.22: XBee PRO SX (L) Development Board, (R) Transceiver Module

The transceiver used can be found on different packages and modules. Even though it can be found as an Arduino shield, these modules shall not be considered because Arduino boards are not able to supply enough current to achieve the RF power output of 1W. The packages considered for this project are the development boards with external pin connections and the radio surface mount module with U.FL antenna connector that can be seen in Figure 3.22.

A DIGI development kit was considered, including: (1) two development boards with XBee PRO SX soldered (2) one extra chip from another XBee model (3) an interface board where transceiver surface mount modules can be attached and (4) antennas, cables, and power supplies for the boards that will be used for the transceivers configuration and usage.

3.6.2.1 Ground Station Board

For the ground station segment, one of the development boards from the kit is considered.

This module can be powered using the USB connection. The transceiver consumes approximately 40 mA when operating only in receiving mode, which can be supplied by a USB connection with the GS laptop. The development kit includes a mini-B USB to USB cable, which can be extended with an active USB cable if needed.

It should be noted that the antenna connector is a female RP-SMA. In most cases, the commercially available RF cables will include SMA connections; therefore, a proper adapter from SMA M/F to RP-SMA F is required.

If needed, the Tx/Rx lines of this board can be externally tested with a microcontroller using VCC, GND, DIN and DOUT pins. The board also includes indicator LEDs for power [XBEE ON], TX [DOUT] and RX [DIN] checking, as well as a group of three LEDs that work as received signal strength indicator [RSSI]. Figure 3.23 presents the indicators of the board.

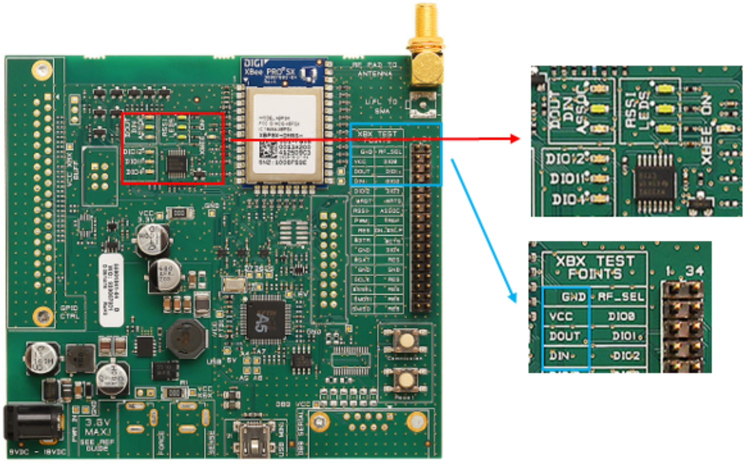


Figure 3.23: Ground Station Transceiver Module - Development Board.

The extra LEDs are GPIOs [DIO11/DIO1] and PWM [DIO12] indicators.

3.6.2.2 Payload Surface Mount Module

For the payload, in order to save weight and space, only the surface mount module from Figure 3.24 is considered. The package of this module considered for the payload includes a u.fl antenna connector.

Figure 3.24: Payload Surface Mount Module with u.fl antenna connector.

The only pins to be considered in this chip are VIN, GND, DIN, DOUT and CTS. Table 3.10 present the usage of those pins:

Table 3.10: XBee SM module pin specifications.

Pin	**Usage/Considerations**
VIN	3.3-3.6V (battery voltage regulated - LM317A)
GND	All the GND pins
DIN	Data to transmit
DOUT	Data received
CTS	(Clear to Send) Data Control Flow

To be able to configure this chip, the module shall be connected to a laptop. To do that, the configuration/interface board from the same company presented in Figure 3.25 can be used:

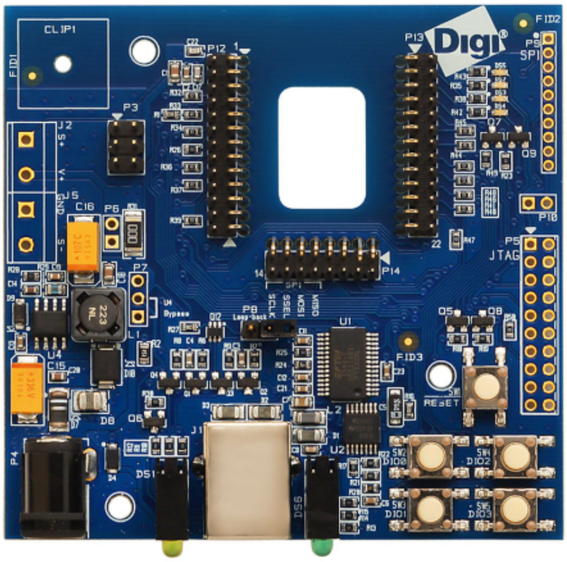


Figure 3.25: Digi Configuration/Interface Board for Surface Mount Chips.

More information about how to use this board and configure the chip can be found in Appendix E.

3.6.3 Design Performance Results

- The XBee PRO SX transceiver was able to broadcast the data when configured in transparent mode and a higher data throughput was achieved (~80 kbps).

- The tracking system worked with a single receiver, but for some launches the prediction file mode of the GS was used and confirmed to work as expected in case of GNSS receiver failure.

- The percentage of losses was acceptable and fairly low, but it was concluded that in some positions between the payload and the GS, the beam pattern of the antenna was causing more packet losses. The linear polarization and position of the dipole antenna used presented a high percentage of losses when the balloon was ascending in a position on top of the GS antenna (i.e. elevation angle close to 90 degrees).

- A processor with Floating Point Unit (FPU) and higher clock speed became one of the other universities requirements in order to properly work with their sensors. The Atmega2560-based microcontroller needed to be changed.

- The cutting system was not always working, even using different mechanism to avoid entanglements.

Figures 3.26 and 3.27 present examples of a payload and a controlled descent unit of this design stage.

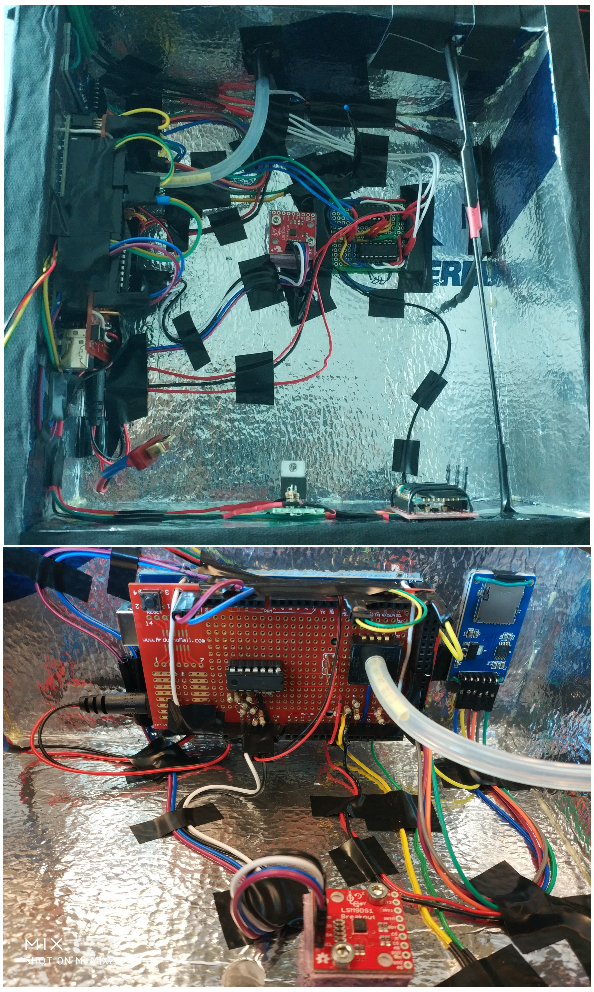


Figure 3.26: Design 2 - Final Payload Design Sample.

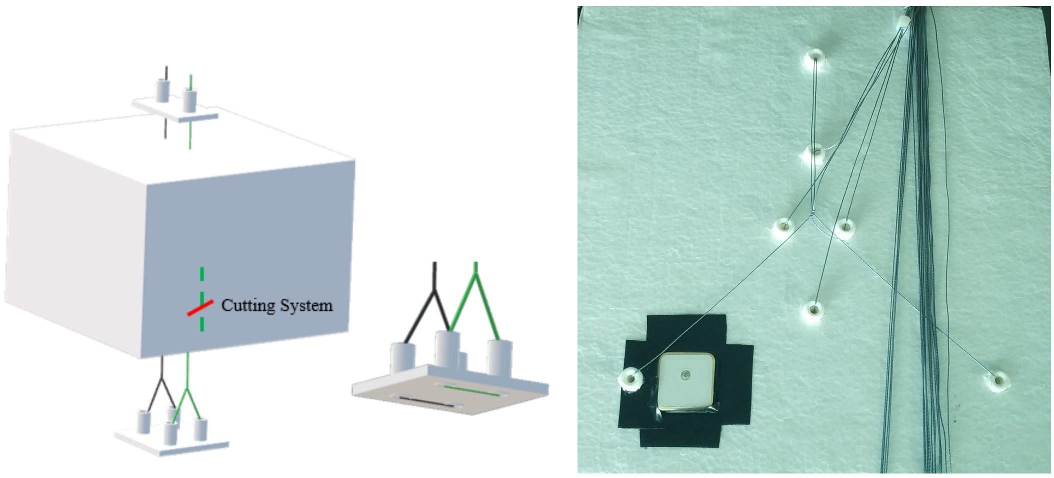


Figure 3.27: Design 2 - Controlled Descent System Mechanism Samples.

3.7 Design Stage 3

3.7.1 Design Overview

The following items are the main updates and upgrades of this design stage:

- The dipole antenna was changed for a cloverleaf antenna with circular polarization to afford payload movements and polarization mismatches with the GS linear polarization used. A ground plane was added to the bottom face of the payload to improve the directivity of the new antenna in the direction of interest.
- An external and independent controlled descent unit close to the neck of the balloon was designed for a single balloon configuration to avoid entanglements, base on a cap released mechanism.
- The payload controller was changed for a Teensy 3.5 ARM Cortex M4 board with FPU. This change will suppose the need of a board where all the components can be soldered, since the board is too small and there are not available shields, as well as another battery voltage regulation to supply the board with 3.6-6 V.

Figure 3.28 presents the block diagram of the payload and the controlled descent unit for this design stage.

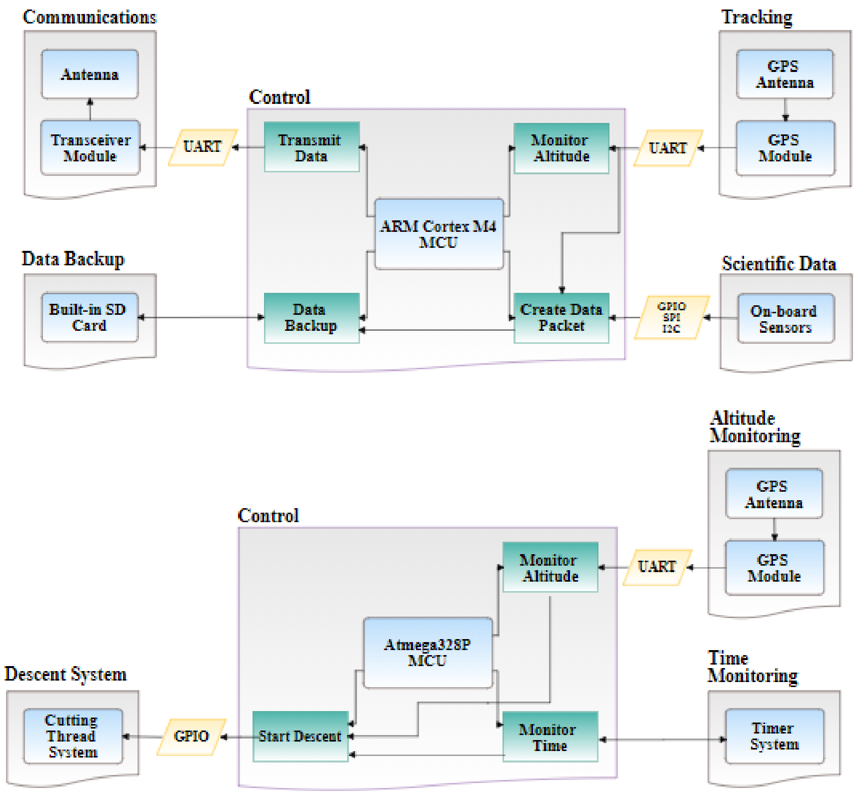


Figure 3.28: Design 3 Block Diagram - (A) Teensy 3.5-based main payload and (B) Atmega328P-based independent/external controlled descent unit.

3.7.2 Teensy 3.5 ARM Cortex

Figure 3.29: Design 3 - Payload Controller: Teensy 3.5 ARM Cortex M4 MCU.

Table 3.11 presents the main characteristics of this board:

Table 3.11: Design 3- Teensy 3.5 board specifications.

Parameter/Specification	**Value**
Operating Voltage	3.3V
Clock Speed	Up to 120 MHz
Floating Point Unit	Included
Digital I/O pins	62
Analog Inputs	25 (13 bit resolution)
UART/Serials	6 (2 with FIFO and Fast Baud Rates)
Flash Memory	512 KB
RAM	256KB
EEPROM	4 KB
SD Card Port	Included

In this design, the payload's controller was changed for a Teensy 3.5[36][37] development board. Teensy comes pre-flashed with a bootloader, so it can be programmed using the on-board USB connection: no external programmer is needed. With the Teensyduino add-on for the Arduino IDE, Arduino sketches can be adapted to be used on this board, which made easier the adaptation from the previous payload controller to this one presented in Figure 3.29.

For approximately $10 more, the payload controller can be upgraded considerably to Teensy 3.5. To power this module, a preliminary board with the battery voltage regulated to approximately 4.5 V for Teensy and 3.5V for the XBee module was made. In that board, all the required payload's connections are included, since the microcontroller shield was not an option anymore.

3.7.3 Payload Antenna - Cloverleaf

From movement data and link performance results obtained from previous launches, it was concluded that a circular polarization in either the GS or the payload was required to improve the overall launch results. To have circular polarization on the ground station, a second Yagi antenna was required, one for vertical and another for horizontal polarization, with a perfectly 90° phase between the antennas, because a circularly polarized antenna working at 900 MHz with similar specifications was not commercially available.

Figure 3.30: Design 3 - Payload Antenna: Cloverleaf Antenna 3 and 4 leaves.

On the other hand, it was easier to add circular polarization to the payload without compromising the previous antenna gain (dipole antenna - 0 dB). Cloverleaf antennas were concluded to be the best commercially available option. For a similar or even cheaper cost than the previously used payload antennas, a circularly polarized cloverleaf antenna with 5 dB of gain from Hobby King[38] was perfect to fit in our payload design. Figure 3.30 presents the selected antenna.

Both antennas designs -with 3 or 4 lobes- are suitable for the payload design in terms of total radiated power compliance and have similar characteristics. 3 lobe antennas usually are a better matched to the 50Ω transmitter which means less reflected power, while 4-lobe antennas have better polarization characteristics. In terms of transmission, both of them can have its pros and cons, but both of them were tested during actual HAB launches and no difference was appreciated when used with the linearly polarized Yagi antenna of the GS.

Considering that the radiation pattern of these antennas is similar to the dipole antenna one (see Figure 3.31), a ground plane was added to the bottom face of the payload in order to improve the directivity in the direction of interest [39].

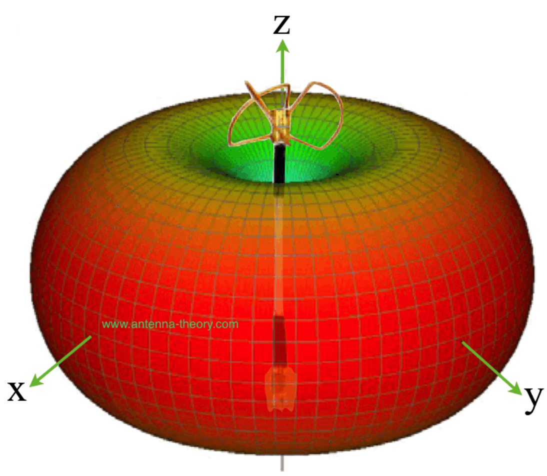


Figure 3.31: Design 3 - Payload Antenna: Cloverleaf Antenna Pattern[40].

As it can be seen in Figure 3.32, the radiation pattern changes depend on the distance between the antenna and the ground plane:

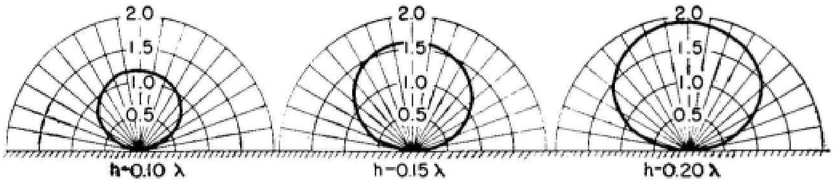


Figure 3.32: Design 3 - Ground plane effects on dipole antenna pattern[41].

Taking into account the cloverleaf position in the payload, the antenna pattern will be affected horizontally and the quality of the ground plane -dirt, imperfections- will be a key point to take into consideration, as it can be seen in Figure 3.33:

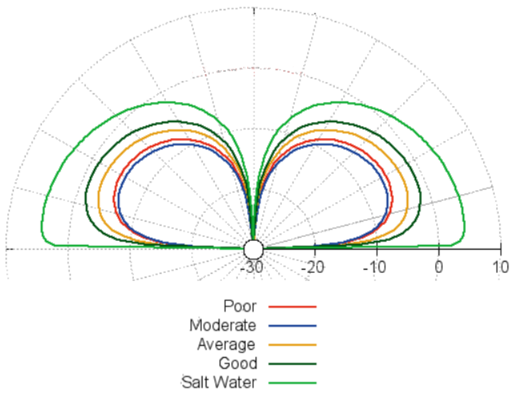


Figure 3.33: Design 3 - Ground plane quality effects on radiation pattern [42].

Considering the previously described effects, a distance of approximately 0.2λ was maintained between the ground plane and the payload antenna using a 3D printed support.

3.7.4 Controlled Descent - Independent Cap System

Considering the entanglements suffered when using the previous cutting system designs, a system independent and external from the payload was designed. This independent unit was based on the previous cutting-thread system with a dedicated ATmega328P-based[43] microcontroller, as it can be seen in Figure 3.28. It still uses the same logic, monitoring time and altitude to decide when it is the right moment to activate the cutting-thread system, but for a different purpose.

Since single balloon launch configurations were also something to consider due to the reduction of the overall launch cost, this independent system was designed to be used with only one balloon. To do that, and considering that the goal was to achieve a slow descent, instead of cutting the actual balloon line, the system was cutting the only thread holding in place a cap attached to a pipe that was connected to the balloon neck (see Figures 3.34 and 3.36).

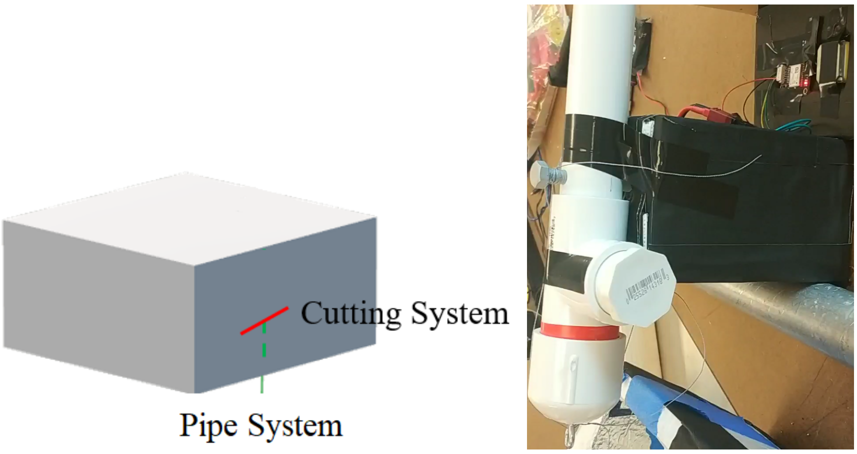


Figure 3.34: Design 3 - Controlled descent unit design. (L) The thread connected to the pipe system passes through (R) the hook at the bottom of the cap and it is attached to the screw of the pipe. Once cut, the cap is released with the force of a spring included inside.

The system required its own power supply for at least 2 hours and being able to supply enough current to cut the thread after that time. Fluoreon batteries were considered for this unit too, due to their excellent performance during the previous HAB launches. A 7.4 V and 1500 mAh rechargeable Fluoreon battery [33] was chosen for this unit. The power budget computed for this unit is presented in Table 3.12:

Table 3.12: Independent Controlled Descent System - Power Budget.

Component	**Current Consumption**	**Voltage Supply**	**%Use/Hour**
Microcontroller	20	5	100
GNSS Receiver	20	3.3	100
Cutting System	1500	7.4	0.041
H-Drivers	25	5	100
Total Consumption/Hour	-	-	65.61 mA
Total Battery Capacity	-	-	1500 mAh
Total Capacity in Hours	-	-	22.86

Considering that this is an external system that will have to handle extremely low temperatures, the battery efficiency/performance will decrease. However, even considering an efficiency of 50%, the total capacity would be 11.43 hours, more than enough for the maximum expected launch duration.

3.7.5 Design Performance Results

- The payload design was completely adapted to the new microcontroller, achieving high data throughputs of approximately 100 kbps.
- The communications link was maintained up to slant ranges of 150 km with at least a total throughput of 90 kbps.

- The cutting system design presented design problems. Since the system was completely independent, it was fully tested in a temperature chamber to confirm its correct performance. Based on the cold temperatures achieved on those tests, it was decided to include hand warmers inside the system, but they were never tested at low pressure levels. It was concluded that they were not performing as expected and the system was possibly too cold to work at the expected altitude.

Figures 3.35 and 3.36 present examples of a payload and a controlled descent unit of this design stage.

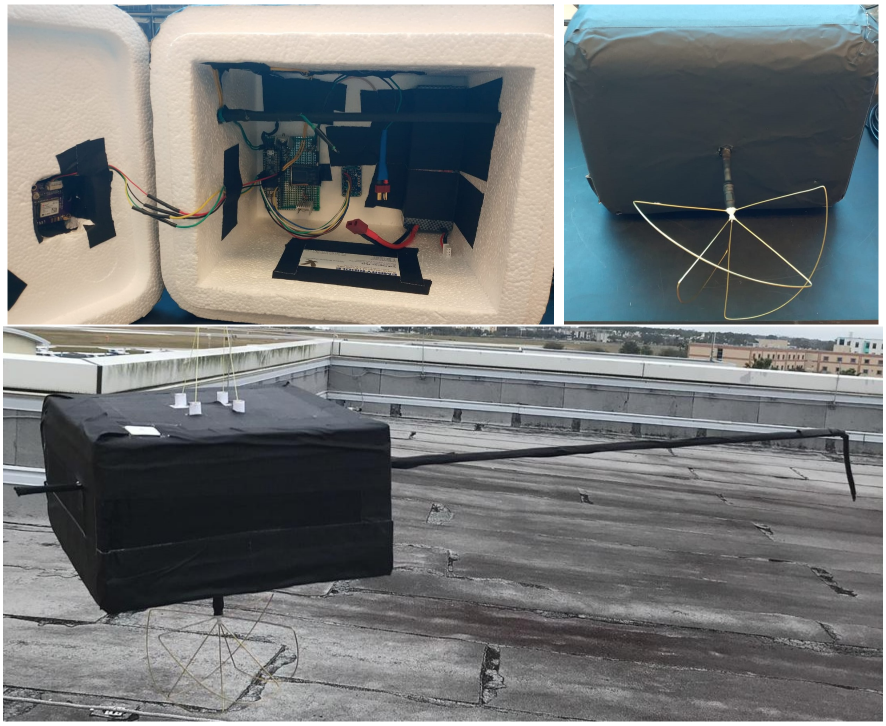


Figure 3.35: Design 3 - Final Payload Design Sample.

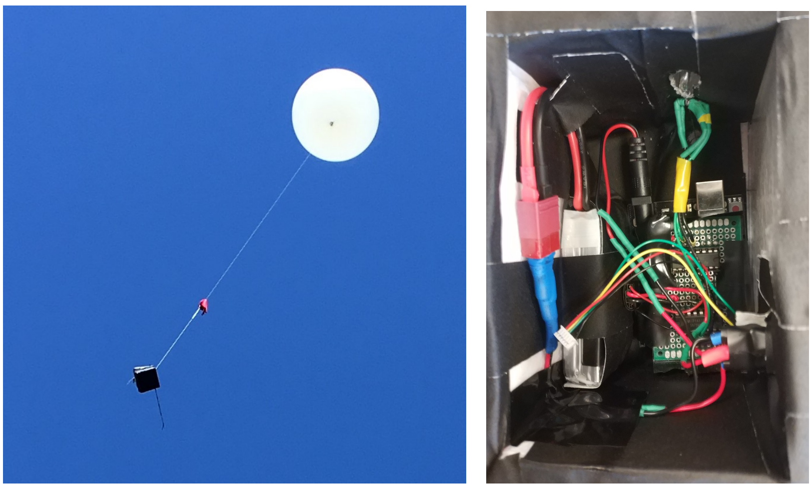


Figure 3.36: Design 3 - Final Controlled Descent Unit Sample.

3.8 Design Stage 4

3.8.1 Design Overview

The following items are the main updates and upgrades of this design stage.

- Adding temperature control system to the controlled descent unit - heating pads and temperature sensor.
- Designing and manufacturing printed circuit boards for the payload and the controlled descent unit designs.
- Payload code upgraded to be able to do data retransmissions out of the altitude range of interest.
- Multiple GS tracking (SIMO systems) being implemented and analysed.

Figures 3.37 presents the block diagrams of the payload and the controlled descent unit of this design stage.

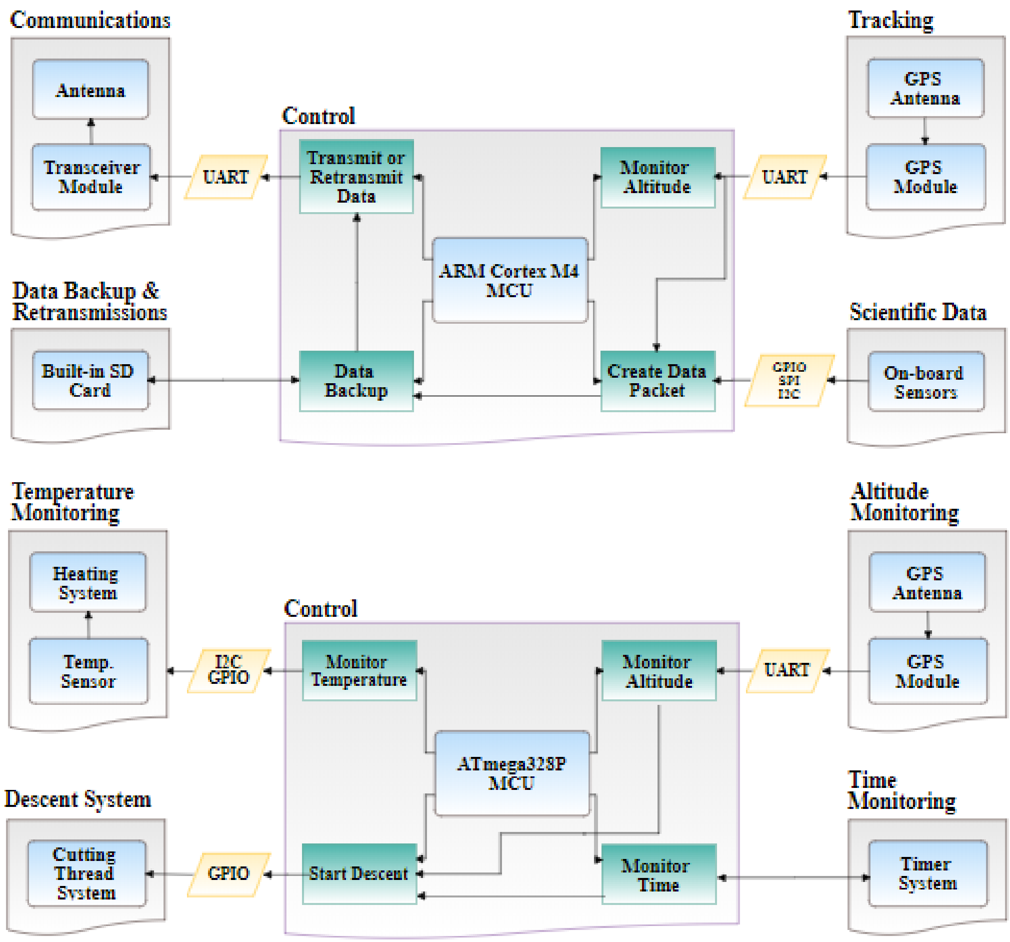


Figure 3.37: Design 4 Block Diagram - (A) Teensy 3.5 Payload with Retransmissions, (B) Independent/External Controlled Descent System with Heating Mechanism.

3.8.2 Payload Re-design

3.8.2.1 Printed Circuit Board

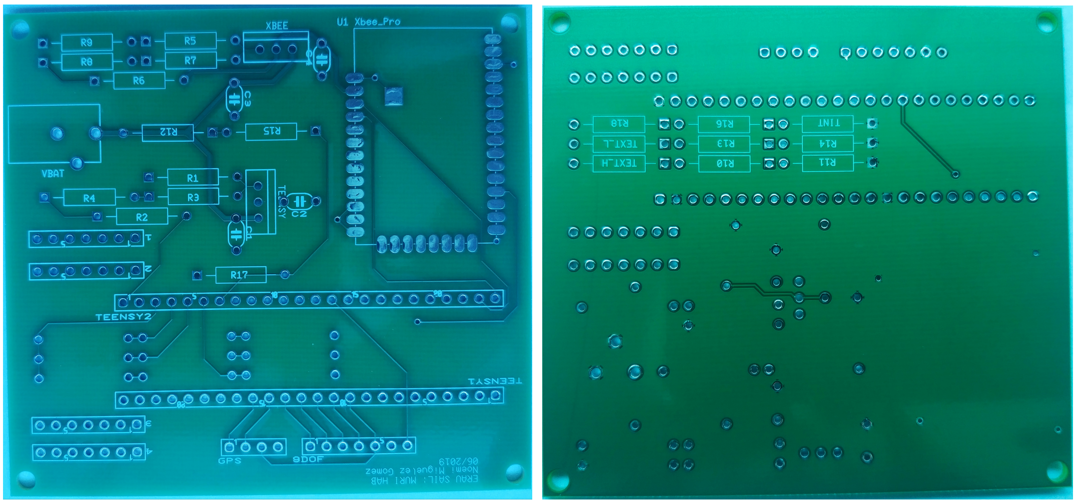


Figure 3.38: Design 4 - Payload Printed Circuit Board: (L) Top and (R) Bottom Face.

Considering that the payload design was closed in terms of microcontroller and transceiver selection, a printed circuit board was designed and manufactured to decrease the amount of time required to make a new payload, using National Instruments Multisim[44] and Ultiboard[45] programs.

The PCB includes two voltage regulator stages for the Teensy 3.5 (∼4.5V) and the XBee PRO SX transceiver (∼3.5V). It also includes the voltage dividers for three thermistors (one internal and two externals -for higher and lower external temperature ranges-) and the battery level monitor. It includes the transceiver footprint required to directly solder it on top of the board, as well as header pins to attach the microcontroller in the same board, where all the required components connections are made. Finally, the board includes pins to connect the wires coming from the GNSS receiver and the 9DoF sensor, which have specific positions in the payload and are kept outside of the board module.

As it can be seen in Figure 3.38, the width of the traces depends on the amount of current that they will need to supply to the respective board components -traces from the battery to the voltage regulators, and from the voltage regulator to the transceiver Vin-. The voltage regulators are LM317A [46], a three-terminal adjustable regulator model, and their recommended circuit design is followed, adding 0.1μF and 1μF capacitors to remove power line noise and to improve the transient response. Moreover, long traces and traces corners of 90° angles are avoided to reduce noise and avoid signal reflections, respectively. Finally, the bottom face of the PCB is a ground plane that simplifies the circuit layout allowing for grounding the components directly with a single via in the required ground connections. The final PCB design can be seen in Appendix F.

3.8.2.2 Data Retransmissions

In previous launches it could be seen how the data losses were increasing significantly after a certain range or if the payload path followed specific directions -due to environment interferences-. When doing long launches under wind conditions, the payload achieved slant ranges of more than 150 km before even starting the descent. In those conditions, even if the data is valid due to the slow descent rate, it can possibly not be enough to extract conclusions after being analysed due to the percentage of losses at that altitude/slant range. Considering that the altitude range of interest is between 20km and +35km, the payload code was updated to be able to detect that the system was descending, consider the next data packets until an altitude below 20km was reached, and start the retransmission of that part of the flight. The data being retransmitted does not consider past GNSS data, but new data coming from the GNSS sensors is transmitted during that time to keep tracking the payload. The data considered for the retransmissions is being sent infinitely until the end of the launch. More information about the code implementation and logic of this system can be found in Appendix H.

3.8.3 Controlled Descent - External Units with Heating System

3.8.3.1 Double Balloon Configuration - Cutting Thread System

Even though the malfunction of the last controlled descent unit was attributed to the cap/temperature system and not the electrical part of the design, it was concluded that a system to maintain the internal temperature of the box as hot as possible was required to confirm that the components temperature will not be affecting the system performance. A TMP102 [47] temperature sensor from Texas Instruments with an accuracy of 0.5°C (between –25°C to +85°C) to monitor the internal temperature of the box and a heating pad to be activated when that temperature is between 0 and 10°C were added to the system. The power budget of this system can be seen in Table 3.13.

Table 3.13: Controlled Descent Unit with Cutting System - Power Budget.

Component	**Current Consumption**	**Voltage Supply**	**%Use/Hour**
Microcrontroller	20	5	100
GNSS Receiver	20	3.3	100
Cutting System	1500	7.4	0.041
H-Bridges	25	5	100
Heating Pads	700	7.4	50
Temperature Sensor	0.085	3.3	100
Total Consumption/Hour	-	-	415.7 mA
Total Battery Capacity	-	-	1500 mAh
Total Capacity in Hours	-	-	3.61

Considering an efficiency of 80% due to minimum internal temperature improvements, the total capacity would be at least 2.9 hours.

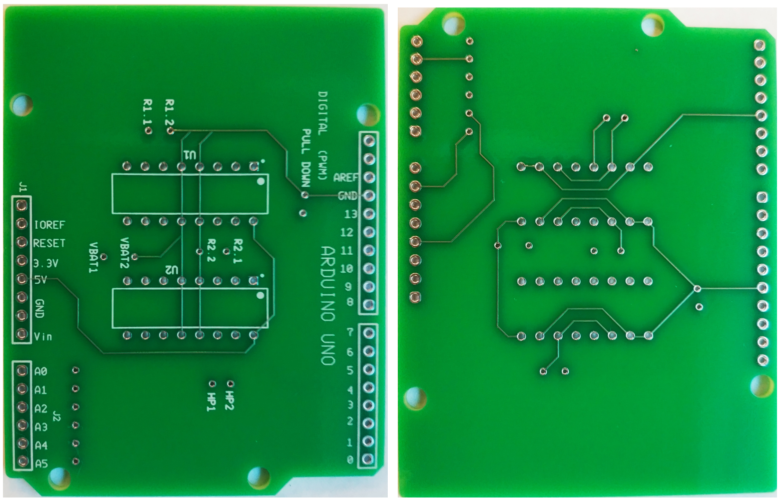


Figure 3.39: Design 4 - Controlled Descent System Printed Circuit Board: (L) Top and (R) Bottom Face.

In this design, once all the connections were confirmed, a PCB was produced as well to decrease the controlled descent unit production process. The same considerations taken into account for the payload's PCB were considered, excluding the ground plane, since some of the traces were routed in the bottom face of the board. The differences from the previous system were the connections for the temperature sensor and another output from one of the H-driver was routed to be used for the heating pads. Since only two out of four outputs of the H-bridge drivers were used for redundancy purposes, one of the non used ones were designated for the heating pads system. Figure 3.39 present this PCB design.

A commercially available styrofoam box was used for this design. A cutting-thread system was implemented to cut the line of the balloon that would fly away once the required altitude was achieved. As it can be seen, this system was a combination of the system seen in the first design stage and on the third one - independent/external system for a double-balloon launch configuration-. Figure 3.40 presents a 3D model of this system.

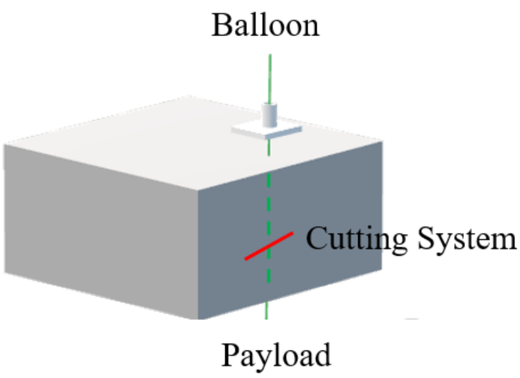


Figure 3.40: Design 3 - Cutting System 3D Model

The final PCB design can be seen in Appendix F.

3.8.3.2 Single Balloon Configuration - Valve System

In order to reduce the total cost per launch, a controlled descent unit based on the previous PCB design for single balloon configuration was implemented.

In this case, the unit design is similar to the cap system presented in the payload design stage 2, but instead of a cap it includes a 2-pieces 3D printed valve that can be open and/or closed by a micro servo motor. The servo motor arm has a thread connected to the valve and it is pre-programmed with two positions: to maintain the valve open and to close it. The cap of the valve is attached to a spring that creates tension to keep the valve open. This way, the motor position controls when the valve is completely open or completely closed. To avoid gas leaking problems, a grease is applied to the edges of the valve when is closed and prepared for the launch. This way, the aperture of the system is sealed. The grease was tested at cold temperatures in a temperature chamber to ensure that it was not frozen at the activation altitude.

For this design it was decided to include a communications link between the payload and the controlled descent unit. To do that, a HC-05 [48] Bluetooth module was selected to be able to command the valve system from the payload. These modules are configured to be paired with each other once they are powered and at a maximum range of about 25 meters from each other, automatically recovering the connection if the link is lost at any moment. These modules can be fully configured and paired via AT commands [49].

Figure 3.41 presents the Bluetooth module selected for this communication link, and its main parameters are presented in Table 3.14.

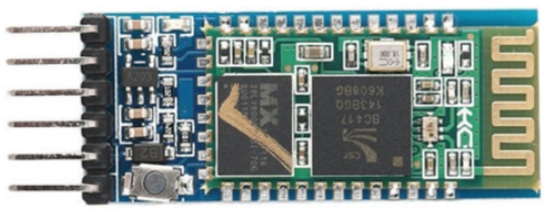


Figure 3.41: Controlled Descent Unit - Bluetooth HC-05 Module.

Table 3.14: Design 4 - Bluetooth Module specifications.

Parameter/Specification	**Value**
Operating Voltage	3.6 - 6V
Operating Current	30 mA
Antenna	PCB Trace
Power Output	$<$ 4 dBm
Range	$<$100 m
Interface	USART/TTL
Mode	Master, Slave
Max. Baud Rate	460k8 bps

The payload is programmed to send a series of 2-byte commands that the controlled descent unit interprets and answers with a specific action. The controlled descent unit is also sending back data about the code received and the internal temperature read by the temperature sensor used for the heating system. That data is included in the data packets that the payload sends to the GS, which allows the monitoring of the controlled descent unit with the GS GUI. The following commands are considered for this communication:

- **Open**: command used to open the valve indefinitely until a new position is commanded.
- **Close**: command used to close the valve indefinitely until a new position is commanded.
- **Open/Close**: command used to open and close the valve several times in a short period of time. Used to avoid possible problems if the grease is partially frozen.
- **Check**: command used to check the status/internal temperature of the system and that the communications link is still maintained.
- **Cut**: command used to activate a cutting-thread system to terminate the flight - cuts the balloon attached to the system, which descents under a parachute for the last 200-300 meters of altitude from the ground-.

The power budget for this system can approximately support a 3-hour launch, considering that it does not include a GNSS receiver, but the Bluetooth module consumes approximately the same amount of current. The added micro servo motor is only used a few times during the launch.

The payload sends check codes every 3-5 seconds to monitor the unit from the GS. Once a certain altitude is reached, the payload sends open/close commands for 1 minute. After that, it keeps sending open commands until a descent rate between 2.5 and 3.5 ms^{-1} is reached, and then close commands are sent indefinitely to terminate the controlled descent part of the launch. The payload can send a cut command when, during the descent part of the launch, the altitude is below 200-300 meters to be able to approximate the final location of the payload. More information about the software used for this system can be found in Appendix H.

3.8.4 Design Performance Results

- The payload design presented data losses due to the heating regulators shutting partially or completely due to payload heating dissipation problems. It was realized that if the payload was opened when the percentage of packet losses was increasing, the packet losses were suddenly solved. Bigger heating sinks were used and temperature chamber tests were performed to identify in which configuration the payload was only suffering packet losses for a short period of time, until the internal temperature was cooling.
- The retransmissions system worked successfully, decreasing considerably the total percentage of packet losses at from 35 to 20 km. In launches where

the payload presented the aforementioned heating problems, the total % of packet losses was reduced from almost 50% to 4% with only two retransmissions.

- The heating system of the controlled descent unit was tested in the temperature chamber and it worked successfully during several intervals of time at the expected temperature ranges. By the time the cutting system was activated, the threads were successfully burnt.

- The cutting system successfully worked in all the launches. However, in one of them the altitude was too low (23-24km) because the ascent rate was slower than expected and the system was activated by the timer at an altitude of approximately 23 km. On other cases, a premature balloon burst happened before the cutting system was activated. In that case, the 9DoF data was used to conclude that the system was activated during the descent with the other balloon attached to the payload.

- The controlled descent unit for a single balloon configuration was able to be commanded from the payload, based on altitude and descent rate. The GS GUI was able to receive data from the controlled descent unit and to present it for monitor purposes. This unit was not able to be tested during a launch.

Figures 3.42, 3.43 and 3.44 present examples of the payload and the controlled descent units for this design stage.

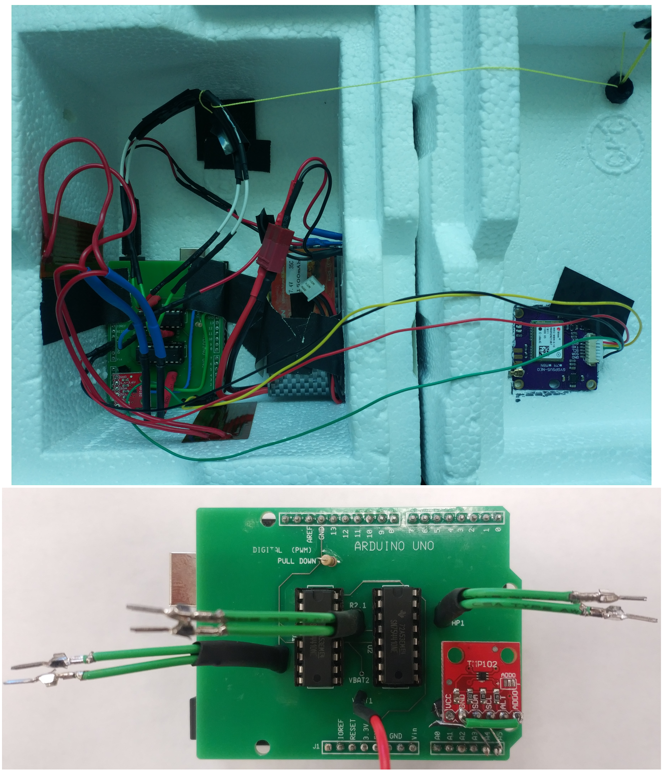


Figure 3.42: Design 4 - Double Balloon Controlled Descent Cutting-Thread System Sample.

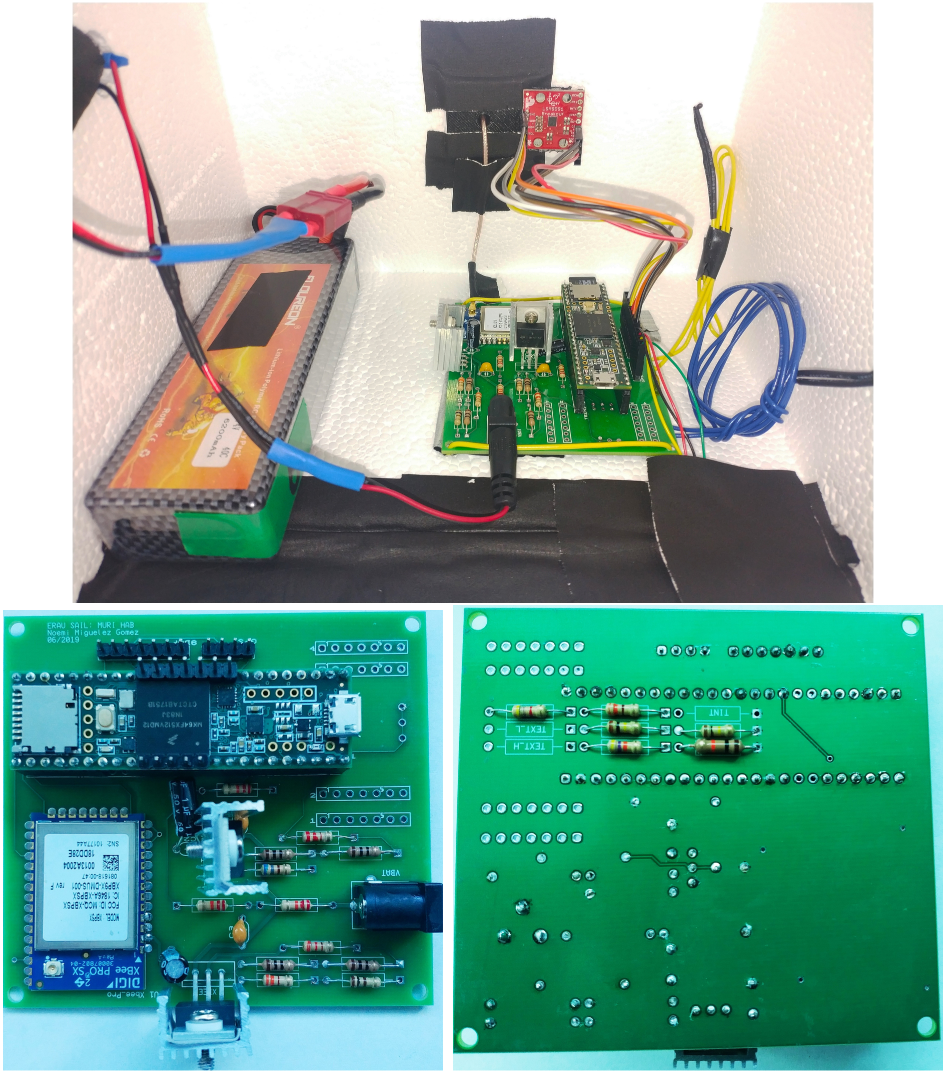


Figure 3.43: Design 4 - Final Payload Design Sample.

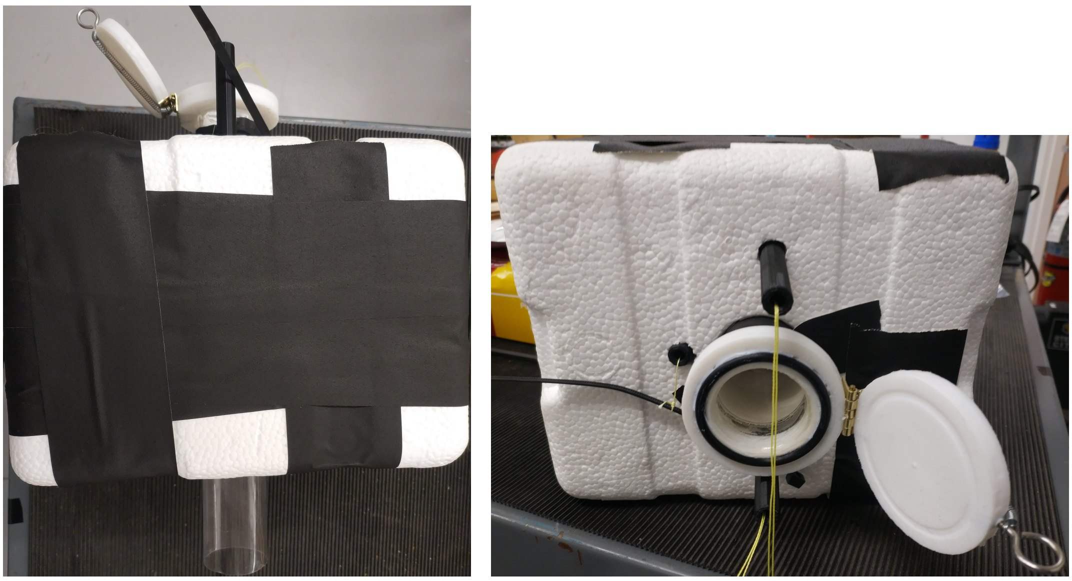


Figure 3.44: Design 4 - Single Balloon Controlled Descent Valve System Sample.

Chapter 4

Results and Analysis

This section present a set of results to analyse the payload and controlled descent unit performance. All the graphics in this section were generated with the data gathered from different launches, except from temperature chamber results. With them, the payload design specifications and requirements of the project will be presented and confirmed.

4.1 Throughput

4.1.1 Design Stage 1

In this stage, the DNT900 transceiver with an ATmega2560-based microcontroller was used. As it can be seen in Figure 4.1, in this flight the maximum slant range achieved was 178km, but only with a data throughput of 20 kbps. However, the data link was maintained with low percentage of packet losses until the slant range between the payload and the GS was higher than 130 km.

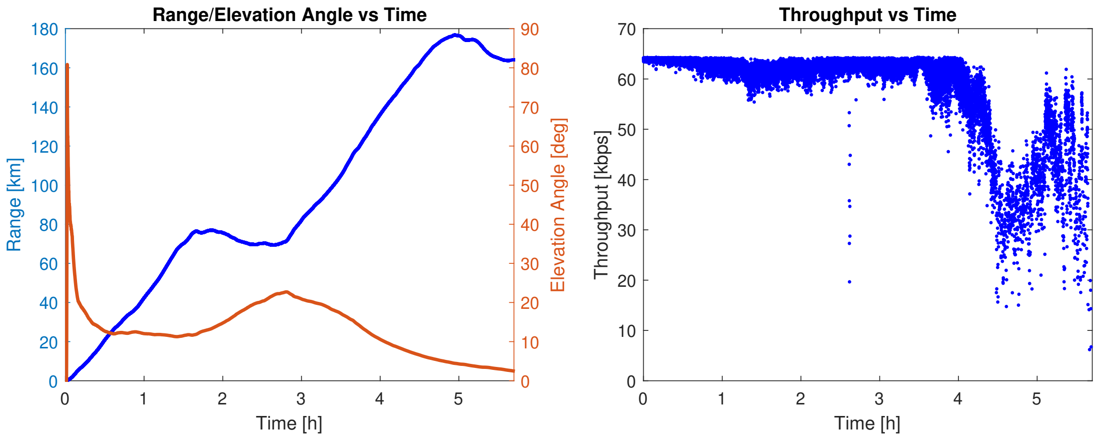


Figure 4.1: (L) Range and elevation data and (R) data throughput data for a 6-hour launch using the stage 1 of the payload design. Maximum slant range: 178 km, maximum data throughput: 65 kbps.

4.1.2 Design Stage 2

For the second stage of the payload design, the DNT900 transceiver was substituted by the XBee PRO SX one. Figure 4.2 presents an improvement in terms of data throughput, which was maintained at 80 kbps for the entire launch. However, during this 3-hour launch, the maximum slant range between the payload and the GS was only 40km.

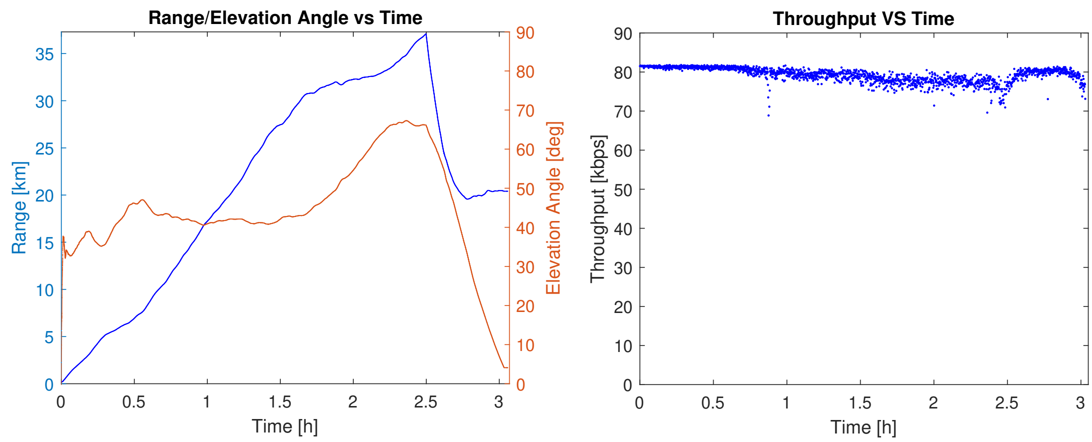


Figure 4.2: (L) Range and elevation data and (R) data throughput data for a 3-hour launch using the stage 2 of the payload design. Maximum slant range: 40 km, maximum data throughput: 82 kbps.

4.1.3 Design Stage 3-4

In the last two designs, Teensy 3.5 was the controller board of the payload. The higher clock speed of this board allowed an increment in the data throughput of the communications link.

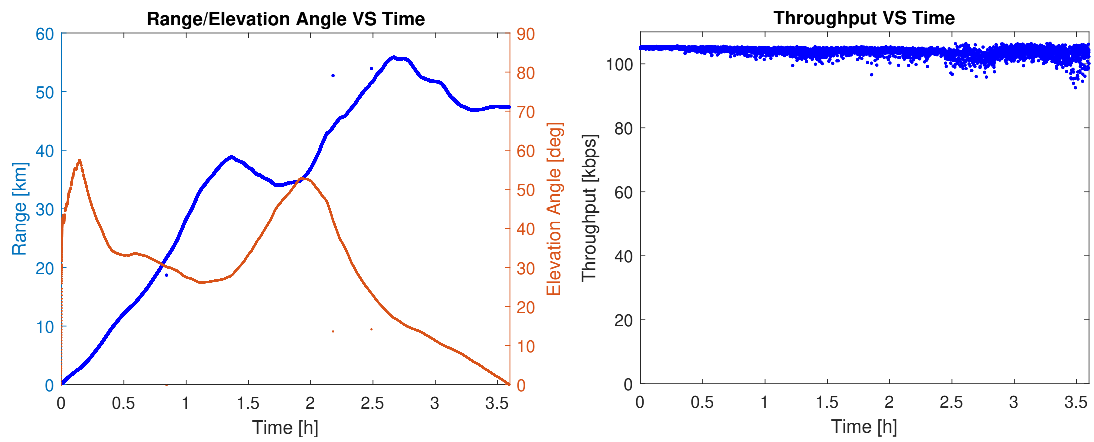


Figure 4.3: (L) Range and elevation data and (R) data throughput data for a 3.5-hour launch using the stage 4 of the payload design. Maximum slant range: 55 km, maximum data throughput: 105 kbps.

As it can be seen in Figure 4.3, a data throughput of approximately 105 kbps was maintained for the whole launch duration. It should be noted that for the last hour of the launch, the data was being retransmitted, which can result in a data throughput a bit higher considering a sequential code implementation and that the data packets are already created and saved in the SD card.

From Figure 4.4, it can be concluded that using retransmissions, a mean data throughput of at least 100 kbps can be maintained up to a slant range of approximately 150 km. For slant ranges below 100 km, a throughput between 105 and 110 kbps could be achieved.

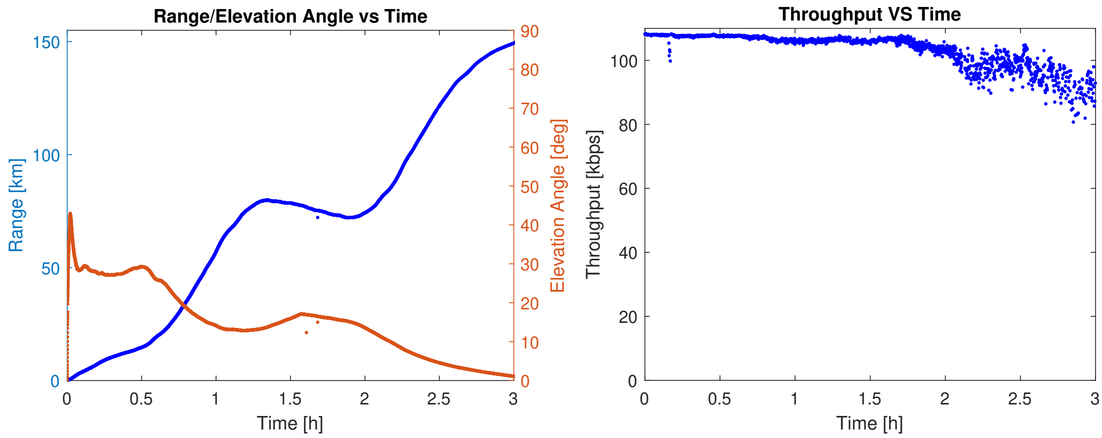


Figure 4.4: (L) Range and elevation data and (R) data throughput data for a 3-hour launch using the stage 4 of the payload design. Maximum slant range: 148 km, maximum data throughput: 108 kbps.

4.2 Measurements Resolution/Accuracy

In order to test the resolution of the measurements, the temperature sensors on board were used to analyse the data at the altitude range of interest. The figures below presents that altitude range, and the temperature data obtained with the first and the final payload designs.

For the highest achieved data throughput, and considering at least 3 sensor readings per packet, about 400 temperature measurements are taken per second. With a mean ascent and descent rates of approximately 3.5 m/s, that would result in a vertical resolution of 0.5 cm. Even though the GNSS receiver has a vertical resolution of 1 cm, the highest accuracy achieved is around 5 m, so it should be upgraded if that sub-cm scale precision needs to be achieved.

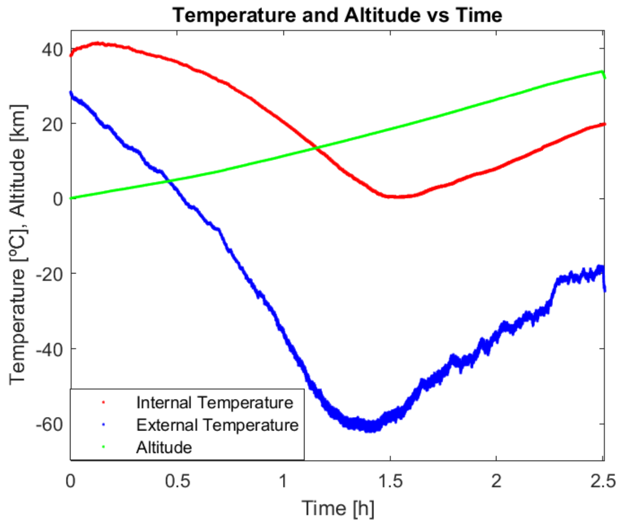


Figure 4.5: Harsh Internal Temperatures to which the components are subjected to (monitoring purposes) and the high resolution (0.1°C) for external temperature between 20 and 35 km (-60°C, -20°C) (scientific data analysis).

In Figure 4.5, it can be seen how the external temperature measurements are noisier for temperatures lower than approximately -45°C. In that case, only one external temperature sensor was being used to cover a big temperature range. Considering that, two sensors were used in next design iterations (stages 3 and 4), so one of them was covering an upper range of external temperatures and the other one was covering the lower range. Figure 4.6 presents the temperature data profile obtained by combining the data recorded from both sensors.

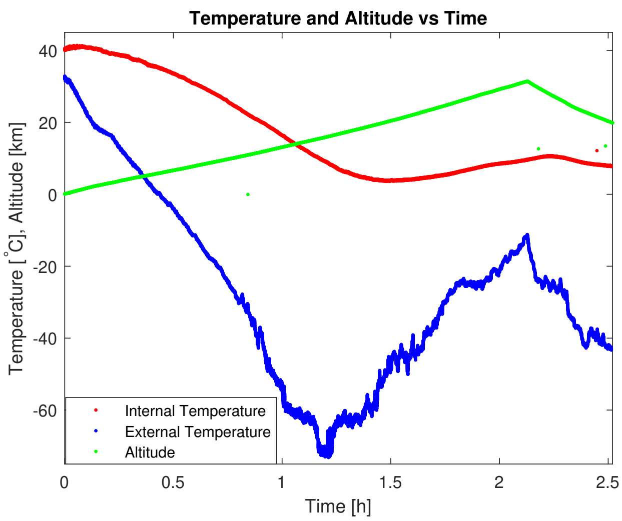


Figure 4.6: External temperature range extended until approximately -75°C with a minimum accuracy of approximately 0.2°C from -50°C to 30°C and 0.5°C -75°C to -50°C, respectively.

As it can be seen, the temperature range was extended until approximately -75°C with a considerable accuracy. These results were extracted from a HAB launch with retransmissions, where the maximum achieved altitude was approximately 31.5km, so the data between that altitude and 20 km was retransmitted during the rest of the launch. That is why the temperature plot only considers data until 20 km for the descent part of the launch.

While improving the accuracy of the external temperature measurements, it was realized that Teensy 3.5 boards have noisier ADC than ATmega2560-based boards, when tested together in the same temperature chamber profile. Therefore, it should be considered than even though the previous results present an improvement in external temperature data range in Figure 4.6, the accuracy is still not as good as it could be with the board used to obtain Figure 4.5 results.

4.3 Controlled Descent Unit

4.3.1 Heating System

In order to validate the heating system added to the controlled descent unit, two different temperature chamber tests were used. For these tests, temperature profiles based on data from the launches were considered.

Figure 4.7 presents the results from these tests. In one case, a heating system was not used and the internal temperature of the CDU box reached minimum temperatures close to -50°C. In the other case, a heating system activated when the internal temperature of the box was between -10 and 0 °C was used and the minimum internal temperature was approximately -15°C:

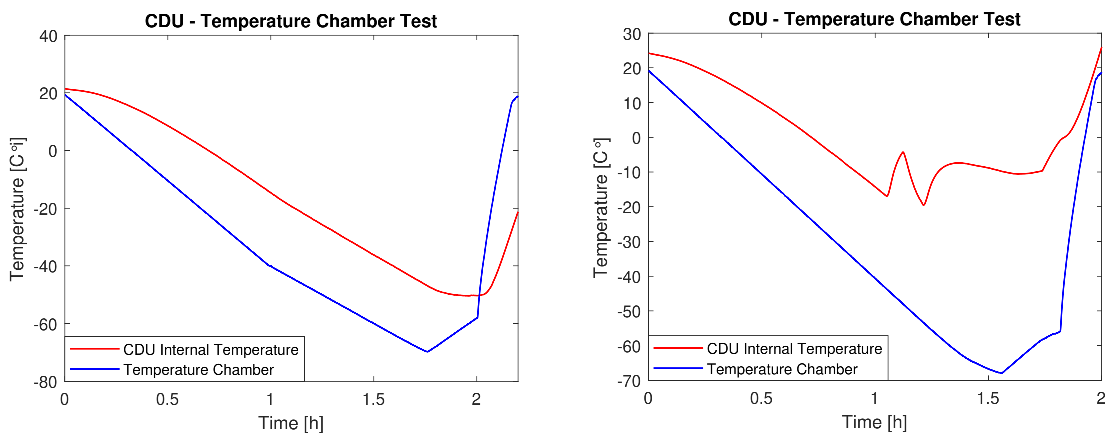


Figure 4.7: CDU temperature chamber tests results: (L) not using an internal heating system (R) activating a heating system when the internal temperature is between -10 and 0 °C.

The first time that the heating system is activated it is able to increase the internal temperature of the box until 0 °C and then it is deactivated. However, the second time the heating system is activated, the external temperature -temperature chamber- is too cold -about -60°C- for the heating pads to heat the box until 0 °C again and the system is permanently on until approximately the end of the test, when the temperature of the chamber is higher than -40°C again.

4.3.2 9DoF Data

The 9DoF sensor data was mainly used when understanding the experienced problems during the different launches as well as the actual flight movement profile that was followed in each case. The data allows to identify unexpected balloon bursts and cutting system entanglements, among other features.

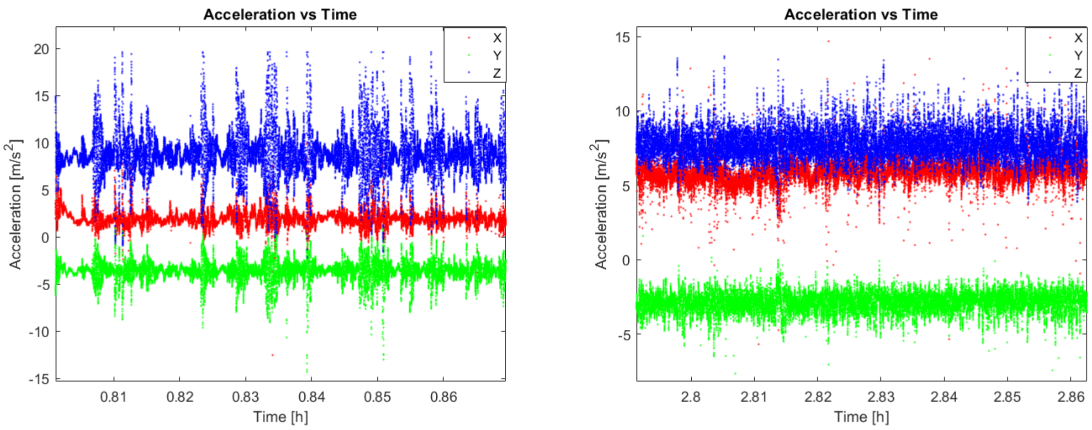


Figure 4.8: 9DoF Data during two different parts of the flight - Two different behaviors of the payload while flying can be distinguished: (L) semi-periodic spikes while two balloons are lifting the payload, (R) and a continuous acceleration while the payload is descending.

The data presented in Figure 4.8 allow for an analysis in case of communication losses or failure of the controlled descent system.

4.3.3 Descent Rates

Figures 4.9 and 4.10 are data plots from two different launches:

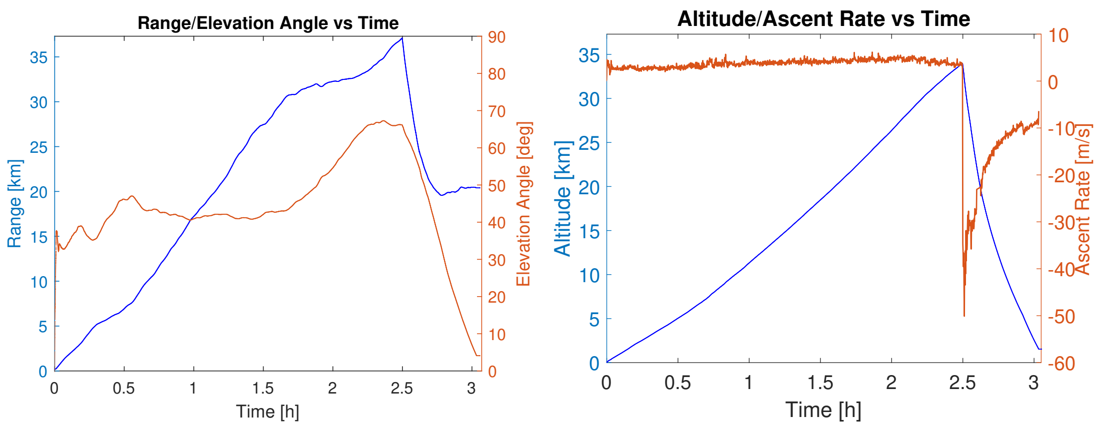


Figure 4.9: Fast Descent with Only a 1m Parachute Case: (L) range and elevation decreasing rapidly because the payload is descending at a fast rate, (R) descending approximately 34km in less than 30 minutes, with descent rates between 10 and 50 m/s.

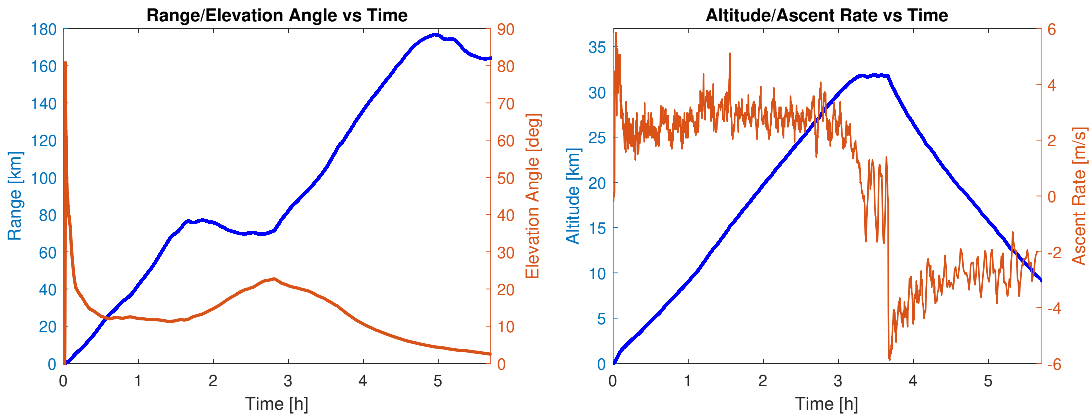


Figure 4.10: Slow Descent with One Balloon Case: (L) elevation decreasing slowly and slant range increasing progressively because the payload is descending at a slow rate and going away from the GS until the last hour of the launch, (R) descending approximately 31km in 2 hours, with descent rates between 2 and 6 m/s.

From the previous plots, it can be concluded that the controlled descent unit enables long duration launches and, therefore, higher resolution measurements during the descent part of the flight. For long launches, the slant range from the payload to the GS can be too much to maintain a high data throughput, but the flight path predictors can be used to choose the best time window for these type of launches, adjusting the ascent and descent rates, as well as considering the weather predictions.

Figure 4.11 presents the results from the controlled descent unit of the stage 4 of the design, in which one of the two balloons being used in that launch configuration was being cut either when a certain altitude or a specific launch time was reached.

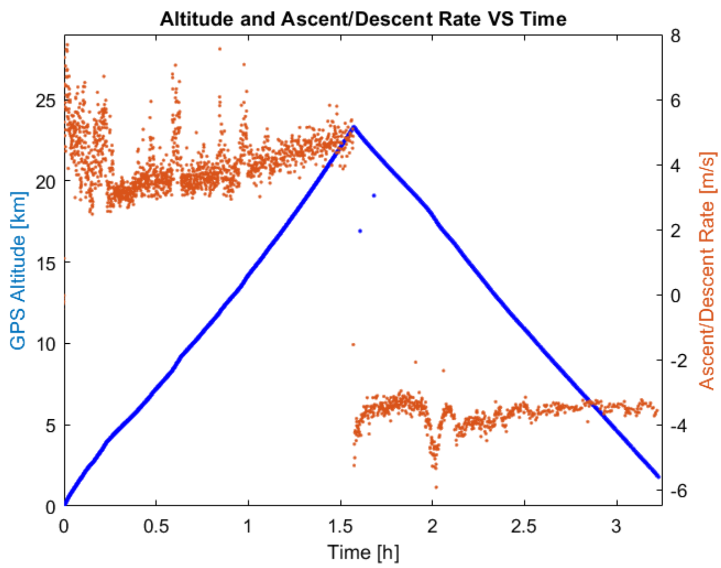


Figure 4.11: Cutting system activation at 23.5 km and slow descent at 3.5-4 m/s.

For the launch that provided the data from Figure 4.11, the controlled descent unit was configured with a timer of 1.5h and an altitude of 30km. Using the flight path prediction tool, it was determined that with an ascent rate of 5m/s, it would take 1h40min to the system to reach 30 km; due to that, the timer considered a few minutes less in case of overfilling the balloon and GNSS receiver failures. The ascent rate was lower than the one considered, resulting in a cutting system activation based on time at only 23.5 km of altitude.

Even though the controlled descent system was activated prematurely, it can be seen how the descent rate was instantaneously slow, between 3.5 and 4 m/s for the rest of the launch.

4.4 Data Retransmissions

As it can be seen in Figure 4.12, the retransmitted data and the new payload coordinates were being received at the same time. It has to be considered that the temperature data is plotted based on the payload altitude in the GUI; thus, it can be seen how the temperature values were periodically repeated, but the altitude is decreasing because the balloon is descending (see Ascent/Descent Rate label).

Using this plot, it can be seen in real-time if both the descent and retransmission modules worked by checking the data plots and the GNSS receiver data labels part of the GS GUI design.

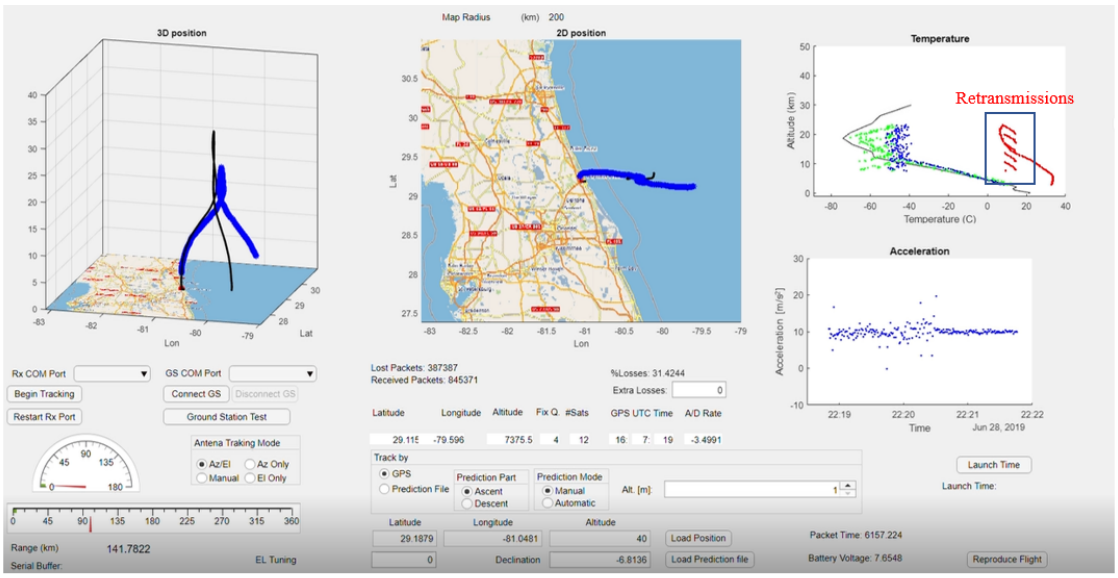


Figure 4.12: GS GUI - Data retransmissions during the descent.

Considering that SAIL owns two ground stations that can be used for HAB launches -a permanent and a mobile one-, two different cases were considered for retransmissions analysis: considering only one ground station tracking the payload (SISO system) and combining the data from two ground stations tracking the same payload (SIMO system). Both cases were analysed for a launch in which the payload was having heating problems. The communication link was completely lost when the voltage regulator supplying the transceiver was shutting down, resulting in several data packets being created with new data but not being sent.

Tables 4.1 and 4.2 present the improvement in total percentage of losses when combining the retransmitted data with the first transmission from 35 to 20 km.

- **Case 1 - 1 Ground Station (SISO System):**

Table 4.1: Retransmissions %Losses - Case 1

	Stage %Losses	Total %Losses
1^{st} Tx	49.95%	49.95%
1^{st} RT	51.14%	23.81%
2^{nd} RT	24.73%	5.89%
3^{rd} RT	19.34%	1.16%

In Table 4.1, with only one retransmission, more than 25% of the data was recovered, achieving less than 1.5% of total packet losses after 3 retransmissions.

- **Case 2 - 2 Ground Stations (SIMO System):**

Table 4.2: Retransmissions %Losses - Case 2

	Stage %Losses	Total %Losses
1^{st} Tx GS 1	49.95%	49.95%
1^{st} Tx GS 2	42.43%	46.23%
1^{st} RT GS 1	51.14%	21.73%
1^{st} RT GS 2	57.57%	17.27%
2^{nd} RT GS 1	24.73%	4.17%
2^{nd} RT GS 2	27.72%	2.52%

In Table 4.2, the data from two ground stations were merged and considered to analyze the data recovered during the retransmissions. As it can be seen, after 1 retransmission from each GS, about 28% of the data was recovered. With 2 retransmissions, only 2.5% of total packet losses was achieved.

This second case launch configuration was achieved by configuring the payload transceiver in broadcast mode with a certain network ID that was shared by the two ground stations as well.

For a MIMO configuration, with multiple payloads and ground stations, each payload is configured with the same network ID as the ground station tracking that system. However, in order to avoid data packets interferences, each pair of payload and ground station is configured in a different network ID.

Chapter 5

Budget and Resources

In this chapter, the project costs for both segments are detailed, as well as the facilities available for testing and calibration. The costs presented are the ones for the final ground station and payload hardware designs, the used software and the launch materials. A summary of all the costs is presented, indicating the average total cost of a single launch for the most expensive scenario -double launch configuration. Finally, the facilities available to develop, calibrate and test those designs is indicated.

5.1 Hardware and Software Cost

5.1.1 Ground Station

Table 5.1: Ground Station Cost Summary

Components[1]	**Quantity**	**Cost ($)**	**Total cost($)**
900MHz Yagi Antenna	1	88.95	88.95
Elegoo Mega2560	1	14.99	14.99
Yaesu G-5500	1	336	336
Tripod	1	34.99	34.99
Mast	1	25	25
Plates	3	10	30
Handles	3	3	9
PCB - Controller Shield	1	25	25
Long Power Cords	1	25	25
XBee PRO SX Board	1	100	100
Active USB Cable	1	28.19	28.19
Waterproof Box - Rotor	1	12.99	12.99
Waterproof Box - Transceiver	1	25.99	25.99
N Male - SMA Male Cable	1	15.85	15.85
SMA Male - RP SMA Male Cable	1	2	2
Tank Regulator	1	63.99	63.99
Tarp	1	6	6
Total	-	-	**843.94**

*[1] All the items can be considered one-time purchases.

5.1.2 Payload Costs

Table 5.2: Payload Cost Summary

Components	Quantity	Cost ($)	Total cost($)
Measurements System			
XBee PRO SX Chip	1	100	100
Teensy 3.5	1	24.95	24.95
Thermistors	3	4.5	13.5
Industrial 8GB SD Card	1	10.99	10.99
Cloverleaf Antenna	1	1.50	1,50
uBlox NEO-M8N	1	26.99	26.99
LSM9DS1 9DoF	1	15.95	15.95
PCB	1	20	20
Styrofoam Box	1	10	10
Floureon 7400mAh Battery	1	30	30
Waterproof Switch	1	2	2
Heatsink	2	2	4
Cables and Misc.	1	5	5
Subtotal 1	-	-	**264.88**
Controlled Descent System			
Elegoo UNO	1	11.86	11.86
SN754410NE Driver	2	2.5	5
ublox NEO-M8N	1	26.99	26.99
Temperature Sensor	1	5	5
Heating Pad	2	2.5	5
PCB	1	10	10
Floureon 1500mAh Battery	1	11	11
Waterproof Switch	1	2	2
Styrofoam Box	1	5	5
Cables and Misc.	1	3	3
Subtotal 2	-	-	**84.85**
Total	-	-	**349.73**

5.1.3 Launch Setup Costs

Table 5.3: Launch Setup Cost Summary

Components	Quantity	Cost ($)	Total cost($)
1500gr Balloon	1	150	150
1000gr Balloon	1	100	100
1m parachute	-	35	35
Helium Tank (200ft^3)	0.75	220	165
Total	-	-	**450**

5.1.4 Software Costs

All the software used in this project is available for ERAU students at no cost or they are open source programs. Therefore, the are no software cost related to this part of the MURI project. The programs used are presented in the following list:

- MATLAB + App Designer Package.
- Arduino IDE.
- Digi XCTU.
- u-center (uBlox Software Center).
- National Instruments Multisim/Ultiboard.

5.2 Summary

- Ground Station cost: **$843.94** (one time purchase).
- Payload costs: **$349.73**
- Launch Setup costs: **$450**
- Software cost: **$0**

Once the design is finished and a ground station is completely prepared, each balloon launch results in the sum of the payload cost and the launch setup cost, which would be approximately **$800** per launch.

5.3 Facilities

- **Space and Atmospheric Instrumentation Laboratory (SAIL)**

  SAIL is part of the Center for Space and Atmospheric Research (CSAR) and it is located within the Physical Sciences Department in the College of Arts and Sciences building. This laboratory includes mechanical hardware building capabilities, desks and workstations and an ESD safe zone for SMT assembly and testing that will be useful for this project development.

- **Plasma Lab**

  This laboratory is where the temperature and pressure calibrations are performed. It also includes machinery required for the ground station development, such as a hydraulic press for the base plates holes.

Chapter 6

Conclusions

This thesis premise was the implementation of a communications bus and tracking and ascent/descent controlled systems for a high-altitude balloon platform. It centers mostly on the payload and the controlled descent unit design, implementation and configuration, as well as the concepts and techniques used to achieved the project requirements.

Taking this into account, in addition of the preliminary goals and the final results of this thesis, these are the conclusions that can be established:

- Altitudes higher than 30 km were achieved, with a controlled descent unit able to slow the descent rates and to obtain valid data even during that leg of the flight.
- A throughput higher than 100 kbps has been validated to be working until slant ranges of approximately 150 km, which allows centimeter scale turbulence measurements when combined with slow descent rates.
- A logic including retransmissions of data of interest below 20 km was validated and it allows to receive almost 100% of the scientific data gathered in the regions of interest.
- The ground station system was validated to be working and easy to be duplicated by our and other universities. This design was shared with the other universities participating in this project, which they used in a permanent position or on top of a mobile vehicle to follow the balloon during the launch.
- Multi-point launches were accomplished thanks to the transceiver configuration. SISO and SIMO configurations were tested and proved to be working, where one payload was tracked by one or two ground stations at different locations. A SIMO system can be useful for long launches, when deploying the second one for being used as a relay system.
- The mass-production of this design is possible thanks to the printed circuit boards produced, which speed up the production and testing process, and its affordable cost of 800$ per double-balloon launch.

Chapter 7

Future Development

As a future approach, the integration of this design with the other universities systems, the final tests of the new controlled descent design, as well as the redesign of the payload PCB to add the other universities sensors and to solve the observed heating problems are considered:

- **Payload PCB Re-design.**

  In order to avoid future communication losses due to heating problems, the payload PCB can be re-designed to increase the distance between the voltage regulators and the transceiver module. Since it has been proven in past launches that this design can work, it can be a possible solution to ensure that the shut down problems will not appear if a styrofoam box with not enough ventilation is used.

  If a double stage voltage regulation is considered, the design could include an intermediate state going from 7.4V to 5V and then from 5V to 3.3V. Doing this, the power dissipated in the voltage regulator will be decrease and it will help with the heating problems.

  Finally, a specific heatsink design to cool the voltage regulator used could be proposed. Using the low external temperatures, the heatsink could be helping dissipating the extremely high temperatures that the chip achieves.

- **Systems Integration.**

  Once the system covered by this thesis is fully tested and proved to be perfectly working, it will have to be integrated in the final HAB design, where the other universities that are involved in this project will be adding the required sensors to be launched during the scientific campaign to collect data of AFOSR interest. The collected data during those launches will allow the analysis and model characterization of stratospheric turbulence that will be considered for the hypersonic vehicles design.

- **Controlled Descent Unit for a Single-Balloon Configuration.**

  A double-balloon configuration system has been proved to be successfully working for a controlled descent system. However, the design for a single balloon configuration based on a valve system commanded from the payload using a Bluetooth communications link was not able to be tested during a HAB launch.

  When writing this document, this system is waiting to be launched to analyse its performance, which will finally confirm that the communications link can be maintained for the whole launch duration and that, therefore, the controlled descent unit can be commanded from the payload.

- **Ground Station Upgrade.**

  The ERAU permanent ground station can be upgraded to improve the communications link results. RF signal conditioning can be added to reduce the received noise and improved the signal-to-noise ratio, such as a low noise amplifier. Moreover, some signal filtering can be added to avoid interferences. Doing this, the percentage of packet losses or errors over a certain slant range can be improved.

  If required, uplink capabilities can be added to the ground station design. To do that, the band regulations shall be taken into account, since the maximum EIRP allowed is 4W. One possible solution is to configure the GS transceiver with a maximum output of 19-20 dBm, or even add a secondary RF chain for the uplink considering the required signal attenuation.

- **Academic Purpose.**

  Although this design will be modified to be able to include the sensors required for the experiments that will be conducted for the AFOSR, it could also be modified to include a different scientific purpose. As long as the microcontroller has pins available, a series of different sensors could be added. It will only require to change the payload packet format/communication with the sensor or system, and to use the same format in the ground station GUI to be able to successfully monitor the whole system.

Bibliography

[1] National Weather Service. *National Weather Service - Radiosonde Observations*. URL: https://www.weather.gov/gjt/education_corner_balloon.

[2] Vaisala. *Vaisala Radiosonde RS92-SGP*. URL: https://www.vaisala.com/sites/default/files/documents/RS92SGP-Datasheet-B210358EN-F-LOW.pdf.

[3] Steven Levy. "How Google Will Use High-Flying Balloons to Deliver Internet to the Hinterlands." In: (June 2013).

[4] WIRED. "Google Laser-Beams the Film Real Genius 60 Miles Between Balloons". In: (February 2016).

[5] Zero 2 Infinity. *Zero 2 Infinity - Simplifying Access to Space*. URL: http://www.zero2infinity.space/.

[6] Zero 2 Infinity. *Bloon - The Near Space Experience of a Lifetime*. URL: http://www.zero2infinity.space/bloon.

[7] Stratodynamics. *HiDRON - Purpose Built to the Stratosphere*. URL: http://www.stratodynamics.ca/about/.

[8] High Altitude Science. *High Altitude Science Website*. URL: https://www.highaltitudescience.com/.

[9] Idoodlelearning. *Cubes in Space - Ingenuity Taking Flight*. URL: http://www.cubesinspace.com/.

[10] National Aeronautics and Space Administration. *The High Altitude Student Platform (HASP)*. URL: https://sites.wff.nasa.gov/code820/docs/outreach/High%5C%20Altitude%5C%20Student%5C%20Platform.pdf.

[11] AFOSR. *AFOSR - Research Areas - Integrated Measurement and Modeling Characterization of Stratospheric Turbulence*. URL: https://community.apan.org/wg/afosr/w/researchareas/22954/integrated-measurement-and-modeling-characterization-of-stratospheric-turbulence/.

[12] Electronic Code of Federal Regulations. *FAA - Part 101—MOORED BALLOONS, KITES, AMATEUR ROCKETS, UNMANNED FREE BALLOONS, AND CERTAIN MODEL AIRCRAFT*. URL: https://www.ecfr.gov/cgi-bin/text-idx?rgn=div5&node=14:2.0.1.3.15.

[13] Electronic Code of Federal Regulations. *FCC - §22.925-Prohibition on airborne operation of cellular telephones*. URL: https://www.ecfr.gov/cgi-bin/text-idx?rgn=div8&node=47:2.0.1.1.2.8.27.12.

[14] A. Krauchi. "Controlled Weather Balloon Ascents and Descents for Atmospheric Research and Climate Monitoring". In: *EGU - Atmospheric Measurements Techniques* (March 2016).

[15] D. Vignelles. "Caractérisation des performances du nouveau minicompteur de particules LOAC embarqué sous ballon météorologique : application à l'étude de la variabilité spatiale et temporelle des aérosols de la haute troposphère et de la stratosphère". In: *Laboratoire de Physique et de Chimie de l'Environnement et de l'Espace LPC2E* (January 2016).

[16] High Altitude Science. *StratoTrack APRS Transmitter*. URL: https://www.highaltitudescience.com/products/stratotrack-aprs-transmitter.

[17] A. Shagger and N. Amilia. "Mission Design and Analysis of USM High-Altitude Balloon". In: *Journal of Mechanical Engineering* (2017).

[18] H. Kimm et al. "Real-Time Data Communication Using High Altitude Balloon Based on CubeSat Payload". In: *Journal of Advances in Computer Networks, Vol. 2, No. 3* (September 2015).

[19] SparkFun. *HAB - Sensor System, Flight Computer, and Radio System*. URL: https://www.sparkfun.com/tutorials/185.

[20] A. Mohammend and Z. Yang. "Broadband Communications and Applications from High Altitude Platforms". In: *ACEEE International Journal on Communication, Vol. 1,No. 1* (January 2010).

[21] LTD Yaesu Musen Co. "Yaesu G-5500 Antenna Azimuth-Elevation Rotators Controller Instruction Manual." In: ().

[22] Wikipedia. *Coordinating Committee for Multilateral Export Controls*. URL: https://en.wikipedia.org/wiki/Coordinating_Committee_for_Multilateral_Export_Controls#Legacy.

[23] Elegoo. *Mega2560 R3 Board*. URL: https://www.elegoo.com/product/elegoo-mega-2560-r3-board-atmega2560-atmega16u2-usb-cable/.

[24] uBlox. *NEO-M8 - u-blox M8 Concurrent GNSS Modules Data Sheet*. URL: https://www.u-blox.com/sites/default/files/NEO-M8-FW3_DataSheet_%5C%28UBX-15031086%5C%29.pdf.

[25] Trimble. *Copernicus II GPS Receiver*. URL: http://trl.trimble.com/docushare/dsweb/Get/Document-481581/63530-10_Rev-B_Manual_Copernicus-II_2009-09-14.pdf.

[26] muRata. *DNT900 - Low Cost 900 MHz FHSS Transceiver Module with I/O*. URL: https://wireless.murata.com/pub/RFM/data/dnt900c.pdf.

[27] Kingston Technology. *Kingston - Ideal for industrial applications and extreme conditions*. URL: https://www.kingston.com/us/memory-cards/industrial-temperature-microsd-uhs-i.

[28] Littlefuse. *PR Series Ultra Precision Interchangeable Thermistors*. URL: https://www.littelfuse.com/~/media/electronics/datasheets/leaded_thermistors/littelfuse_leaded_thermistors_interchangeable_thermistors_ultra_precision_pr_datasheet.pdf.pdf.

[29] SparkFun. *SparkFun 9DoF IMU Breakout - LSM9DS1t*. URL: https://www.sparkfun.com/products/13284.

[30] Honeywell. *ASDX Series Silicon - Pressure Sensors*. URL: https://sensing.honeywell.com/index.php/ci_id/45330/la_id/1/document/1/re_id/0.

[31] Texas Instruments. *SN754410 Quadruple Half-H Driver*. URL: http://www.ti.com/lit/ds/symlink/sn754410.pdf.

[32] Habhub. *Balloon Burst Calculator*. URL: http://habhub.org/calc/.

[33] Floureon. *Floureon - RC Batteries*. URL: http://www.floureon.com/rc-battery-e_16/.

[34] Digi. *XBee®/XBee-PRO SX - Radio Frequency (RF) Module*. URL: https://www.digi.com/resources/documentation/digidocs/pdfs/90001477.pdf.

[35] Digi. *XCTU - Configuration Platform for XBee/RF Solutions*. URL: https://www.digi.com/products/embedded-systems/digi-xbee-tools/xctu.

[36] SparkFun. *Teensy 3.5*. URL: https://www.sparkfun.com/products/14055.

[37] NXP. *K64 Sub-Family Reference Manual*. URL: https://cdn.sparkfun.com/datasheets/Dev/Arduino/Boards/K64P144M120SF5RM.pdf.

[38] Hobby King. *900Mhz Circular Polarized Antenna Set*. URL: https://hobbyking.com/en_us/900mhz-circular-polarized-antenna-set-sma-lhcp-short.html.

[39] HAM TV - Tom O'Hara. *ATV in Rockets and Balloons*. URL: http://www.hamtv.com/rocket.html.

[40] Antenna Theory. *Cloverleaf Antenna*. URL: http://www.antenna-theory.com/antennas/cloverleaf.php.

[41] Radio Antenna Engineering. *Radiation Pattern*. URL: http://www.vias.org/radioanteng/radio_antenna_engineering_03_06_02.html.

[42] Ham Radio. *Ground Effects*. URL: https://www.hamradio.me/?s=ground+plane.

[43] Elegoo. *R3 Board ATmega328P*. URL: https://www.elegoo.com/product/elegoo-uno-r3-board-atmega328p-atmega16u2-with-usb-cable/.

[44] National Instruments. *NI - Multisim*. URL: http://www.ni.com/sv-se/shop/select/multisim?skuId=58527.

[45] National Instruments. *NI - Ultiboard*. URL: https://www.ni.com/sv-se/shop/select/ultiboard.

[46] Texas Instruments. *LM317A 1% Accurate Three-Terminal Adjustable Regulator*. URL: http://www.ti.com/lit/ds/symlink/lm317a.pdf.

[47] Texas Instruments. *TMP102 - Low Power Digital Temperature Sensor With SMBus™/Two-Wire Serial Interface in SOT563*. URL: https://www.sparkfun.com/datasheets/Sensors/Temperature/tmp102.pdf.

[48] ITEAD. *Serial Port Bluetooth Module: HC-05*. URL: https://www.itead.cc/wiki/Serial_Port_Bluetooth_Module_(Master/Slave):HC-05.

[49] Instructables. *HC-05 Embedded Bluetooth Serial Communication Module-AT Command Set*. URL: https://cdn.instructables.com/ORIG/FHJ/PL61/IRXT0HXV/FHJPL61IRXT0HXV.pdf.

Appendix A

Ground Station Design

The ERAU ground station consists of 5 modules: antenna module, rotor, mast, tripod, base plates.

A.1 Base Plates

A.1.1 Materials List

For the base plates, the following list of materials was used:

- 12' x 12' x $\frac{1}{2}$' Steel Plate.
- 8-32 x 1 Flat Phillips Machine Screw.
- 10-24 x 1-$\frac{1}{2}$ Flat Phillips Machine Screw.
- 4 3/4 in. Screen Door Pull Handle.
- 8-32 Nylon Insert Lock Nuts.
- 8 Flat Washers.
- 10-24 Wing Nut.
- 10 Lock Washers.
- $\frac{1}{2}$ in. 82 Metal Countersink Drill Bit.

Those materials were all obtained via standard home improvement store locations and are all commercially available products. They are all required to complete the base assembly outlined in the next subsection. Any changes made to this assembly process may alter the above material list.

A.1.2 Assembly and Disassembly

Step 1. With all materials listed in Materials section obtained, the three 12in x 12in x $\frac{1}{2}$in steel base plates must be properly drilled. This includes seven 82° countersink holes at the locations specified in Figure A.1.

It is recommended that the drilling be done with a machine drill to maintain dimensional accuracy and due to the duration, it may take to drill the steel plates. Once each plate was drilled with a machine drill, the in. 82° countersink drill bit was then used on each drill hole. With the completion of this, the steel plates are ready for further assembly.

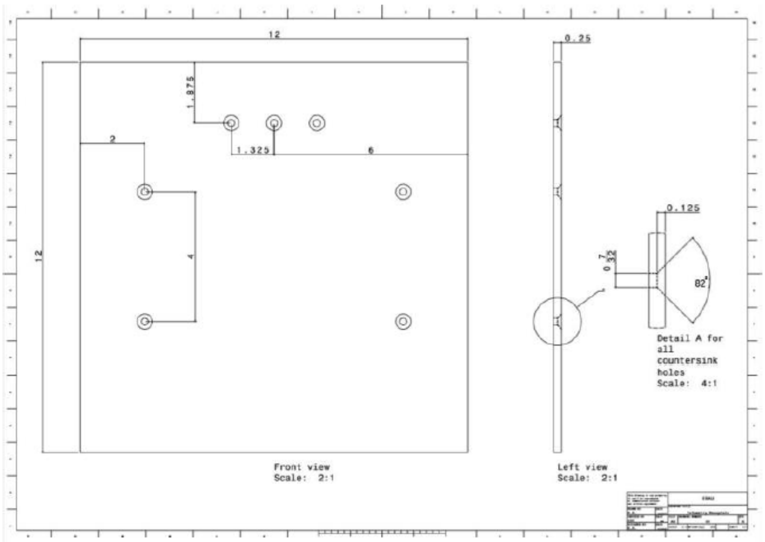


Figure A.1: Design template for steel base plate drilling configuration. Each drilled hole is identical in diameter and countersink. Note that the two holes on either side are symmetrical and are constrained to the same dimensions, not displayed.

Step 2. With the countersink portion of the steel plate facing down, feed four 8-32 x 1 screws through the side holes as displayed below.

Step 3. Place a 4 $\frac{3}{4}$ in. door pull handle over each of the two screw pairs. Over the handle, slide a 8 washer onto each screw, followed by an 8-32 nylon insert lock nut for each. Tighten the lock nut with a wrench until all parts are securely fastened. The completed configuration is shown in the following depiction.

Figure A.2: Completed view of Assembly Steps 2 and 3.

Step 4. Feed three 10-24 x 1-$\frac{1}{2}$ screws facing up through the top three holes of each base plate. Note - these screws should line up nearly flush with the bottom, as they are a larger screw size than the 8-32 screws.

Step 5. Place the three open sockets of the L-bracket of an antenna leg over the three open screws. Make sure that the base plate is facing away from the structure to allow for ease of use following assembly. On each screw, place a 10 lock washer, and secure the base plate with a 10-24 wing nut for each screw. Tighten each wing nut until the screw is secured and all parts are flush to each other. The configuration should look as follows.

Figure A.3: Completed assembly of steps 4 and 5, with base plate attached to the leg of the antenna tripod.

Step 6. If not done so already, repeat Steps 1-5 for each of the remaining base plate assemblies. Make sure to have each base plate facing outward of the antenna so that it is more accessible for weights or for disassembly.

The disassembly process does not contain as many steps as assembly. Simply undo the wing nuts from each leg and pull the L-bracket up from the screw configuration. Make sure to house the wing nuts and screws in a secure location for repeated use and assembly.

A.2 Tripod

The tripod considered in this design is a 3 feet commercially available tripod. A pack of this tripod with a 2-inch OD mast is commercially available.

The tripod needs to be completely extended for maximum stability, as well as to be secured with the base plates.

Adding enough weights on the base plates, the ground station demonstrated to handle winds up to 40 mph.

Figure A.4: ERAU Ground Station Tripod

A.3 Mast

The mast considered in this design is a 2-inch OD mast.

Figure A.5: ERAU Ground Station Mast

Two different sizes are used for indoor tests and actual launch setups. Both are commercially available products in home improvement store locations, and their price is about $4/ft.

The mast needs to be completely secured by the tripod screws. After the ground station is north aligned, the tripod shall be able to avoid the mast movements. If not, the antenna pointing offset can be a problem during the launch.

A.4 Rotor

The Yaesu G-5500 is a rotor system that has both azimuth and elevation controls. The azimuth of the rotor has a turning range of 0°- 450°. The elevation of the rotor has a rotation range of 0° - 180°. Table A.1 presents the main specifications of the selected rotor.

Table A.1: Yaesu G-5500 Rotor Specifications.

Voltage Requirement	110-120 or 200-240 VAC
Motor Voltage	24 VAC
Rotation Time (@60Hz)	Elevation(180°) : 67$secs$
Maximum Continuous Operation	5 minutes
Rotation Torque	Elevation: 14 kg-m (101ft-lbs) Azimuth: 6 kg-m (44 ft-lbs)
Braking Torque	Elevation: 40 kg-m (289 ft-lbs) Azimuth: 40 kg-m (289 ft-lbs)
Vertical Load	200 kg (440 lbs)
Pointing accuracy	±4 percent
Wind Surface Area	1 m^2
Control Cables	2x6 conductors = 20 AWG or larger
Mast Diameter	38-63mm (1-1/2 to 2-1/2 inches)
Boom Diameter	32-43 mm(1-1/4 to 1-5/8 inches)
Weight	Rotators: 9 kg (20 lbs) Controller: 3 kg (6.6lbs)

Figure A.6: ERAU Ground Station Rotor

A.5 Control

A.5.1 Components - Parts

The list of components used for the electronics design of the ground station control part is presented below:

- Arduino Mega2560
- uBlox NEO M8N GNSS Sensor
- 7 conductor cable (Digikey part # T1348-5-ND)

- Female crimp pins (Digikey part # A25969CT-ND)
- TE 10 pin connector housing (Digikey part # A25901-ND)
- 4 NPN BC337 transistors (Digikey part # BC33740TACT-ND)
- 2 channel LMC6483 OPerational Amplifier (Digikey part # LMC6482IN/NOPB-ND)
- 5 qty (1x8) header pins (Digikey part # 609-3301-ND)
- 1 qty (2x16) header pins (Digikey part # 732-5309-ND)
- 4 qty 10K resistors
- TE Male 10 pin connector (Digikey part # A33179-ND)
- 5 pin Molex Picoblade connector (Mouser part # 538-53048-0510)
- PCB shield

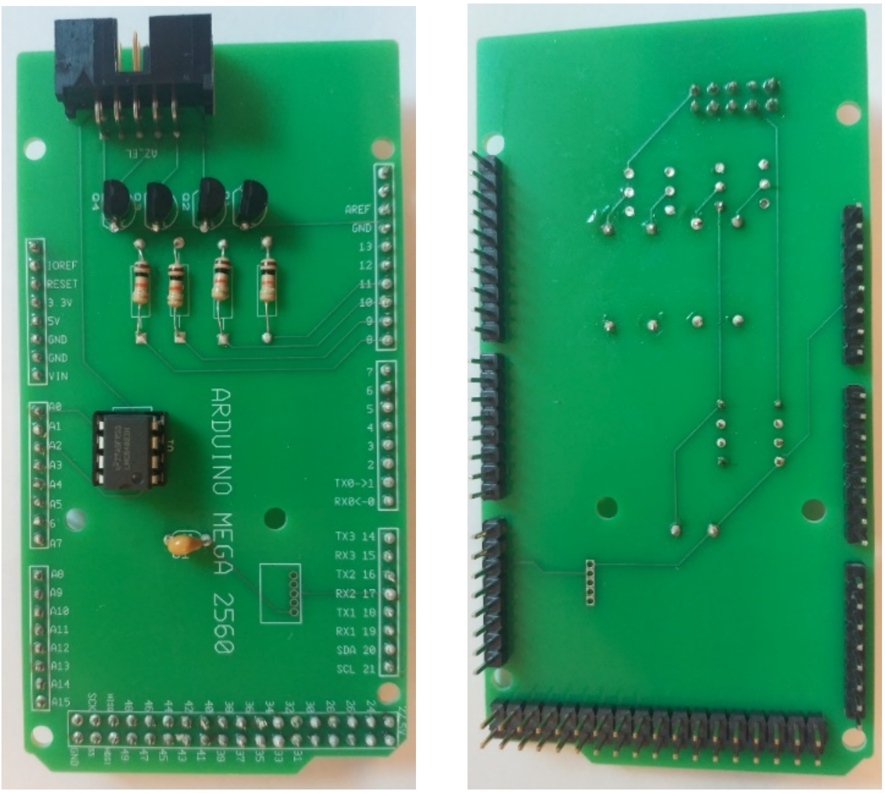


Figure A.7: ERAU Ground Station Rotor Control Arduino Mega2560 Shield

A.5.2 Calibration

First of all, the rotor gauges need to be calibrated. To do that, there are two screws on the back of the rotor that can be adjusted after the rotor is in the minimum azimuth and elevation positions. Once the gauges are calibrated, an algorithm to calibrate the analog signals of the Arduino for each rotor position is considered. In order to use this code, there are two screws on the rotor box that must be adjusted to be able to cover the expected azimuth and elevation range with the available Arduino analog signal range. The G-5500 rotor control box has a 8 pin DIN external control connection that controls the different movements by connecting the respective pin to the Az/El positions of the rotor. There are two pins that supply a DC voltage from 2 to 4.5 V corresponding to azimuth and elevation positions from the rotor. To calibrate these positions, the rotor can be manually moved to Azimuth: 360 degrees and Elevation: 120 degrees. Then, the rotor calibration algorithm is used to read the analog values for those azimuth and elevation positions. For both of them, we expect the analog values to be approximately at 95% of the maximum expected value, to ensure enough resolution and to avoid possible rotor errors. To do that, the screws of the rotor box are used to reduce the signal level send to the Arduino in that position. Once the maximum value is adjusted, the rotor is manually moved to different positions, while the arduino analog readings are being considered. With those equivalences, a polynomial fit is used to be able to translate the analog readings to actual rotor positions for both azimuth and elevation coordinates.

The aforementioned calibration should be performed when a new G-5500 is assembled, or anytime there is reason to believe the OUT VOL ADJ set screws may have been adjusted. If the microcontroller of the operational amplifier have been replaced, but the G-5500 OUT VOL ADJ screws have NOT been adjusted, then the steps before step 14 can be omitted. Follow the next steps for a full rotor calibration process:

- Use the G-5500 Control to rotate the azimuth and elevation of the G-5500 fully left and down using the respective rocker switches until the limit of movement is reached. The G-5500 has limit switched internal to the rotor.
- If the gauges above the Elevation and Azimuth do not read 0, use the small 0 adjust set screw on the bottom of the respective gauges to zero the gauges.
- Attach a voltmeter to measure the voltage between pin 1 and pin 6.
- Mark the position of the Azimuth housing across the rotating section. There is a small raised vertical line on the upper portion of the Azimuth rotator that makes a good reference to align a mark for the lower portion.
- Rotate the azimuth rotor clockwise 1 complete revolution until the marks are realigned using the RIGHT rocker switch.
- Use the FULL SCALE ADJ set screw above the AZIMUTH connection on the back of the Rotor Control to adjust the reading of the azimuth gauge until the gauge reads 360°.
- Rotate the azimuth rotor clockwise to the end-stop using the RIGHT rocker switch.
- Use the OUT VOL ADJ set screw above the AZIMUTH connection on the back of the Rotor Control to adjust the voltage reading on the voltmeter to 2.5V.

- Attach a voltmeter to measure the voltage between pin 1 and pin 8.
- Notice the markings on the elevation rotor. There is an indication line and the raised portion on the housing will indicate 0°, 90°, and 180°.
- Rotate the elevation rotor clockwise 1 the indicator and the 180° mark are realigned using the UP rocker switch.
- Use the FULL SCALE ADJ set screw above the Elevation connection on the back of the Rotor Control to adjust the reading of the azimuth gauge until the gauge reads 180°.
- Use the OUT VOL ADJ set screw above the ELEVATION connection on the back of the Rotor Control to adjust the voltage reading on the voltmeter to 2V.
- Connect the USB Rotor Control to the G-5500 control. If the controller software has not been flashed, do so now.
- Open a hyper terminal and connect to the USB Rotor Control.
- Rotate the Azimuth and Elevation to 0° using their respective switches.
- Use the intCal command to start the calibration routine
- Open an Excel spreadsheet and record the displayed counts that correspond with the rotor position. Rotate the rotor to the indicated lines on the gauges starting with the Azimuth. The lines are in 15° increments on the Azimuth and 7.5° increments for the elevation.
- Using the average of a minimum of 3 runs for each. Fit a trendline to determine the conversion between ADC counts and degrees.
- Edit the source code of the getAzDegrees() and the getElDegrees() functions to match the results of step 19.
- Flash the new source code to the Arduino.

A.6 Antenna Module

For the antenna module, a boom to connect the antenna to the rotor is required. The size of the boom will depend on the size and the number of antennas used for communication purposes, while the diameter will be limited by the rotor. The antenna will depend on the frequency band selected for the communications between the HAB payload and the ground station. The antenna considered for the ERAU ground station is a 900MHz – 17dBi Yagi antenna. For this antenna, a boom of approximately 6ft long and 1-inch OD is used. A 6ft by 1-inch OD boom is commercially available for $8-10. If you are interested in using the same frequency band and antenna, the model used on the ERAU ground station is the "MSQ-90217" from DXEngineering/M2inc with a cost of $88.95.

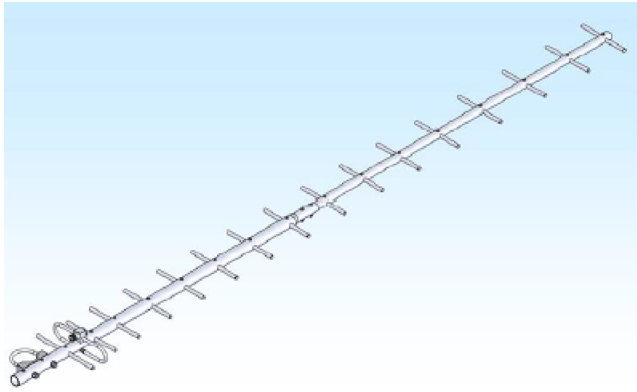


Figure A.8: Ground Station antenna: 900MHz-17 dBi Yagi antenna.

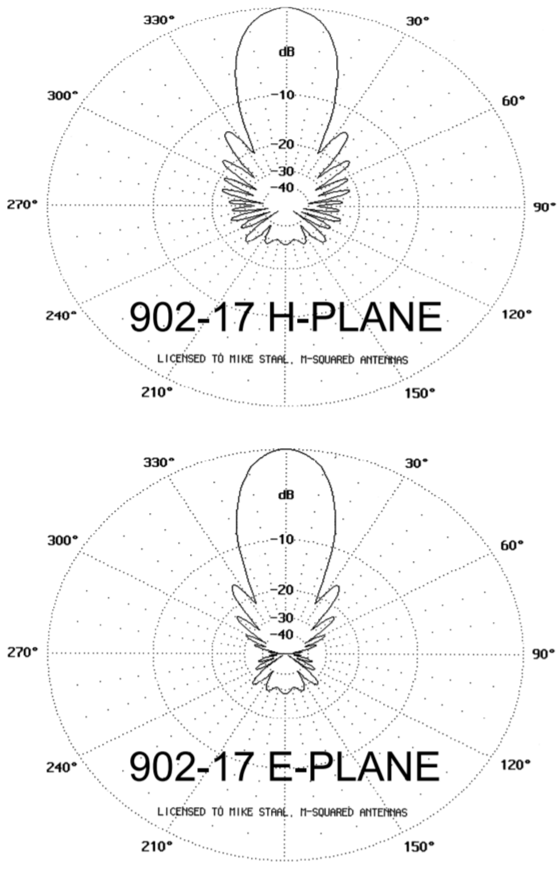


Figure A.9: Ground Station antenna: (A) H and (B) E planes radiation patterns.

Appendix B

Tracking System

This appendix contains design process to develop and use the payload tracking system, from the payload perspective to the antenna pointing calibration.

B.1 GNSS Sensor

In this section, the configuration and decoding of the data from the GNSS receiver of the payload is presented. The receiver is used to obtain the payload 3D position in order to point the antenna towards it.

The sensor used for the presented payload designs was uBlox M8N. This GNSS module provides high sensitivity, customizable configurations and an altitude operational limit of 50.000 meters, which makes it compliance with the project requirements. Moreover, the low power consumption of these devices make them easy to operate with hobby boards, such as Arduino or Teensy.

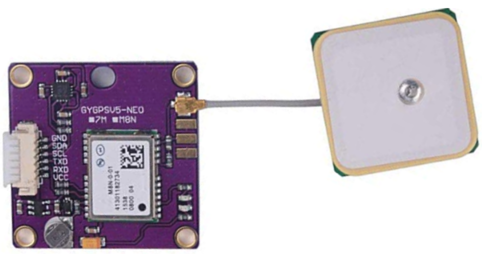


Figure B.1: GPS Module

B.1.1 Sensor Configuration

uBlox provides GNSS evaluation software for their devices, including configuration and control features, as well as real-time displays for the received data: the uBlox Center or uCenter.

Connecting the uBlox device to a computer (FTDI-UART USB cable) and opening a connection on the u-center, several real-time displays for the data received from the device can be seen once the device is properly configured.

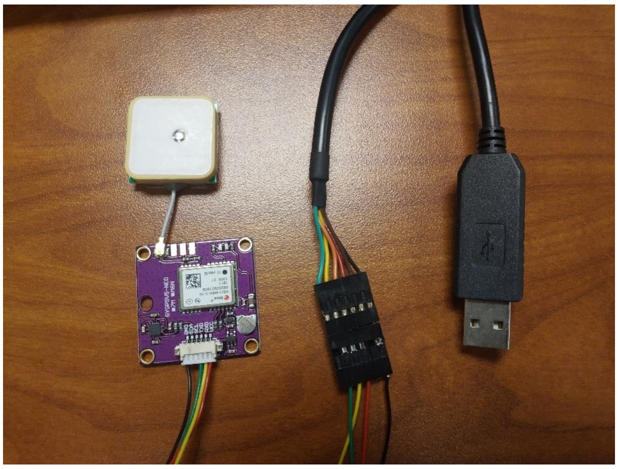


Figure B.2: Ublox USB Connection

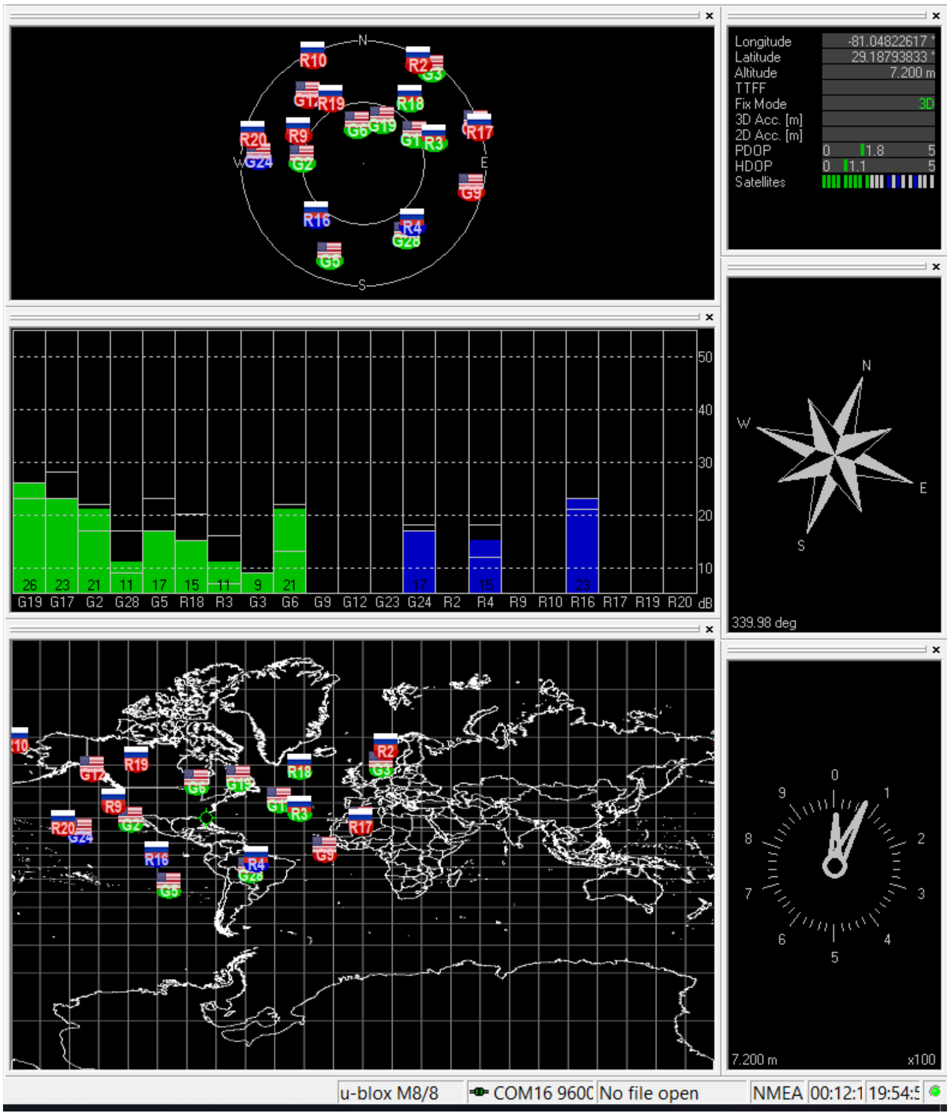


Figure B.3: uBlox Center - Information View

The following steps presents how to configure the devices using uCenter.

- uCenter View: Configuration View.

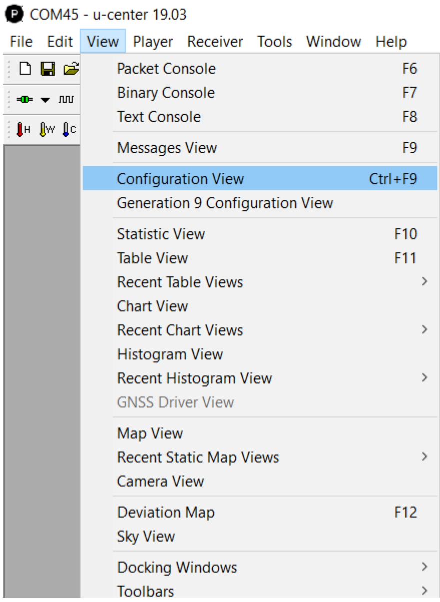


Figure B.4: Ublox Center View for Configuration

- Reference Datum: 'WGS 84'.

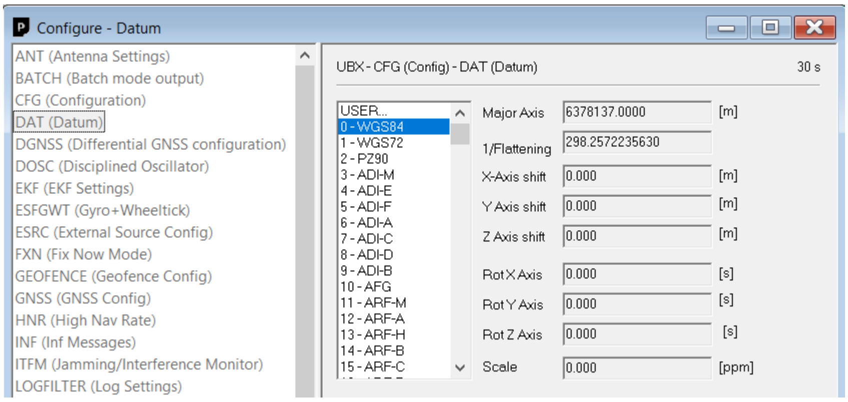


Figure B.5: uBlox Datum Configuration

- Message Configuration: GGA, GSA, GSV, GNS.

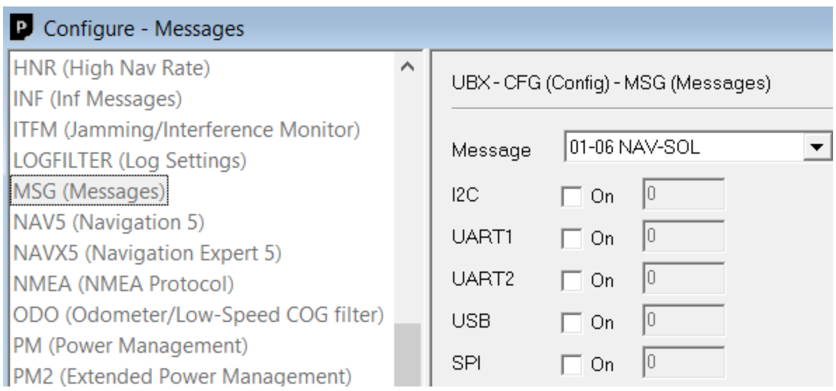


Figure B.6: uBlox Message Configuration. Output NMEA Sentences.

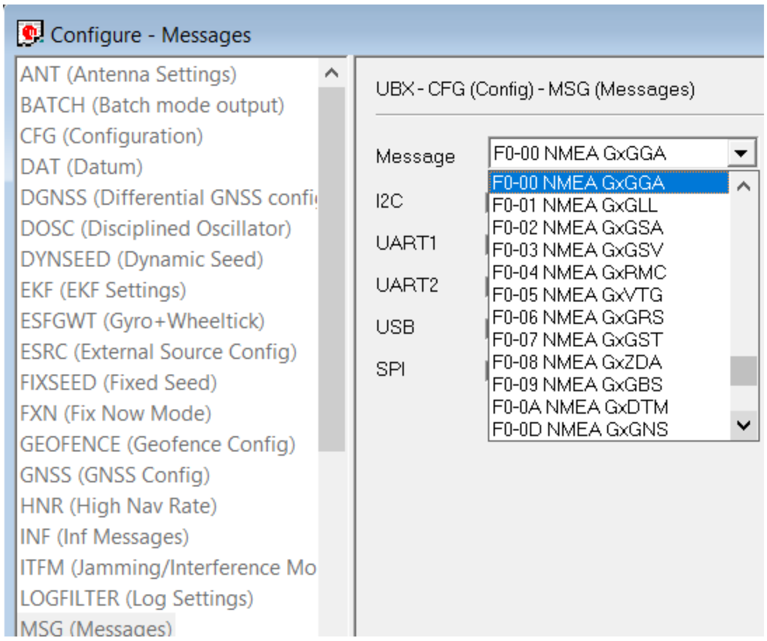


Figure B.7: uBlox Message Configuration. NMEA Message Selection.

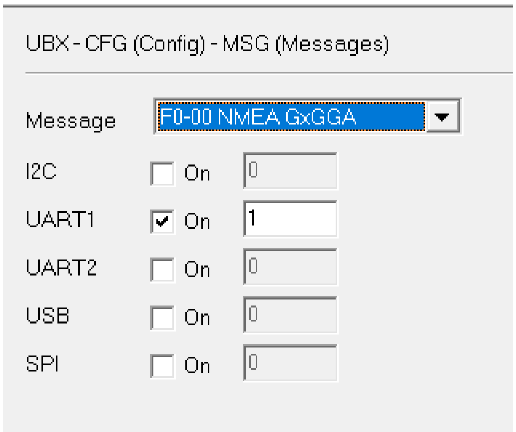


Figure B.8: uBlox Message Configuration. Configured NMEA Message - Communication Protocol Selected.

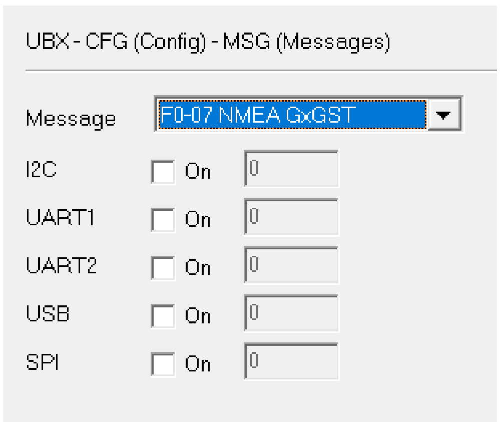


Figure B.9: uBlox Message Configuration. Not configured NMEA Message.

- Navigation Mode Configuration: Dynamic Model - Airborne 1g, Fix Mode - 3D fix type only.

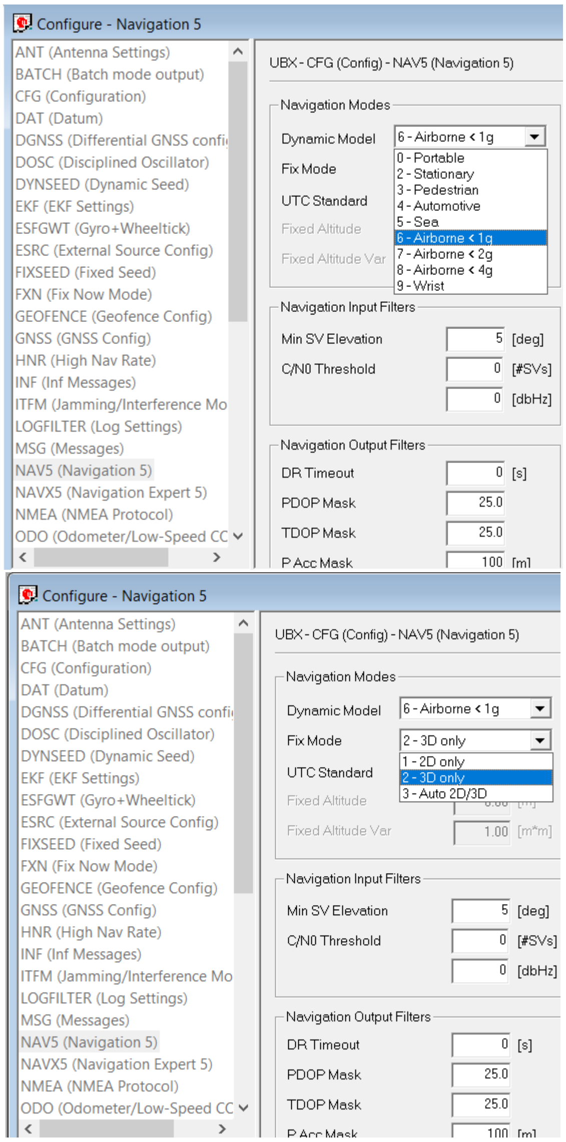


Figure B.10: uBlox Navigation Mode Configuration

- Ports Configuration: UART, NMEA Protocol, 9600 bps 8N1.

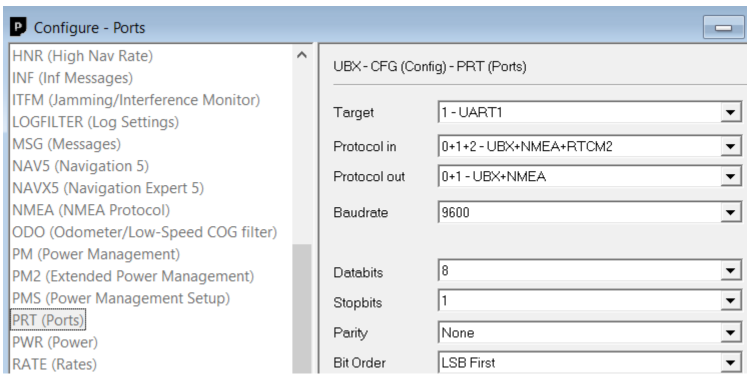


Figure B.11: uBlox Ports Configuration

- Measurement Period/Frequency: 1000 - 200 ms (1 – 5 Hz).

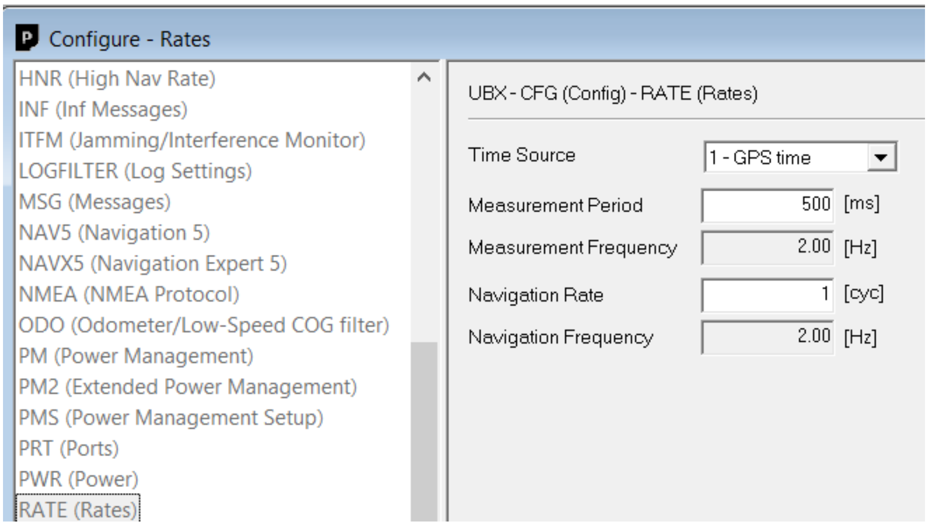


Figure B.12: uBlox Measurement Period/Frequency Configuration

The LLA parameters are also used to analyse the sensors data at each position. Therefore, in order to obtain cm-scale accuracy, the sampling rate of the receiver shall be as high as possible, always ensuring a minimum number of satellites in view to be able to track the payload properly.

The selected model, uBlox M8N, can work at 10 Hz if only GPS satellites are considered. However, using any other combination of satellites constellations, 5 Hz is the maximum achievable sampling rate. In the previously presented designs, the maximum configured sampling rate was 5 Hz, using normally GPS and GLONASS as the main satellite constellations being used to obtain the payload LLA parameters.

- Save Configuration: EEPROM, FLASH.

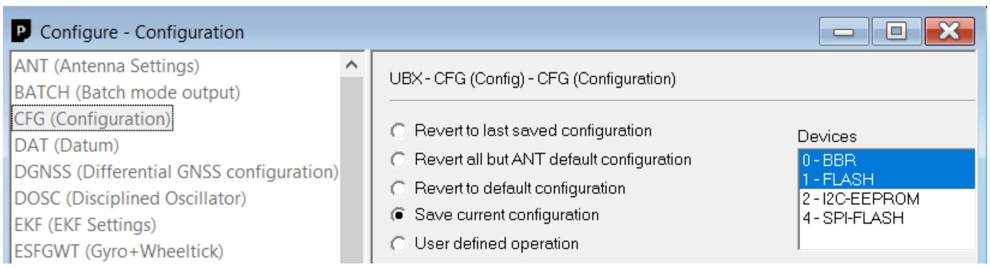


Figure B.13: uBlox Saving Configuration

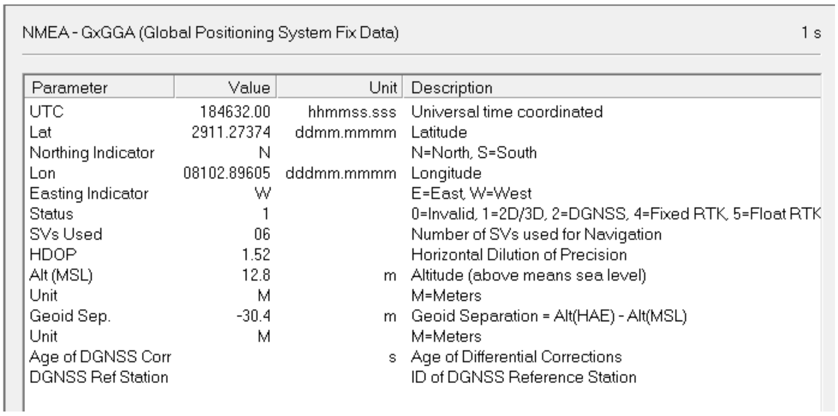
There are different view options for the actual NMEA messages and information being received. If the Message View option is selected, the directly parsed information from the NMEA sentences can be examined. By using this, the uBlox decoded information by the u-center can be compared with the Arduino library that is used in the final application to decode these sentences.

B.1.2 Raw Output - NMEA Sentences

For position and accuracy purposes, NMEA -GxGGA and NMEA -GxGSA sentences are considered, as it can be seen in the following figures:

```
$xxGGA,time,lat,NS,long,EW,quality,numSV,HDOP,alt,M,sep,M,diffAge,diffStation*cs<CR><LF>
```

Figure B.14: NMEA - GxGGA Sentences Format



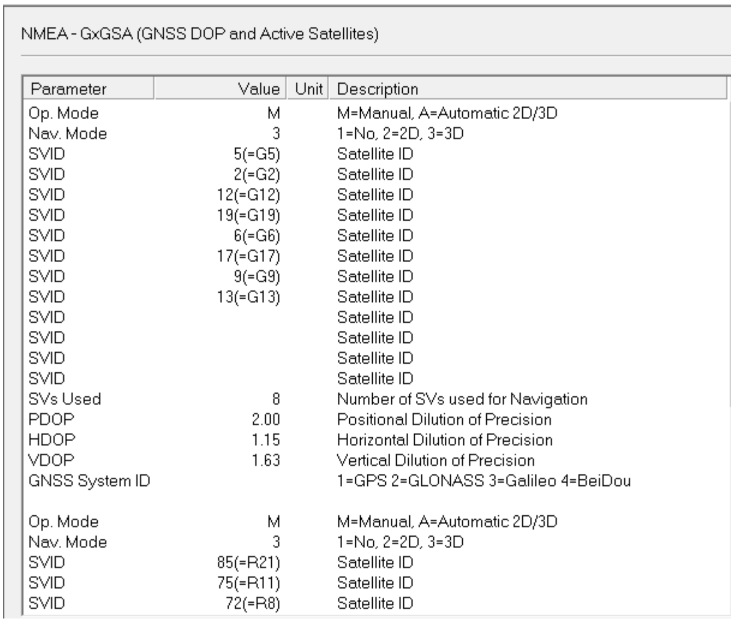
NMEA - GxGGA (Global Positioning System Fix Data) 1 s

Parameter	Value	Unit	Description
UTC	184632.00	hhmmss.sss	Universal time coordinated
Lat	2911.27374	ddmm.mmmm	Latitude
Northing Indicator	N		N=North, S=South
Lon	08102.89605	dddmm.mmmm	Longitude
Easting Indicator	W		E=East, W=West
Status	1		0=Invalid, 1=2D/3D, 2=DGNSS, 4=Fixed RTK, 5=Float RTK
SVs Used	06		Number of SVs used for Navigation
HDOP	1.52		Horizontal Dilution of Precision
Alt (MSL)	12.8	m	Altitude (above means sea level)
Unit	M		M=Meters
Geoid Sep.	-30.4	m	Geoid Separation = Alt(HAE) - Alt(MSL)
Unit	M		M=Meters
Age of DGNSS Corr		s	Age of Differential Corrections
DGNSS Ref Station			ID of DGNSS Reference Station

Figure B.15: NMEA -GxGGA uBlox Center View

```
$xxGSA,opMode,navMode{,sv},PDOP,HDOP,VDOP,systemId*cs<CR><LF>
```

Figure B.16: NMEA - GxGSA Sentences Format



NMEA - GxGSA (GNSS DOP and Active Satellites)

Parameter	Value	Unit	Description
Op. Mode	M		M=Manual, A=Automatic 2D/3D
Nav. Mode	3		1=No, 2=2D, 3=3D
SVID	5(=G5)		Satellite ID
SVID	2(=G2)		Satellite ID
SVID	12(=G12)		Satellite ID
SVID	19(=G19)		Satellite ID
SVID	6(=G6)		Satellite ID
SVID	17(=G17)		Satellite ID
SVID	9(=G9)		Satellite ID
SVID	13(=G13)		Satellite ID
SVID			Satellite ID
SVID			Satellite ID
SVID			Satellite ID
SVID			Satellite ID
SVs Used	8		Number of SVs used for Navigation
PDOP	2.00		Positional Dilution of Precision
HDOP	1.15		Horizontal Dilution of Precision
VDOP	1.63		Vertical Dilution of Precision
GNSS System ID			1=GPS 2=GLONASS 3=Galileo 4=BeiDou
Op. Mode	M		M=Manual, A=Automatic 2D/3D
Nav. Mode	3		1=No, 2=2D, 3=3D
SVID	85(=R21)		Satellite ID
SVID	75(=R11)		Satellite ID
SVID	72(=R8)		Satellite ID

Figure B.17: NMEA -GxGSA uBlox Center View

In Figure B.14, GGA sentences has a “quality” field. On the other hand, in Figure B.16 GSA messages contains a “navMode” field. Considering both fields, one can draw conclusions about the quality of the information that it is being received. Specially in terms of altitude, that a “3D fix” should be considered.

NMEA Message *Field*	*GLL, RMC* *status*	*GGA* *quality*	*GSA* *navMode*	*GLL, VTG, RMC, GNS* *posMode*
NMEA Message *Field*	*GLL, RMC* *status*	*GGA* *quality*	*GSA* *navMode*	*GLL, VTG, RMC, GNS* *posMode*
No position fix (at power-up, after losing satellite lock)	V	0	1	N
GNSS fix, but user limits exceeded	V	0	1	N
Dead reckoning fix, but user limits exceeded	V	6	2	E
Dead reckoning fix	A	6	2	E
2D GNSS fix	A	1 / 2	2	A / D
3D GNSS fix	A	1 / 2	3	A / D
Combined GNSS/dead reckoning fix	A	1 / 2	3	A / D
	See below (1)	See below (2)	See below (3)	See below (4)

(1) Possible values for *status*: V = Data invalid, A = Data valid

(2) Possible values for *quality*: 0 = No fix, 1 = Autonomous GNSS fix, 2 = Differential GNSS fix, 4 = RTK fixed, 5 = RTK float, 6 = Estimated/Dead reckoning fix

(3) Possible values for *navMode*: 1 = No fix, 2 = 2D fix, 3 = 3D fix

(4) Possible values for *posMode*: N = No fix, E = Estimated/Dead reckoning fix, A = Autonomous GNSS fix, D = Differential GNSS fix, F = RTK float, R = RTK fixed

Figure B.18: Status, Quality, Navigation Mode NMEA Messages Parameters

To clarify some of the quality values, the following descriptions should be considered:

- **Differential GNSS fix** provides a higher accuracy than Autonomous GNSS fix. This technique uses a network of fixed ground reference stations to broadcast the difference between the positions indicated by the satellite systems and the known fixed positions.

- **Real Time Kinematic (RTK) fixed** satellite navigation is a technique used in land survey based on the use of carrier phase measurements of the GPS, GLONASS and/or Galileo signals where a single reference station provides the real-time corrections of even to a centimeter level of accuracy.

- **Float Real Time Kinematic (RTK Float)** is very similar to the fixed RTK method of calculating location, but is not as precise, typically around 20 cm to 1-meter accuracy range

- **Estimated Fix or Dead Reckoning** is the determination of a location based on computations of position given an accurately known point of origin and measurements of speed, heading and elapsed time. Dead reckoning can be used to “fill in the gaps” when there is insufficient satellite signal strength to obtain an accurate position.

  In the next section, it can be seen how the considered Arduino library is merging both fields to extract the status information of the received data.

B.1.3 NEOGPS Library

```
21:04:25  $GNGLL,2911.27387,N,08102.89996,W,210425.00,A,A*6B

21:04:26  $GNRMC,210426.00,A,2911.27373,N,08102.89983,W,0.091,,050718,,,A*7D
21:04:26  $GNVTG,,T,,M,0.091,N,0.168,K,A*3A
21:04:26  $GNGGA,210426.00,2911.27373,N,08102.89983,W,1,09,0.98,37.9,M,-30.4,M,,*4D
21:04:26  $GNGSA,M,3,12,19,05,06,17,09,,,,,,,2.30,0.98,2.08*1D
21:04:26  $GNGSA,M,3,85,71,75,,,,,,,,,,2.30,0.98,2.08*13
21:04:26  $GPGSV,2,1,08,05,50,208,31,06,43,038,18,09,17,071,23,12,52,284,27*7E
21:04:26  $GPGSV,2,2,08,17,30,109,13,19,49,092,12,23,02,043,,24,00,240,*77
21:04:26  $GLGSV,3,1,09,69,20,034,,70,60,092,,71,36,171,27,74,02,198,*6E
21:04:26  $GLGSV,3,2,09,75,23,245,26,76,21,294,,84,21,080,,85,46,020,19*66
21:04:26  $GLGSV,3,3,09,86,21,317,*54
21:04:26  $GNGLL,2911.27373,N,08102.89983,W,210426.00,A,A*67

21:04:27  $GNRMC,210427.00,A,2911.27357,N,08102.89976,W,0.081,,050718,,,A*71
```

Figure B.19: uBlox Sensors Serial Output: NMEA Messages

In the previous figure, an example of the NMEA sentences that the uBlox is sending once per second can be seen. Among others, the GxGGA and GxGSA sentences can be detected.

NEOGPS is the Arduino library that Elegoo Mega2560 is using to parse the uBlox outputs and to create the structs with the most important information easily accessible. Moreover, this library can and should be properly prepared and analyzed for the project application.

The following header files are considered when using this library:

- GPSport.h: used to declare your own GPS port variable, GPS port name string, and debug print port (radio-Arduino main serial) variable. It can be really useful to avoid possible errors/confusions if more than one GNSS sensor are used for the same payload.
- NMEAGPS_cfg.h: used to enable/disable the parsing of specific sentences.
- GPSfix.h: used to check the expected output from the available functions to have access to the latitude, longitude, altitude and fix status information.

B.1.4 Microcrontroller Parsing - Encoding

In order to study the access to the uBlox NEO M8N and Copernicus II sensors and their output parsing with the Arduino library, a sample code was implemented. The Arduino loop checks if there are available bytes on the Arduino buffers for both serial connections. If there are available bytes with the expected NMEA sentences, the gps_fix variable is filled with the latest values sent to the serial. After that, the code checks if a valid location was received and only in that case the parameters of interest are considered and printed on the Arduino Serial Console for monitoring purposes.

To check if the updated frequency is configured as expected and how much time the Arduino needs is using to parse the data, the Arduino code includes time references before and after reading from the uBlox serial port and after parsing the data.

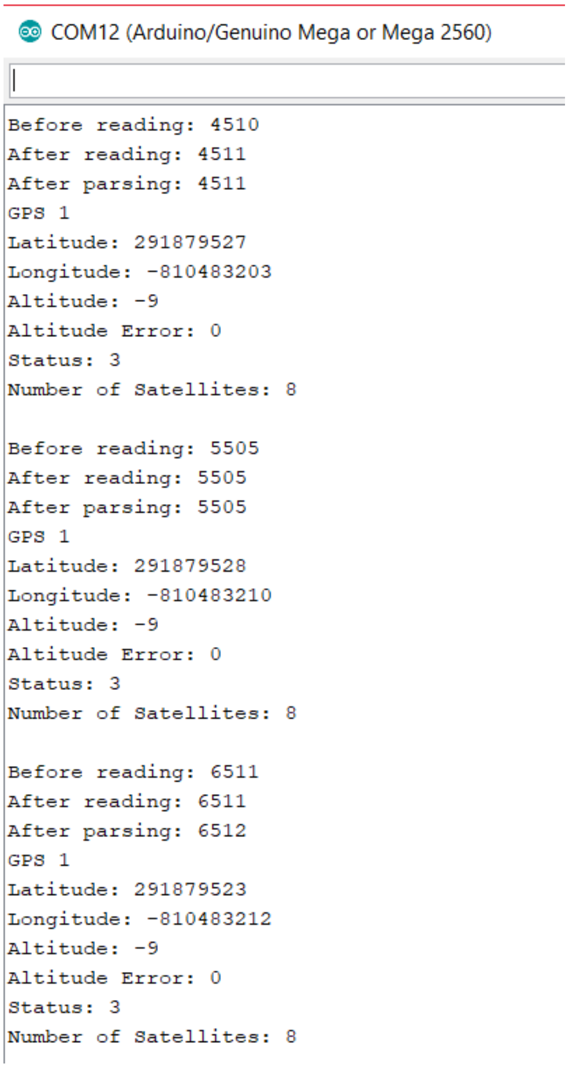


Figure B.20: Arduino Library Program Time References.

As it can be seen in Figure B.20, there are available bytes from the uBlox sensors approximately every second. Moreover, it takes only about 1 ms to read, validate and parse the data. These type of checks can be performed using the uCenter in packet view.

The microcontroller of the payload will be in charge of the data packets creation -information encoding-. Before sending any information to the on-board transceiver, it will check if there is data available from the GPS and it will parse it using the following code, and encode it in using a specific format.

```
 while (uBloxEX.available( gpsuBloxExt ))
 {
      uBloxEXFix = uBloxEX.read();
      if (uBloxEXFix.valid.location) {
        lat1 = uBloxEXFix.latitudeL();
        lon = uBloxEXFix.longitudeL();
        alt = uBloxEXFix.altitude_cm();
        stat = uBloxEXFix.status;
        numSats = uBloxEX.sat_count;
        utcHour = uBloxEXFix.dateTime.hours;
        utcMin = uBloxEXFix.dateTime.minutes;
        utcSec = uBloxEXFix.dateTime.seconds;

        send_packet(3);
      }
}
```

B.1.5 GS GUI

B.1.5.1 Coordinates Conversion and Presentation

On the ground station segment, when a GPS data packet is successfully received, the data is decoded, prepared to be directly printed on the MATLAB GUI.

1. Packet Decoding: the packet format considered in MATLAB matches the one considered by the payload during its creation. In this case, the packet identifier specifies that the packet contains GNSS data and the information of interest is decoded as it can be seen in Figure B.21.

```
lat = double(typecast(uint8(messages(5:8)),'int32'))/10000000;
lon = double(typecast(uint8(messages(9:12)),'int32'))/10000000;
h = double(typecast(uint8(messages(13:16)),'int32'))/100;
stat = messages(17);
numSats = messages(18);
utcHour = messages(19);
utcMin = messages(20);
utcSec = messages(21);
gps_time = typecast(uint8(messages(27:30)),'uint32');
```

Figure B.21: MATLAB Code Decoding Sample.

2. Pre-conversion – GUI Data Presentation: the data decoded is presented in the GUI for monitoring purposes. Part of the data is considered for additional parameters computation and presentation, such as ascent/descent rates.

```
%Print Current GPS Data
app.gpsLAT.Text  = num2str(lat);
app.gpsLONG.Text = num2str(lon);
app.gpsALT.Text  = num2str(h);
app.gpsFIX.Text = num2str(stat);
app.gpsSATS.Text = num2str(numSats);
app.gpsHOUR.Text = [num2str(utcHour), ':'];
app.gpsMIN.Text = [num2str(utcMin), ':'];
app.gpsSEC.Text = num2str(utcSec);
```

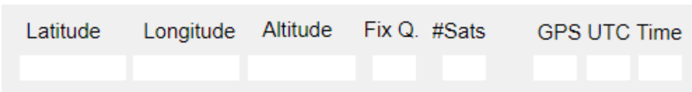


Figure B.22: MATLAB GUI Position Sample.

While the GUI is showing the latest received data, the latitude, longitude and altitude values are converted to azimuth, elevation and range.

The range will be plotted on the GUI and it will be used as one of the thresholds to determine how many GPS packets per second will be evaluated to move the ground station antennas.

The azimuth and elevation values will be sent to the ground station rotor box controller for antenna pointing purposes to track the payload.

B.1.5.2 Rotor Communication

The next step is to send the Az/El coordinates to the rotor to point the antennas to the payload position. To do that, there are different available modes:

1.- Manual: the rotor is not moving based on the received data.

2.- Az/El: the rotor is moving completely based on the received data.

3.- Only AZ: the rotor is only moving horizontally considering the received data and the ground station user should control the vertical pointing.

4.- Only EL: the rotor is moving only vertically considering the received data and the ground station user should control the horizontal pointing.

Please consider that the “Elevation Tuning” value specified in the GUI field will be added to the elevation value computed, as well as the “Declination” value will be added to the azimuth computed. If during the flight, some pointing offsets are detected, these fields can be changed in real-time to compensate the pointing errors.

B.2 Rotor Controller

In this section, the rotor controller codes for calibration and tracking are presented.

B.2.1 Calibration Code

The code used to calibrate the rotor controller of the ground station is the one that can be seen below:

```
// Pin definitions
const int _elSensePin = A2;
const int _azSensePin = A0;

void setup() {
  Serial.begin(230400);
  while (!Serial) {};              // Wait for serial to connect for native USB connection
  pinMode( _elSensePin , INPUT); // Elevation ADC input
  pinMode( _azSensePin , INPUT); // Azimuth ADC input
}
/*****************************************************************
 * A continuous 16 count average of the Azimuth counts
 * are sent to the Serial object. When the integer ’1’ is sent to arduino, the source becomes the elevation
 *   counts until the integer ’2’ is sent to the *arduino. Then the loop will restart.
 */
void loop() {
  int ans = 0;
  while (ans != 1){
      int sum = 0;
      for (int k = 1; k<17; k++){
        sum += analogRead(_azSensePin);
      }
      Serial.print("Az count: ");
      int total = sum/16;
      Serial.println(total);
      while (Serial.available()) {
        ans = Serial.parseInt();
      }
  }
  while (ans != 2){
      int sum = 0;
      for (int k = 1; k<=16; k++){
        sum += analogRead(_elSensePin);
      }
      Serial.print("El count: ");
      Serial.println(sum/16);
      while (Serial.available()) {
        ans = Serial.parseInt();
      }
  }
}
```

B.2.2 Tracking Code

The code used to track the payload of to point the rotor to a desired position can be seen below:

```
/*******************************************************************
 * NAME: GSRotor_v2.ino
 * AUTHOR: Nick Purvis, Noemi Miguelez Gomez (miguelen.my.erau.edu)
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON.
 *
 * DEVELOPMENT HISTORY:
 *   Date     Author  Version            Description Of Change
 * --------  ------  -------  ------------------------------------
 * 06/12/2018   NP     1             Rotor Controller Logic
 * 26/05/2019   NMG    2             Linear Fit Az/El Calibration and Real-Time GS Position
 ******************************************************************/


#include <NMEAGPS.h>
static NMEAGPS  uBloxEX;   //uBlox GPS
static gps_fix  uBloxEXFix;
int id;

int32_t lati;    //Latitude
int32_t lon;     //Longitude
int32_t alt;     //Altitude

int validFlag;

/**************************************************************

   Global values and Definitions

 *************************************************************/
// constant pin variables
const int _upPin = 10;
const int _downPin = 11;
const int _ccwPin = 8;
const int _cwPin = 9;
const int _elSensePin = A2;
const int _azSensePin = A0;
const int _minAzPoint=0;
const int _minElPoint=0;


// enumeration for movement switch case
enum rotor {off, UP, DOWN, CW, CCW, azOff, elOff};

// Flags
bool position_flag = false;
bool cmdFlag = false;
bool elFlag = false;
bool azFlag = false;

// String for serial command decoding
String cmdString = "";

// Constants for the position adc calculations and movement ranges
volatile float globalAz = 0;
volatile float globalEl = 0;
const int _maxAzPoint = 450;
const int _maxElPoint = 180;

//Set for ~2 deg dead zones to avoid chattering the motors
const int _azDeadZone = 1.5;
const int _elDeadZone = 2;

/****************************************************************

   SETUP

 ***************************************************************/

void setup() {
  Serial.begin(230400); // Set Baud rate to 230400
  Serial2.begin(9600);

  while (!Serial) {}; // Wait for serial to connect for native USB connection
  //Set each of the respective pins to IO type
  pinMode(_upPin , OUTPUT);
  pinMode(_downPin , OUTPUT);
  pinMode(_ccwPin , OUTPUT);
  pinMode(_cwPin , OUTPUT);
  pinMode( _elSensePin , INPUT);
  pinMode( _azSensePin , INPUT);

  //Write a low logic voltage value to each of the pins centered around up, down,
  // cw, and ccw
  digitalWrite(_upPin, LOW);
  digitalWrite(_downPin , LOW);
  digitalWrite(_ccwPin , LOW);
  digitalWrite(_cwPin , LOW);

  cmdString = "";
    // handShaking is the act of controling the data transmission between two systems
    //or devices
  Serial.println('1');
```

```
  analogReference(INTERNAL2V56);
}



/****************************************************************

    Main Function

 ****************************************************************/

void loop() {
  if (Serial.available()) {
    cmdString = Serial.readString();
    cmdFlag = true;
  }
  if (cmdFlag == true) {
    serialParse();
  }
  if (position_flag == true) {
    setPosition();
    position_flag = false;
  }
}

/***************************************************************

    Sensing functions for the Rotor

 ***************************************************************/
//Function to return the azimuth based on voltage from pin and number of counts
float getAzDegrees()
{
  int azInd = analogRead(_azSensePin);
  for (int i=0; i<15; i++) {
    azInd += analogRead(_azSensePin);
  }
  azInd = azInd/16;
  // Edit below to convert from counts to degrees from measurement tredline
  float azimuth = float(0.40545 * azInd - 3.35577 ); //Linear fit Rotor 1

  if (azimuth < 0) azimuth = 0;
  else if (azimuth > 450) azimuth = 450;
  return azimuth;
}

//Function to return the elevation degrees based on voltage from pin and number
//of counts
float getElDegrees()
{
  int elInd = analogRead(_elSensePin);
  for (int i=0; i<15; i++) {
    elInd += analogRead(_elSensePin);
  }
  elInd = elInd/16;
  // Edit below to convert from counts to degrees from measurement tredline
  float elevation = float(0.19925*elInd - -0.960655223701884); //Linear fit Rotor 1

  //the coeffs from matlab calibration are inputted
  if (elevation < 0)
      elevation = 0;
  else if (elevation > 180)
      elevation = 180;
  return elevation;
}

/*****************************************************************

    Rotor pointing function

*****************************************************************/

void setPosition() {
  // The comanded poistion ( globalAz and globalEl ) are checked against the max and min range of the rotor.
  // If not within the range, the commanded postion become the max or min based on whether over or under
  //operating range.
  if (globalAz > _maxAzPoint) globalAz = _maxAzPoint;
  if (globalAz < _minAzPoint) globalAz = _minAzPoint;
  if (globalEl < _minElPoint) globalEl = _minElPoint;
  if (globalEl > _maxElPoint) globalEl = _maxElPoint;

  // Current position is read from the respective adc and converted to degrees.
  float azInd = getAzDegrees();
  float elInd = getElDegrees();

  // If rotor position is within the deadzone for both Az and El then all movement stops.
  // Solved error when changing commanded postion in the middle of a movement.
  //Prevents rotor from trying to move further than allowed or desired.
  if ((abs(globalAz - azInd) <= _azDeadZone) && (abs(globalEl - elInd) <= _elDeadZone))
    pointRotor(off);

  // While either rotor is not at the desired position, loop to move rotor begins.
  while ((abs(azInd - globalAz) >= _azDeadZone) || (abs(globalEl - elInd) >= _elDeadZone)) {

    // Accepts new commands while in the process of moving.
    while (Serial.available()) {
      cmdString = Serial.readString();
      cmdFlag = true;
    }
```

```

    // Parses command string and allows position flag to set.
    if (cmdFlag == true) {
      serialParse();
      position_flag = false;
    }

    // The comanded poistion ( globalAz and globalEl ) are checked against the max and min range of the rotor.
     //If not within the range, the commanded postion become the max or min based on whether over or under
    //operating range. Needed again incase new position entered by receieved command.

    if (globalAz > _maxAzPoint) globalAz = _maxAzPoint;
    if (globalAz < _minAzPoint) globalAz = _minAzPoint;
    if (globalEl < _minElPoint) globalEl = _minElPoint;
    if (globalEl > _maxElPoint) globalEl = _maxElPoint;

    // If indicated az is withing the deadzone centered on commanded az, stop moving Az.
    if (abs(globalAz - azInd) <= _azDeadZone){
      pointRotor(azOff);
      azFlag = false;
      }

    // move towards commanded position.
    else if  ((azInd < globalAz) && azFlag) pointRotor(CW); // Go CW
    else if ((azInd > globalAz) && azFlag) pointRotor(CCW); // GO CCW

    // If indicated El is withing the deadzone centered on commanded El, stop moving El.
    if (abs(globalEl - elInd) <= _elDeadZone) {
    pointRotor(elOff);
    elFlag = false;}

    // Move towards commanded El
    else if ((elInd < globalEl) && elFlag) pointRotor(UP); // Go UP
    else if ((elInd > globalEl) && elFlag)pointRotor(DOWN); // Go DOWN

    // Check current positions
    elInd = getElDegrees();
    azInd = getAzDegrees();
  }
  // Turn rotors off if outside of while.
  pointRotor(off);
}



/**********************************************************
 * pointRotor() is the funtion to move the rotor. Directions are UP, DOWN, CW, CCW defind as an ENUM.
 * A High is sent to a group of transitor switches that closes a circuit of the corresponding control wire
 * and the ground of the extenal control of the YAESU GS-5500. HIGH moves in direction indicated.
 *  All LOW stops. HIGH on opposing directions can damage equipment.
 */


void pointRotor(rotor x) {
  // Swtich/Case to control rotor direction. HIGH moves in direction indicated. All LOW stops.
  //HIGH on opposing directions can damage equipment.

  switch (x)
  {
    case off:
      digitalWrite(_upPin , LOW);
      digitalWrite(_downPin , LOW);
      digitalWrite(_ccwPin , LOW);
      digitalWrite(_cwPin , LOW);
      break;

    case UP:
      digitalWrite(_upPin , HIGH);
      digitalWrite(_downPin , LOW);
      break;

    case DOWN:
      digitalWrite(_upPin , LOW);
      digitalWrite(_downPin , HIGH);
      break;

    case CW:
      digitalWrite(_ccwPin , LOW);
      digitalWrite(_cwPin , HIGH);
      break;

    case CCW:
      digitalWrite(_ccwPin , HIGH);
      digitalWrite(_cwPin , LOW);
      break;

    case azOff:
      digitalWrite(_ccwPin , LOW);
      digitalWrite(_cwPin , LOW);
      break;

    case elOff:
      digitalWrite(_upPin , LOW);
      digitalWrite(_downPin , LOW);
      break;

  }
}

```

```
/*****************************************************************

    Read and Parse Serial Commands

 *****************************************************************/

// azFlag and elFlag determine if new respective movement command has been recieved.
// position_flag tells the main loop() that new position has been recieved.

void serialParse()
{
  if (cmdString.substring(0, 4).equals("ElAz")) {
    globalEl = cmdString.substring(4, 7).toInt();
    globalAz = cmdString.substring(7, 10).toInt();
    cmdString = "";
    cmdFlag = false;
    position_flag = true;
    azFlag = true;
    elFlag = true;
    return;
  }
  if (cmdString.substring(0, 5).equals("setAz")) {
    globalAz = cmdString.substring(5, 8).toInt();
    cmdString = "";
    cmdFlag = false;
    position_flag = true;
    azFlag = true;
    return;

  }
  if (cmdString.substring(0, 5).equals("setEl")) {
    globalEl = cmdString.substring(5, 8).toInt();
    cmdString = "";
    cmdFlag = false;
    position_flag = true;
    elFlag = true;
    return;

  }
  if (cmdString.substring(0, 5).equals("getAz")) {
    //    int az = getAzDegrees();
    //    Serial.println(az);
    Serial.println(getAzDegrees());
    cmdString = "";
    cmdFlag = false;
    return;
  }
  if (cmdString.substring(0, 5).equals("getEl")) {
    //    int el = getElDegrees();
    //    Serial.println(el);
    Serial.println(getElDegrees());
    cmdString = "";
    cmdFlag = false;
    return;
  }
  if (cmdString.substring(0, 6).equals("getLoc")) {
    while (!Serial2.available());
    send_GSCoords();
    cmdString = "";
    cmdFlag = false;
    return;
  }
}


/*****************************************************************

   Get GS position from a GPS connected to the rotor controller shield.

 *****************************************************************/
void send_GSCoords() {
  while(validFlag == 0){
    while(uBloxEX.available(Serial2)){
      uBloxEXFix = uBloxEX.read();

      if (uBloxEXFix.valid.location) {
        lati = uBloxEXFix.latitudeL();  // Scaled by 10,000,000
        lon = uBloxEXFix.longitudeL();  // Scaled by 10,000,000
        alt = uBloxEXFix.altitude_cm();
        validFlag = 1;

        Serial.print(lati); Serial.print(',');
        Serial.print(lon); Serial.print(',');
        Serial.println(alt);
      }
    }
  }
}
```

B.3 Antenna Pointing Calibration

In order to calibrate the antenna pointing, a set of steps needs to be followed:

First of all, the antenna is aligned with the magnetic north. To do that, the tripod screws are loosen in order to be able to move the antenna towards the desired position. A compass is placed on top of the Yagi antenna in three different positions, confirming that it is pointing north. Once the antenna pointing is confirmed, the tripod screws are tighten to the antenna mast.

Once the antenna is aligned to the magnetic north, the pointing offset caused by the other sources of error, such as the GNSS sensor accuracy, is analysed. To do that, the GS test mode of the GS GUI is considered. A previously created prediction file is used to artificially add the coordinates of three different known points -i.e. a tall building that it is far away but in line of sight-. Using the 'Declination' field of the GUI, the pointing precision to the selected testing locations is adjusted, finishing the antenna pointing calibration.

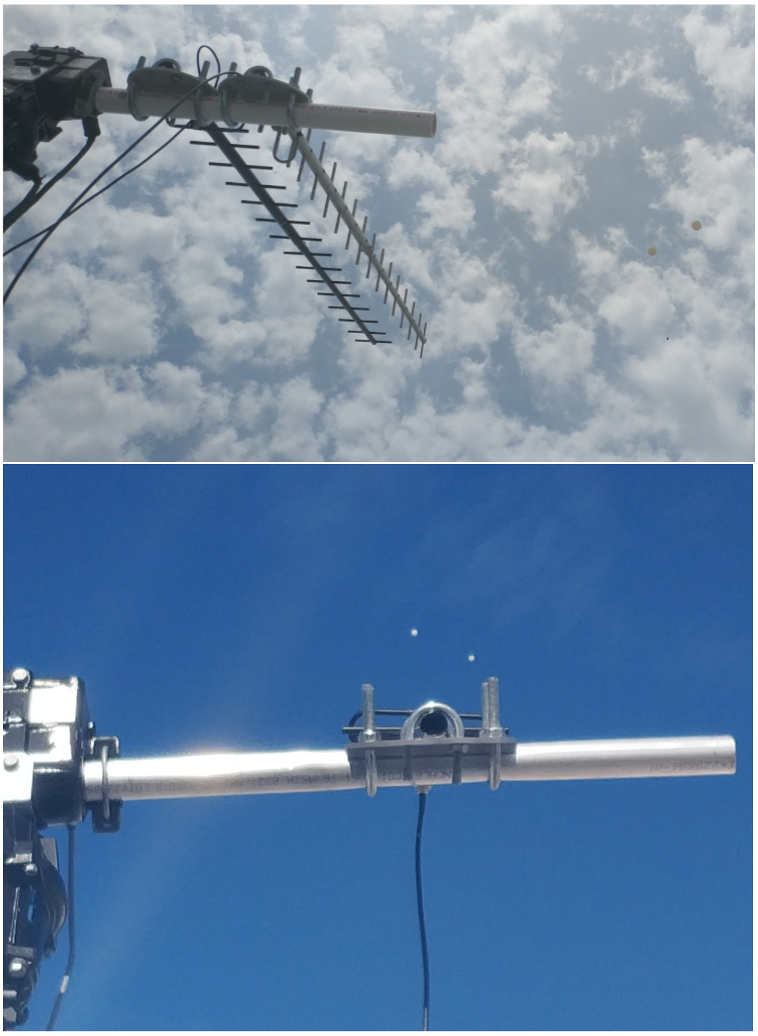


Figure B.23: GS Pointing - (A) Double Antenna Design, (B) Single Antenna Design.

Appendix C

Ground Station GUI

This appendix presents the ground station GUI design and modes definition. The different utilities of this app are explained, as well as the different steps to be considered when using one of its modes: (1) ground station check, (2) balloon launch, and (3) flight reproduction.

C.1 GUI Design Overview

As it can be seen in the next figure, the GUI used for this HAB project has several buttons, labels and graphics. In order to have an overall idea of the different parts of this app, the following specifications shall be considered:

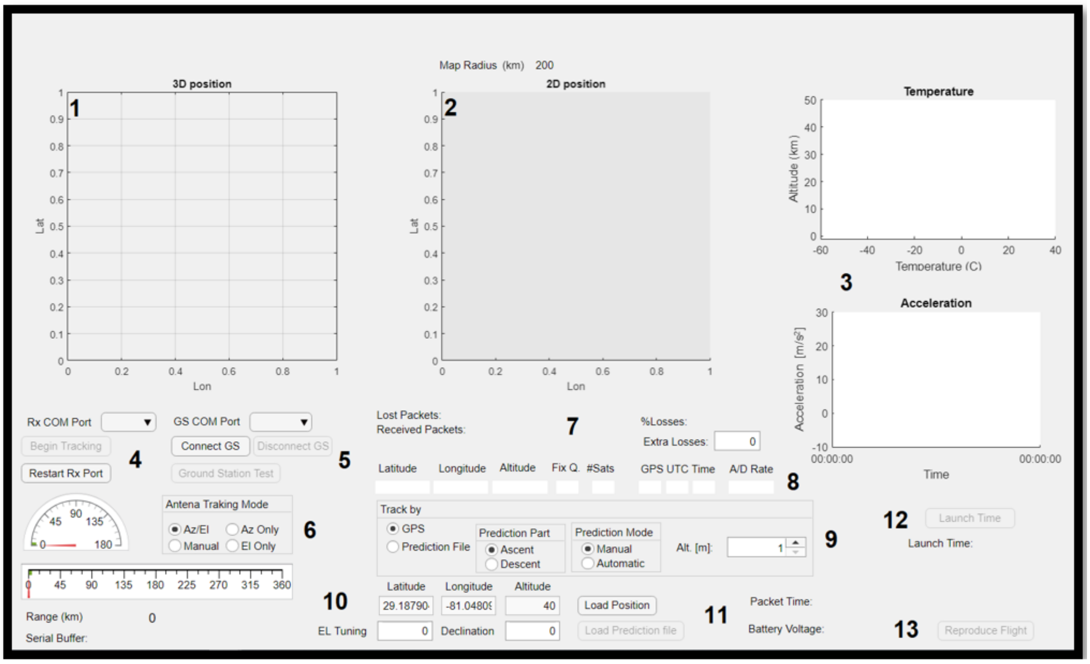


Figure C.1: GUI - Design Overview.

1. 3D Position Map.
2. 2D Position Map.
3. On-board Sensors Data Graphs.

4. Radio Communication Buttons.
5. Ground Station Buttons.
6. Antenna Tracking Mode.
7. Throughput Information.
8. Payload GPS Data.
9. Payload Tracking Mode.
10. Antenna Position/Pointing Tuning.
11. Load Position/Prediction.
12. Launch Time Info.
13. Reproduce Flight Mode.

The rest of labels and indicators are for extra information monitoring, such as payload battery voltage, the Rx serial buffer of the MATLAB application, among others.

C.2 Prediction Files

When planning a balloon launch, it is important to consider the path that it will follow in order to confirm that it can be a good launch window. The path prediction will help to see whether the payload is following the expected trajectory or not, and it will help us tracking the payload in case the GPS sensor on-board fails.

Even though the temperature and pressure sensors on board are calibrated, it is important to see which profiles they are expecting to follow during a certain launch. The temperatures and pressure predictions can be useful in order to accurately calibrate them for the expected ranges. Moreover, these predictions are used to confirm that the sensors on board are working as expected during the launch, and that the internal temperature of the payload is not affecting the functionality of any part of the hardware used for the payload development.

C.2.1 Path Prediction - CUSF Predictor

1) Go to http://habhub.org/.

2) First of all, the coordinates of the launch site shall be specified. They can be saved for next launches, if needed. After that, the burst altitude shall be specified, as well as the expected ascent rate. Please be sure to create several prediction files with different ascent rates, so in case it is lower or higher than expected, a different prediction file can be used to track the payload.

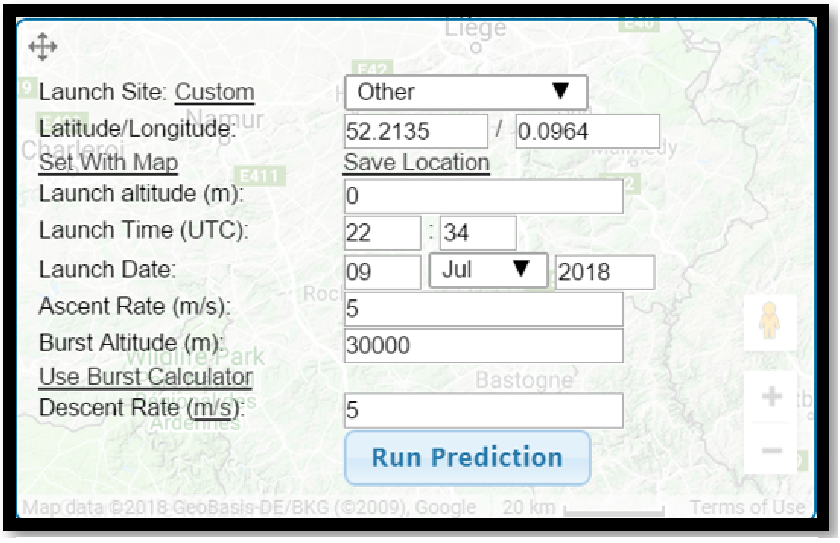


Figure C.2: CUSF Predictor - Input Parameters.

3) Once all the information is specified, "Run" the predictor and if the plotted path is correct, click the "CSV" on the right top of the screen. This will download the prediction information on a .csv format.

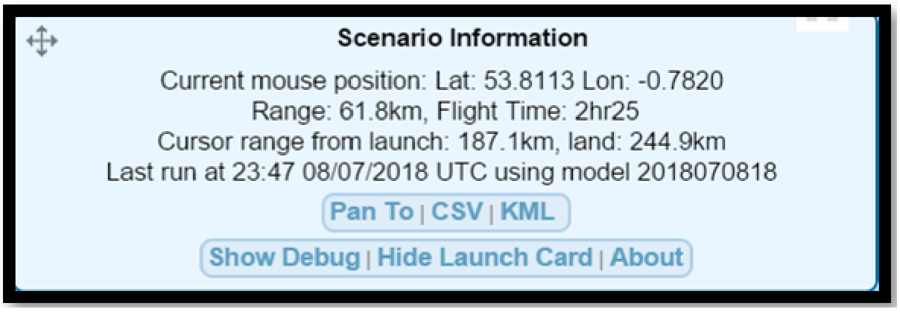


Figure C.3: CUSF Predictor - Export .csv File.

4) Open the downloaded file. This file will contain for columns: 'Time of Week', 'Latitude', 'Longitude', 'Altitude'. Another column shall be added starting at 0 and adding 50 accumulatively to the next rows. This column will represent flight time and it will be used in case a prediction file is used to track the payload automatically.

C.2.2 Measurements Prediction - Wyoming Predictor

1) Go to http://weather.uwyo.edu/upperair/balloon_traj.html.

2) First of all, the latest available time shall be selected. Note this can only be done up to 6 hours before the launch due to the model options available.

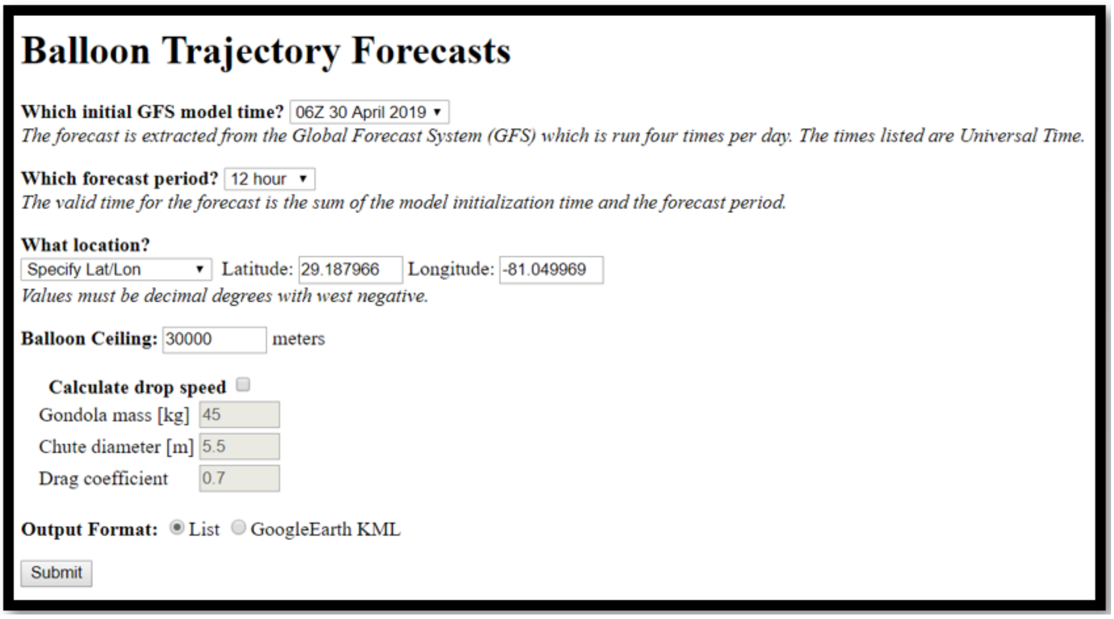
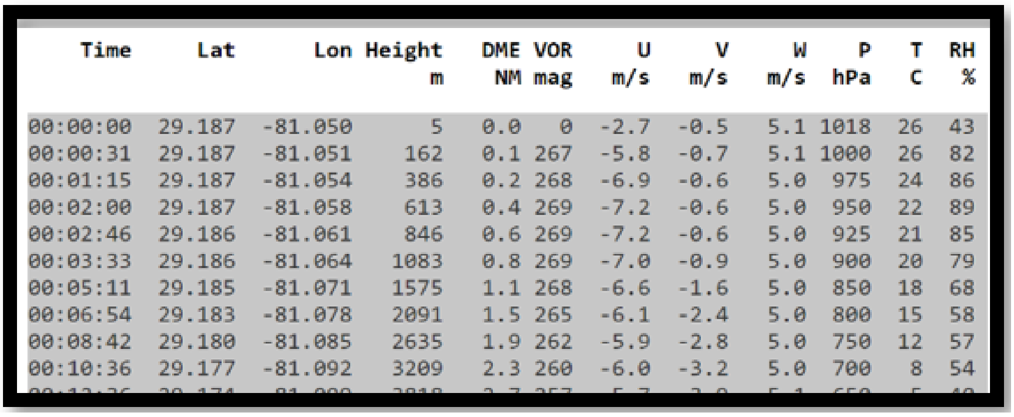


Figure C.4: Wyoming Predictor - Input Parameters.

3) Then, the launch site coordinates and the expected balloon ceiling shall be specified. The output format would be 'list'.

4) Once all the information is specified, submit it, and a list of information will appear at the bottom of the page. That information shall be copied, ignoring the headings.



Time	Lat	Lon	Height m	DME NM	VOR mag	U m/s	V m/s	W m/s	P hPa	T C	RH %
00:00:00	29.187	-81.050	5	0.0	0	-2.7	-0.5	5.1	1018	26	43
00:00:31	29.187	-81.051	162	0.1	267	-5.8	-0.7	5.1	1000	26	82
00:01:15	29.187	-81.054	386	0.2	268	-6.9	-0.6	5.0	975	24	86
00:02:00	29.187	-81.058	613	0.4	269	-7.2	-0.6	5.0	950	22	89
00:02:46	29.186	-81.061	846	0.6	269	-7.2	-0.6	5.0	925	21	85
00:03:33	29.186	-81.064	1083	0.8	269	-7.0	-0.9	5.0	900	20	79
00:05:11	29.185	-81.071	1575	1.1	268	-6.6	-1.6	5.0	850	18	68
00:06:54	29.183	-81.078	2091	1.5	265	-6.1	-2.4	5.0	800	15	58
00:08:42	29.180	-81.085	2635	1.9	262	-5.9	-2.8	5.0	750	12	57
00:10:36	29.177	-81.092	3209	2.3	260	-6.0	-3.2	5.0	700	8	54

Figure C.5: Wyoming Predictor - Output Data.

5) Open a new Excel spreadsheet and copy that information on the first column.

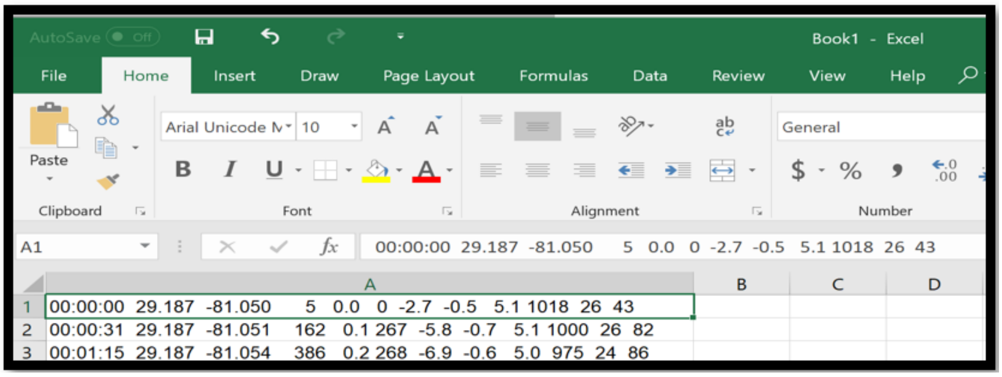


Figure C.6: Wyoming Predictor - Copied Data.

6) On the ‘Data’ tab, select ‘Text to Columns’, and select ‘Delimited’ on the next window. Click next.

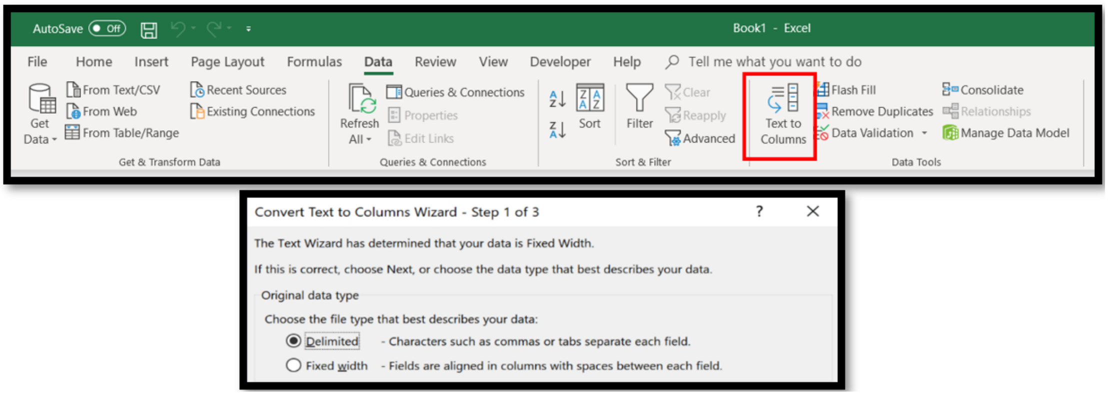


Figure C.7: Wyoming Predictor – Text to Columns and Data Delimited.

7) Uncheck ‘Tab’ and select ‘Space’. Click next.

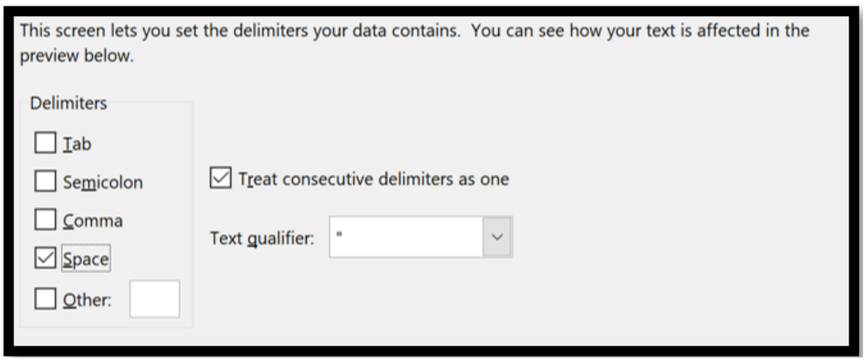


Figure C.8: Wyoming Predictor – Data Space Delimited.

8) Leave the data type as ‘General’ and select finish.

9) Save the file as a .csv.

C.3 GUI Modes

C.3.1 GUI Setup

The following steps must be followed for all the GUI modes in order to prepare the GUI environment.

First, the map/ground station position will be specified. For that, "Load Position" will be pushed and a small window will appear asking if we want to get the GS coordinates in real-time or not. Please consider that the ground station position can be hard coded on the design view, if this position is expected to be always the same. However, if a GPS sensor is connected to the GS controller shield, these coordinates can be computed and uploaded to the GUI by clicking yes to that first window. In this case, the GS needs to be connected beforehand.

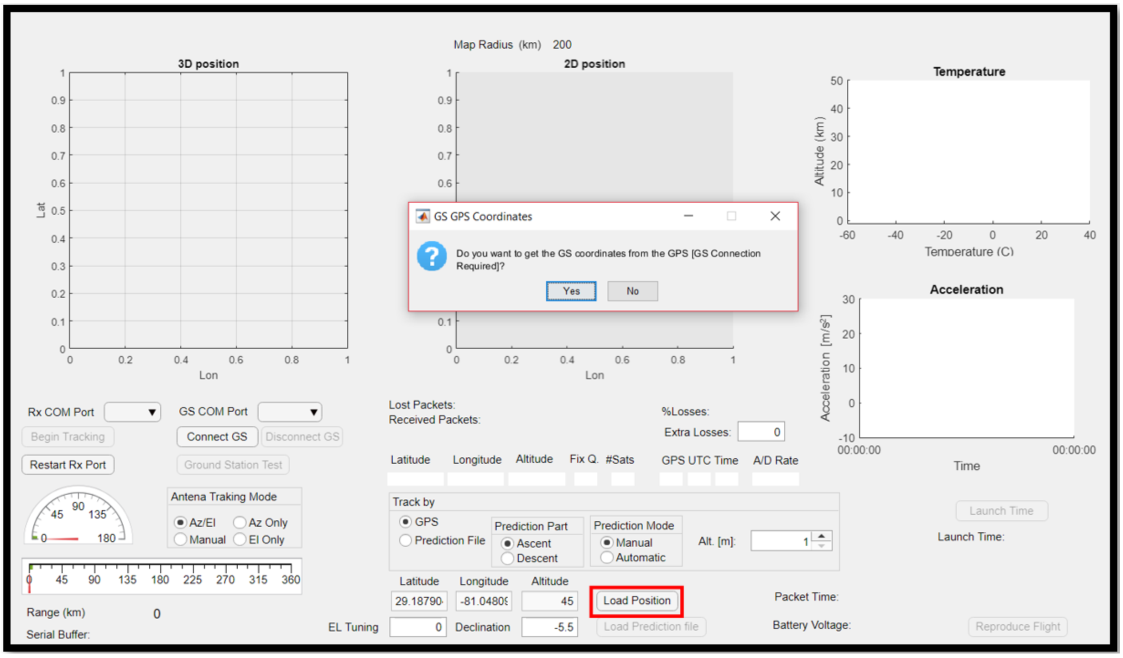


Figure C.9: GUI - Load Position.

The computed coordinates will appear on the next window. In this window, the coordinates can be changed by hand if required. Moreover, the desired map radius will need to be introduced, considering the predicted path data.

For the Map Radius, the expected maximum range for that launch shall be taken into account to achieve the proper map resolution while tracking the balloon path.

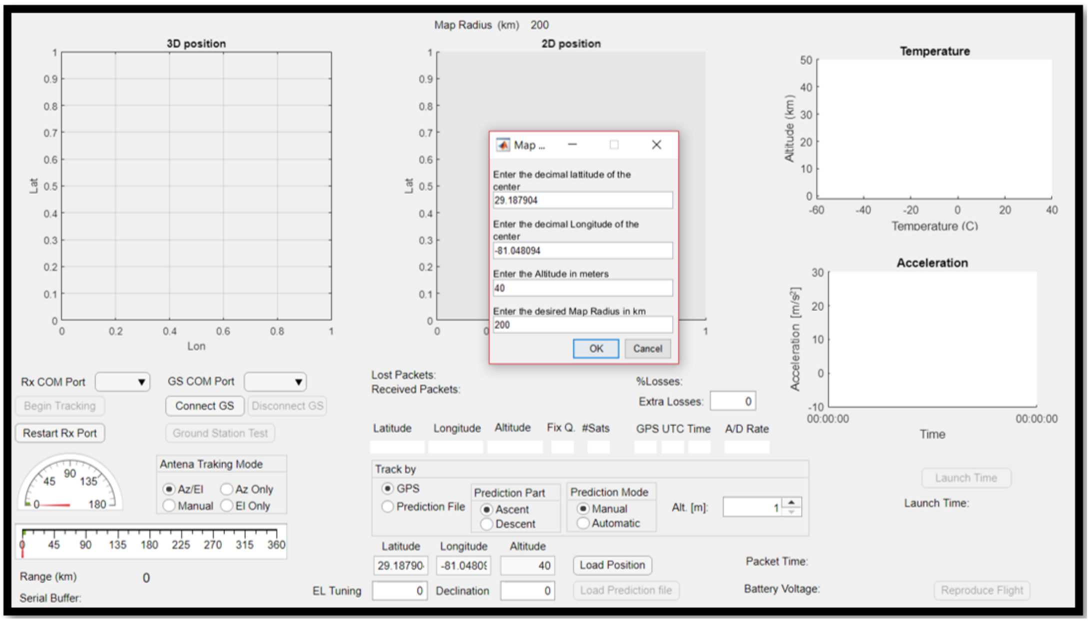


Figure C.10: GUI - GS Coordinates, Map Position.

After that, the GUI will ask us if the antenna position is the same as the map center, and if we want to upload a Map or download a new one. For the last option, an internet connection is required.

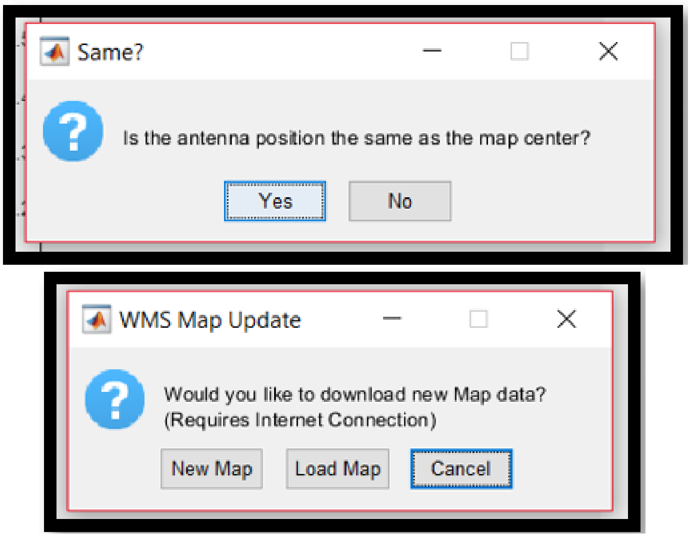


Figure C.11: GUI - Antenna Location and Map Setup.

At this point, a flight data file can be reproduced, but it is more convenient to fully prepared the GUI to contain the prediction files to be able to visualize all the information available during that flight.

For the ground station checks and the flight modes, the prediction files will be needed. So next step will be to load the prediction files that were created previously.

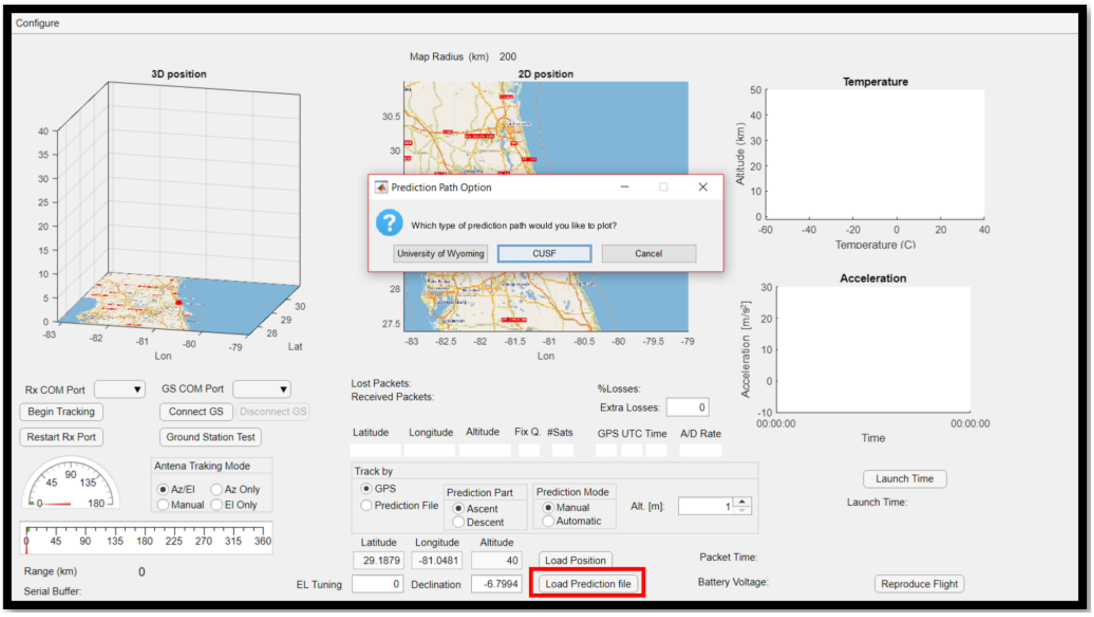


Figure C.12: GUI - Load Prediction Files.

Please take into account that each prediction file it is used for different purposes, so the GUI is expecting one format or the other, according to the pushed button.

After that, the GUI is completely prepared for a launch or a ground station check.

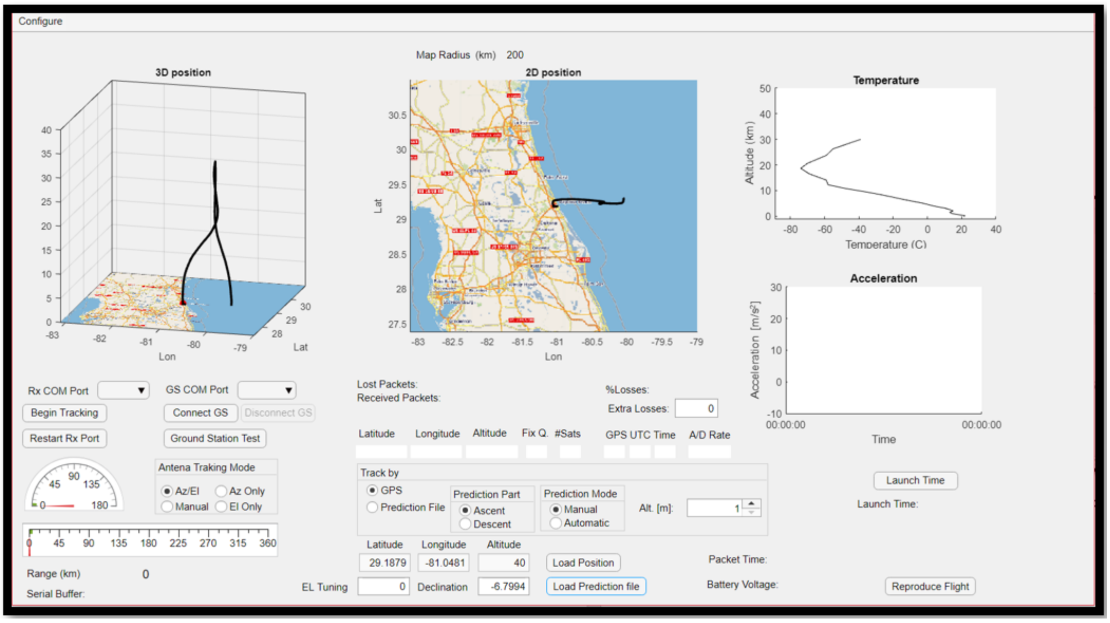


Figure C.13: GUI - Loaded Prediction Files.

C.3.2 Reproduce Flight

For this mode, only the data file from a previous flight is required.

The GUI code can be changed to allow more or less time between data points being plotted.

```
min_gps = 10*gpsRate;
min_sci = 10*125;
```

Figure C.14: Minimum Messages to be Considering between Data plotted.

A window to choose which .bin file is going to be reproduced will appear after pushing 'Reproduce Flight'.

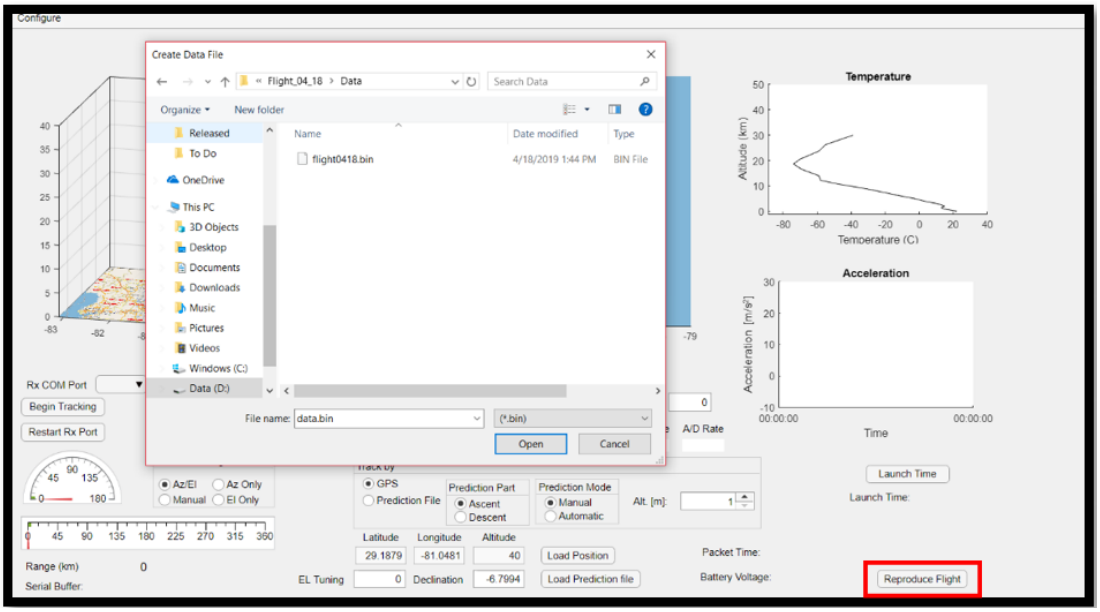


Figure C.15: GUI - Reproduce Flight Selection.

After a few seconds, the data will start being plotted automatically.

C.3.3 Ground Station Check

For this mode, there are several things to consider.

First of all, the GUI needs to be connected to the Ground Station controller. To do that, select the proper 'GS COM port' drop down list/button and push the 'Connect GS' button. After a few seconds, the button 'Disconnect GS' will be activated, meaning that the GUI and the GS controller connection was successfully made.

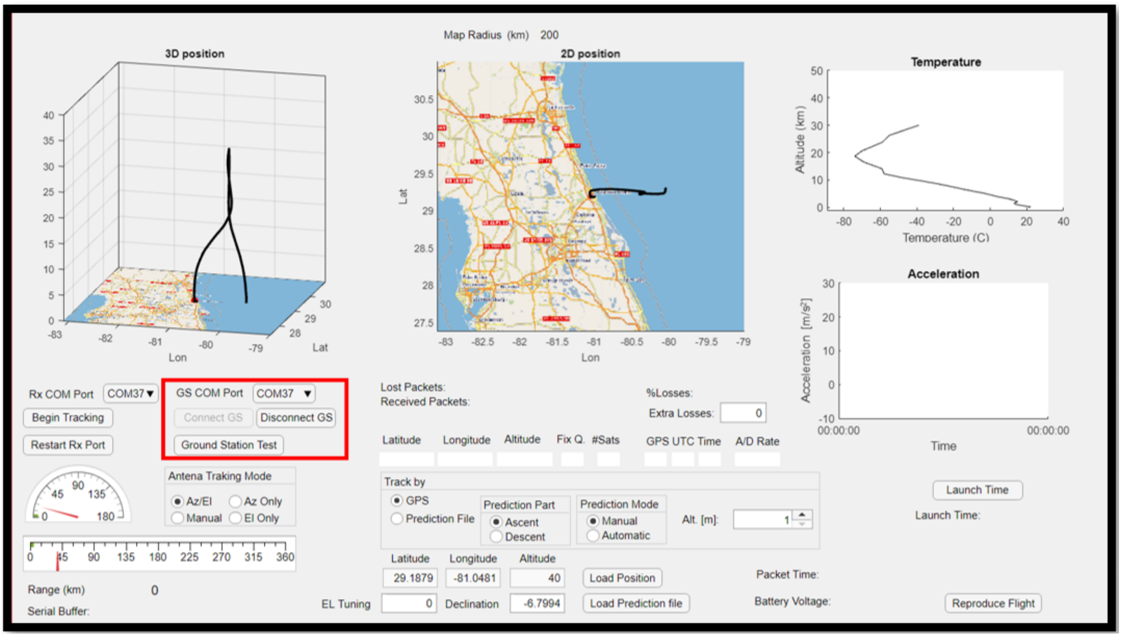


Figure C.16: Ground Station Check - GS Connection.

Once the GS connection is completed, the 'Ground Station Test' button can be pushed. The tracking mode shall be changed to 'Prediction File'. The 'Prediction Mode' will define if the ascent or the descent part of the launch is going to be checked. Finally, the 'Prediction Mode' will define if the GS check is going to be performed manually of automatically.

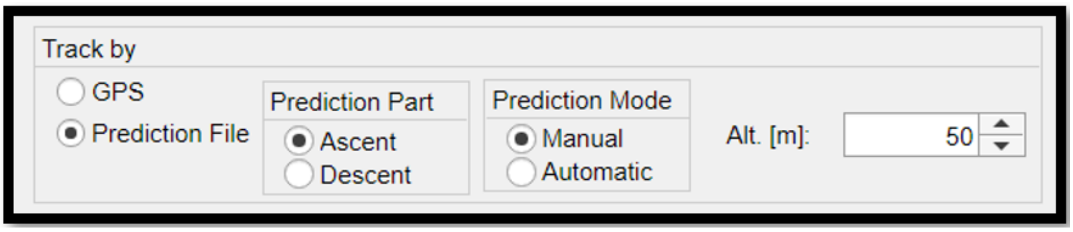


Figure C.17: Ground Station Check - Prediction File Tracking Options.

If the prediction mode is manual, the altitude input label on the right side must be changed accordingly. If the automatic mode is selected, the GS coordinates will be automatically updated from the last altitude input until the end of the launch predicted data.

By selecting 'Manual' and pushing 'Ground Station Test' again, the prediction mode can be changed again.

The 3D and 2D position maps will show the corresponding data points during the GS checks, and the GPS labels will show the predicted balloon LLA coordinates:

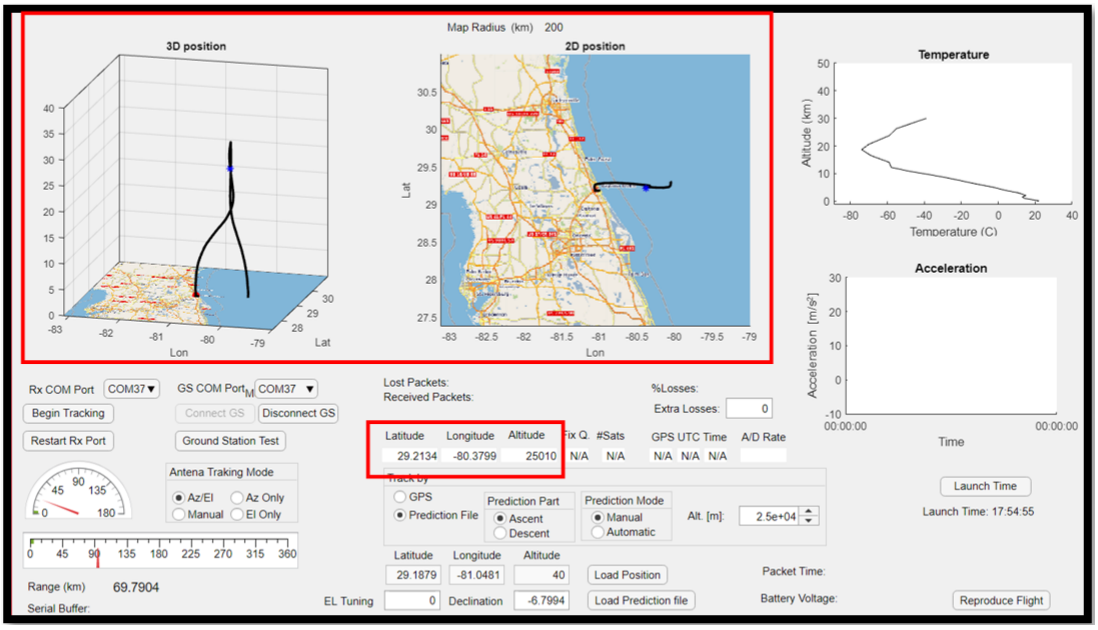


Figure C.18: Ground Station Check - Predicted Position Plotting.

C.3.4 HAB Launch

The first thing to do in this mode is the Radio connection. To do that, the 'Rx COM Port' must be used to select the GS radio port before pushing the 'Begin Tracking' button.

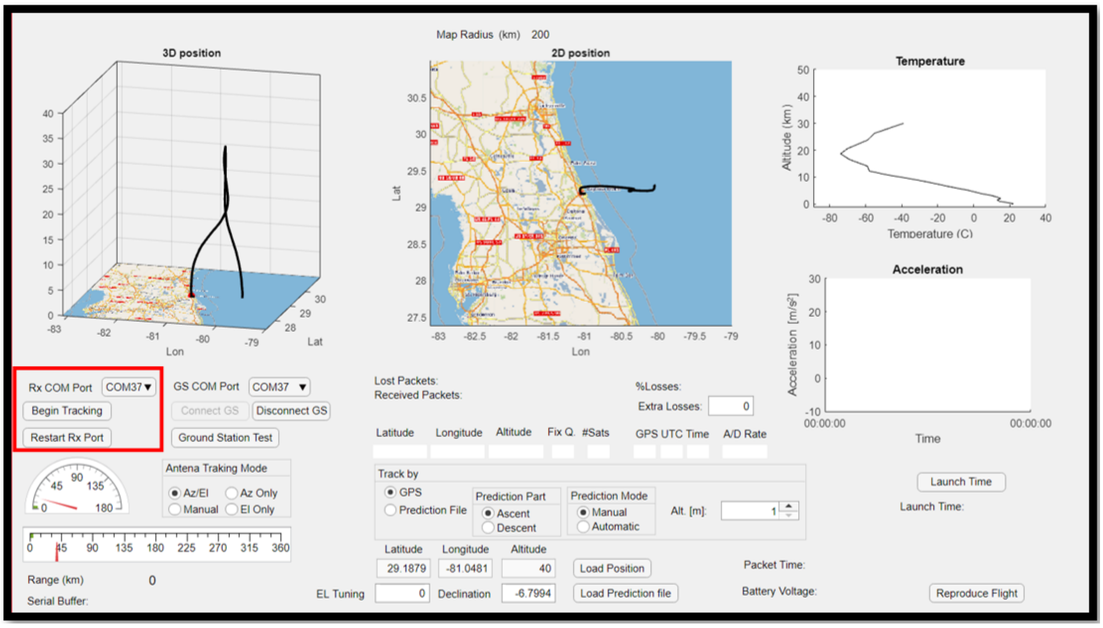


Figure C.19: HAB Launch Mode - GS Radio Connection.

Once the payload is launched, the 'Launch Time' button can be pushed in order to keep track of the exact launch time. It can be useful, if some timer is included on our cutting system design.

The 'Restart Rx Port' button can be pushed if the serial connection with the radio fails, in order to restart it.

During a launch, the antenna tracking mode can be changed to point the ground station only considering elevation angles, azimuth angles, both of them and none of them for a manually pointing.

The payload tracking mode can be changed as well from using the on-board GPS coordinates to the prediction file information.

The ground station antenna is aligned to the magnetic North using a compass during the antenna setup. As the magnetic north is different from true north, there needs to be a declination correction. Furthermore, no matter how good a compass is, there are always local stray fields that will affect the compass. The magnetic alignment will be off by a few degrees in addition to declination. This is where the tuning fields come in. By editing the "Declination" field, an azimuth correction can be applied so that the antenna points exactly at the payload.

Similarly, the elevation can also end up having a few degrees of offset. The "EL Tuning" field is included so that it can be altered to correct the pointing offsets.

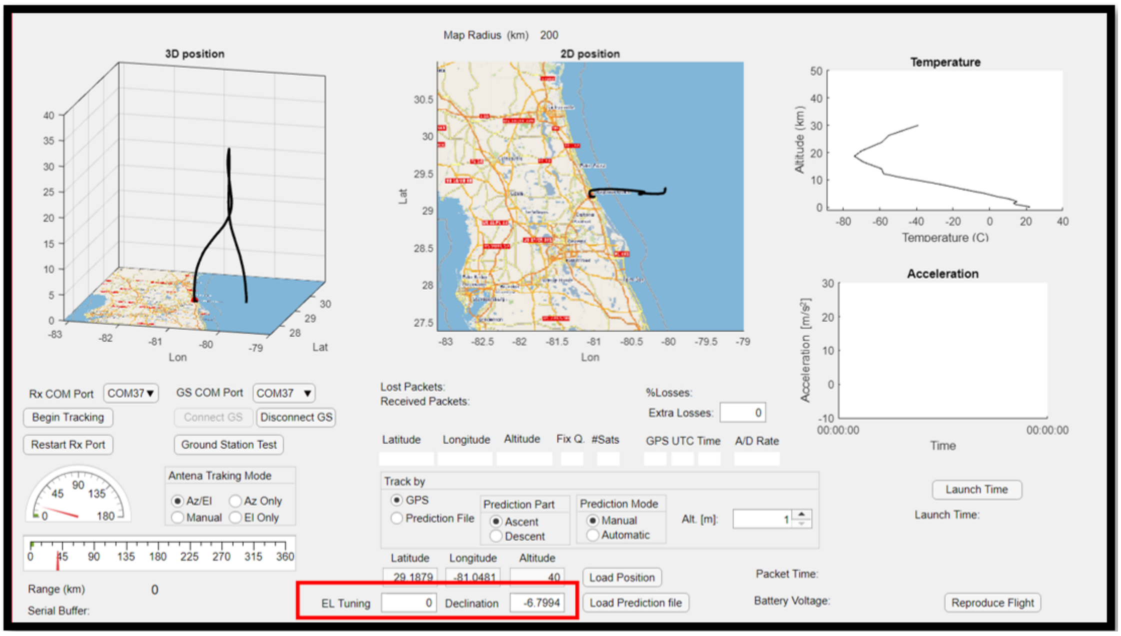


Figure C.20: MATLAB GUI - Az/El tuning fields.

Once the final Az/El coordinates to point the antenna to are computed, the GUI will update the Az/El indicators. They are only indicative of what the calculated Az/El are, based on the received payload GPS location and the tuning fields. These are the Az/El values sent to the Arduino shield that then controls the rotor controller. The GUI does not show what the rotor is set to. This GUI's intention is to help the user by showing visually what is the calculated Az/El, and then the user can visually check if the rotor is actually pointing there by looking at the rotor dials. Therefore, if the GS mode is set to manual, the GUI will not show where the rotor is at.

Currently, the actual Az/El from the rotor is only read by the Arduino shield, to determine how much it is required to be moved to point towards the expected Az/El coordinates. The GUI is blind to the actual rotor position.

Additional communication between this GUI and the rotor controller can be added to be able to plot the actual rotor position even in manual mode.

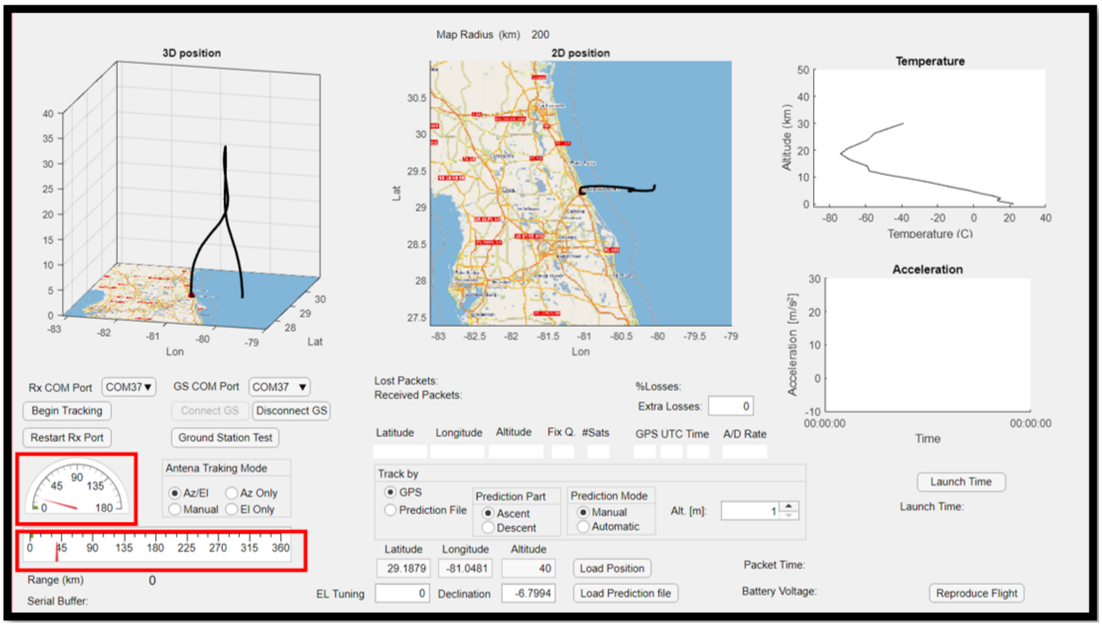


Figure C.21: MATLAB GUI - GS Az/El indicators.

While tuning the antenna pointing during a flight or when the antenna azimuth value changes from 360 degrees to 0, a lot of packet losses can be experienced. If these, or other possible, extra packet losses are desired to be subtracted from the actual packet losses value in order to not considering then when computing the total packet losses percentage, the 'Extra Losses' label can be used. The number specified in that field will be subtracted from the "Lost Packets" field.

Finally, the 'Serial Buffer' label will present the status of the MATLAB Rx serial buffer when the chunks of data are selected to be processed. This value should be similar to the hardcoded number of available bytes that it is specified in the GUI code:

```
%Read and Process Data
while(true)
    if(app.s.BytesAvailable>13000)

        app.SerialBufferLabel.Text = ['Serial Buffer: ',num2str(app.s.BytesAvailable)];
```

Figure C.22: HAB Launch - Rx Serial Buffer Monitor.

If MATLAB is not able to handle the amount of received data, while decoding the sensors information and plotting them, it can be possible to experience a buffer overflow. This problem can be detected with the 'Serial Buffer' label.

C.3.5 GUI Code

```
classdef MURI_HAB_GUI_v14PL2_Mobile_GS < matlab.apps.AppBase

    % Properties that correspond to app components
    properties (Access = public)
        UIFigure                       matlab.ui.Figure
        ConfigureMenu                  matlab.ui.container.Menu
        RefreshCOMPortsMenu            matlab.ui.container.Menu
        LoadMapsMenu                   matlab.ui.container.Menu
        LoadPredictionFileMenu         matlab.ui.container.Menu
        Location3D                     matlab.ui.control.UIAxes
        Location2D                     matlab.ui.control.UIAxes
        LoadPositionButton             matlab.ui.control.Button
        AltitudeEditFieldLabel         matlab.ui.control.Label
        GS_Altitude                    matlab.ui.control.EditField
        LongitudeEditFieldLabel        matlab.ui.control.Label
        GS_Longitude                   matlab.ui.control.EditField
        LatitudeEditFieldLabel         matlab.ui.control.Label
        GS_Latitude                    matlab.ui.control.EditField
        BeginTrackingButton            matlab.ui.control.Button
        ElevationGauge                 matlab.ui.control.SemicircularGauge
        AzGauge                        matlab.ui.control.LinearGauge
        TempInt                        matlab.ui.control.UIAxes
        sltRange                       matlab.ui.control.Label
        RangekmLabel                   matlab.ui.control.Label
        ConnectGSButton                matlab.ui.control.Button
        ConnectingLabel                matlab.ui.control.Label
        DisconnectGSButton             matlab.ui.control.Button
        DeclinationEditFieldLabel      matlab.ui.control.Label
        DeclinationEditField           matlab.ui.control.EditField
        LostPackets                    matlab.ui.control.Label
        ReceivedPackets                matlab.ui.control.Label
        Voltage                        matlab.ui.control.UIAxes
        GSCOMPortDropDownLabel         matlab.ui.control.Label
        GSCOMPortDropDown              matlab.ui.control.DropDown
        RxCOMPortDropDownLabel         matlab.ui.control.Label
        RxCOMPortDropDown              matlab.ui.control.DropDown
        SerialBufferLabel              matlab.ui.control.Label
        LoadPredictionfileButton       matlab.ui.control.Button
        TrackbyButtonGroup             matlab.ui.container.ButtonGroup
        GPSButton                      matlab.ui.control.RadioButton
        PredictionFileButton           matlab.ui.control.RadioButton
        PredictionFileButtonGroup      matlab.ui.container.ButtonGroup
        AscentButton                   matlab.ui.control.RadioButton
        DescentButton                  matlab.ui.control.RadioButton
        PredictionMode                 matlab.ui.container.ButtonGroup
        ManualButton2                  matlab.ui.control.RadioButton
        AutomaticButton                matlab.ui.control.RadioButton
        AltmSpinnerLabel               matlab.ui.control.Label
        AltmSpinner                    matlab.ui.control.Spinner
        MapRadiusLabel                 matlab.ui.control.Label
        radius                         matlab.ui.control.Label
        kmLabel                        matlab.ui.control.Label
        BatteryVoltageLabel            matlab.ui.control.Label
        gpsLAT                         matlab.ui.control.Label
        gpsLONG                        matlab.ui.control.Label
        gpsALT                         matlab.ui.control.Label
        gpsFIX                         matlab.ui.control.Label
        LatitudeLabel                  matlab.ui.control.Label
        LongitudeLabel                 matlab.ui.control.Label
        AltitudeLabel                  matlab.ui.control.Label
        FixQLabel                      matlab.ui.control.Label
        gpsSATS                        matlab.ui.control.Label
        gpsHOUR                        matlab.ui.control.Label
        gpsMIN                         matlab.ui.control.Label
        gpsSEC                         matlab.ui.control.Label
        SatsLabel                      matlab.ui.control.Label
        GPSUTCTimeLabel                matlab.ui.control.Label
        ELTuningEditFieldLabel         matlab.ui.control.Label
        ELTuningEditField              matlab.ui.control.EditField
        AscentRate                     matlab.ui.control.Label
        ADRateLabel                    matlab.ui.control.Label
        LossesLabel                    matlab.ui.control.Label
        AntenaTrakingModeButtonGroup_2 matlab.ui.container.ButtonGroup
        AzElButton                     matlab.ui.control.RadioButton
        AzOnlyButton                   matlab.ui.control.RadioButton
        ElOnlyButton                   matlab.ui.control.RadioButton
        ManualButton                   matlab.ui.control.RadioButton
        GroundStationTestButton        matlab.ui.control.Button
        ReproduceFlightButton          matlab.ui.control.Button
        LaunchTimeButton               matlab.ui.control.Button
        LaunchTimeLabel                matlab.ui.control.Label
        PacketTimeLabel                matlab.ui.control.Label
        RestartRxPortButton            matlab.ui.control.Button
```

```
        ExtraLossesEditFieldLabel       matlab.ui.control.Label
        ExtraLossesEditField            matlab.ui.control.NumericEditField
    end

    properties (Access = private)
        xdata % For plotting x data
        ydata % For plotting y data
        zdata % For plotting z data
        wdata % For plotting w data
        flag = 0;
        flag_GS = 0;
        hC = 0;
        maxC =0;
        myGSCoord;
        WYpredicted;
        spheroid = referenceEllipsoid('WGS 84');
        CUSFpredicted;

        startTime = 0;
        newMapFlag = 0;
        predFileFlag = 0;
        predFileCol = ['k', 'r', 'b', 'g', 'c'];

    end

    properties (Access = public)
        dataFile;
        gpsData;
        scientificData;
        serial_GS;
        A; B;
        s;

    end

    methods (Access = private)

        function [latlim, lonlim] = getMapLimits(app,lat0,lon0,h0)
            if lat0 <= 90 && lat0 >= -90 && lon0 <= 180 && lon0 >= -180 && isnumeric(h0) && isreal(h0)
                az = [0 90 180 270];
                slantRange = str2double(app.radius.Text)*1000;
                elev = 0;
                lat = [0 0 0 0];
                lon = lat;
                h=lat;
                for f = 1:4
                    [lat(f),lon(f),h(f)] = aer2geodetic(az(f),elev,slantRange,lat0,lon0,h0,app.spheroid);
                end
                latlim = [lat(3) lat(1)];
                lonlim = [lon(4) lon(2)];
            else
                errordlg('Check your position and try again');
            end
        end

        function ZA = loadMaps(app,latlim,lonlim)
            ZA=[];
            prompt = {'Would you like to download new Map data?'; '(Requires Internet Connection)'};
            title = 'WMS Map Update';
            answ = questdlg(prompt,title,'New Map','Load Map','Cancel','Cancel');
            switch answ
                case 'New Map'
                    numberOfAttempts = 5;
                    attempt = 0;
                    info = [];
                    mundalisServer = 'http://ows.mundialis.de/services/service?';
                    OSM_WMS_Uni_Heidelberg = 'http://129.206.228.72/cached/osm?';

                    serv2 = 0;
                    while(isempty(info))
                        try
                            if serv2 == 0
                                info = wmsinfo(mundalisServer);
                                orthoLayer = info.Layer(2);
                            elseif serv2 == 1
                                info = wmsinfo(OSM_WMS_Uni_Heidelberg);
                                orthoLayer = info.Layer(2);
                            end
                        catch

                            attempt = attempt + 1;
                            if attempt > numberOfAttempts && serv2 == 0
                                warning('Server 1 is not available. Trying Server 2');
                                serv2 = 1;
                                attempt = 0;
                            end
                        end
                        if serv2 == 1 && attempt > numberOfAttempts
                            warndlg ({'WMS servers are not available.';'Please load an existing Map'});
                            return
                        end
                    end
                    [ZA, ~] = wmsread(orthoLayer, 'Latlim', latlim, 'Lonlim', lonlim, ...
                        'ImageFormat', 'image/png');
                    app.newMapFlag = 1;
                case 'Load Map'
                    [newfile,path] = uigetfile('*.map','Load Map File','map1.map');
                    figure(app.UIFigure);
                    if newfile == 0
```

```
                        return;
                    end
                    filename=fullfile(path,newfile);
                    load(filename,'ZA','-mat');
                    app.newMapFlag = 0;
                case 'Cancel'

                    return
            end

        end

        function results = DrawMaps(app,ZA,latlim,lonlim)
            results = 0;
            imagesc(app.Location2D,lonlim,latlim,flipud(ZA));
            imagesc(app.Location3D,lonlim,latlim,flipud(ZA));

            %                  demcmap(double(ZA))
            %                          lat=linspace(latlim(2),latlim(1),size(ZA,1));
            %                          lon=linspace(lonlim(1),lonlim(2),size(ZA,2));
            %
            %                          pcolor(app.Location3D,lon,lat,ZA);
            %                          pcolor(app.Location2D,lon,lat,ZA);
            %             app.Location2D.DataAspectRatio= [abs(diff(lonlim)),abs(diff(latlim)),1];
            %                          [cmap,~] = demcmap(ZA);
            %                          colormap(app.Location3D,cmap);
            %                          shading(app.Location3D, 'interp');
            %                          colormap(app.Location2D,cmap);
            %                          shading(app.Location2D, 'interp');
            xlim(app.Location2D,lonlim)
            ylim(app.Location2D,latlim)
            xlim(app.Location3D,lonlim)
            ylim(app.Location3D,latlim)
            view(app.Location3D,15,15)
            zlim(app.Location3D,[-.001,40])
            if app.newMapFlag == 1
                q2 = questdlg('Would you like to save the map data?','Save?','Yes','No','Yes');
                switch q2
                    case 'Yes'
                        [newfile,path] = uiputfile('*.map','Create Data File','map1.map');
                        if newfile == 0
                            return;
                        end
                        filename=fullfile(path,newfile);

                        save( filename, 'ZA');

                    case 'No'
                end
            end
            results = 1;

        end

    end


    methods (Access = private)

        % Code that executes after component creation
        function startupFcn(app)
            app.RxCOMPortDropDown.Items = cellstr(seriallist);
            app.GSCOMPortDropDown.Items = app.RxCOMPortDropDown.Items;
            hold(app.Voltage, 'on');
            hold(app.TempInt, 'on');
            datetick(app.Voltage,'x', 'HH:MM:SS');
            hold(app.Location3D,'on');
            hold(app.Location2D,'on');
            app.myGSCoord = [str2double(app.GS_Latitude.Value), ...
                str2double(app.GS_Longitude.Value),str2double(app.GS_Altitude.Value)];
            app.UIFigure.Position = [0 0 1280 700];
            app.UIFigure.WindowState = 'maximized';

        end

        % Callback function: LoadMapsMenu, LoadPositionButton
        function LoadPositionButtonPushed(app, event)
            q1 = questdlg('Do you want to get the GS coordinates from the GPS [GS Connection Required]?','GS GPS Coordinates','Yes','No','Yes');
            switch q1
                case 'Yes'
                  fprintf(app.serial_GS,'%s\n','getLoc');
                  while (app.serial_GS.BytesAvailable == 0)
                  end

                  [coords] = fscanf(app.serial_GS,'%d,%d,%d\n');

                  app.GS_Latitude.Value = num2str(coords(1)/10000000);
                  app.GS_Longitude.Value = num2str(coords(2)/10000000);
                  app.GS_Altitude.Value = num2str(coords(3)/100);

            end
            prompt = {'Enter the decimal latitude of the center', 'Enter the decimal Longitude of the center', ...
                'Enter the Altitude in meters', 'Enter the desired Map Radius in km'};
            title = 'Map Configuration';
            dims = [1,35];
            default = {app.GS_Latitude.Value, app.GS_Longitude.Value, app.GS_Altitude.Value, app.radius.Text};
            answer = inputdlg(prompt,title,dims,default);
```

```
            if ~isempty(answer) && isreal(str2double(answer))
                mapZ = str2num(answer{3});
                mapLat = str2num(answer{1});
                mapLon = str2num(answer{2});
                app.radius.Text = answer{4};
            else
                errordlg('Check the center position and try again');
                return
            end

            q2 = questdlg('Is the antenna position the same as the map center?','Antenna Position','Yes','No','Yes');
            switch q2
                case 'Yes'
                    h0 = mapZ;
                    lat0 = mapLat;
                    lon0 = mapLon;
                    app.GS_Latitude.Value = num2str(lat0);
                    app.GS_Longitude.Value = num2str(lon0);
                    app.GS_Altitude.Value = num2str(h0);
                case 'No'
                    prompt = {'Enter the decimal latitude', 'Enter the decimal Longitude', ...
                        'Enter the Altitude in meters'};
                    title = 'Antenna Position';
                    dims = [1,35];
                    default = {app.GS_Latitude.Value, app.GS_Longitude.Value, app.GS_Altitude.Value, app.radius.Text};
                    answer = inputdlg(prompt,title,dims,default);
                    if ~isempty(answer) && isreal(str2double(answer))
                        h0 = str2num(answer{3});
                        lat0 = str2num(answer{1});
                        lon0 = str2num(answer{2});
                        app.GS_Latitude.Value = answer{1};
                        app.GS_Longitude.Value = answer{2};
                        app.GS_Altitude.Value = answer{3};
                        app.radius.Text = answer{4};
                    else
                        errordlg('Check your position and try again');
                        return
                    end
            end

            [latlim, lonlim] = getMapLimits(app,mapLat,mapLon,mapZ);

            ZA = loadMaps(app,latlim,lonlim);
            success =0;
            if ~isempty(ZA)
                success = DrawMaps(app,ZA,latlim,lonlim);
            end
            if success == 1
                [y,m,d,~,~,~]=datevec(datetime('now'));
                dec=decyear(y,m,d);
                [~,~,declination,~,~] = wrldmagm(h0,lat0,lon0,dec);
            app.DeclinationEditField.Value=string(declination);

            plot3(app.Location3D, lon0, lat0, h0/1000, 'r*', 'LineWidth',2)
                plot(app.Location2D, lon0, lat0, 'r*')
                app.LoadPredictionfileButton.Enable = 'on';
                app.LoadPredictionFileMenu.Enable = 'on';
                app.ReproduceFlightButton.Enable='on';
                app.LaunchTimeButton.Enable='on';
                app.BeginTrackingButton.Enable='on';
                app.GroundStationTestButton.Enable = 'on';

            end
            figure(app.UIFigure);

        end

% Button pushed function: BeginTrackingButton
function BeginTrackingButtonPushed(app, event)

    [newfile,path] = uiputfile('*.bin','Create Data File','data.bin');
    figure(app.UIFigure);
    if newfile == 0
        return;
    end
    filename=fullfile(path,newfile);
    app.dataFile=fopen(filename,'w+');


     lat0=app.myGSCoord(1);
     lon0=app.myGSCoord(2);
     h0=app.myGSCoord(3);


    %Serial for the radio communication or file
    app.s = serial(app.RxCOMPortDropDown.Value);


    %Set serial parameters
    app.s.InputBufferSize = 15000000;
    set(app.s, 'DataBits', 8);
    set(app.s, 'StopBits', 1);
    set(app.s, 'BaudRate', 230400);
    set(app.s, 'Parity', 'none');

    %Open the serial port
    try
```

```
        fopen(app.s);
    catch err
        fclose(app.s);
        warndlg('Make sure you select the correct Radio COM Port.');
    end

    id_scient=[160,177]';
    id_gps=[192,209]';
    binary_file = app.dataFile;
    messages=zeros(1,100);
    rcvd_packets = 0;
    packets_sci = 0;
    packets_gps = 0;
    min_gps = 1;
    lost_packets = 0;
    lost_total = 0;
    packet_num = 0;
    new_packet_number = 0;
    range = 0;
    timer_1 = tic;
    timer_3 = tic;
    timer_5 = tic;


    %External High Thermistor Coefficients:
    p1_ex = 0.1522;
    p2_ex = 0.8645;
    p3_ex = 0.7656;
    p4_ex = 12.9;
    p5_ex = -6.172;
    mean_ex = 533.5;
    std_ex = 179.3;

    %Extra External Low Thermistor Coefficients:
    p1_ex2 = 0.5933;
    p2_ex2 = 1.197;
    p3_ex2 = 0.4364;
    p4_ex2 = 11.58;
    p5_ex2 = -48.05;
    mean_ex2 = 587.5;
    std_ex2 = 211.6;

    %Internal Thermistor Coefficients:
    p1_in = -0.4915;
    p2_in = -1.88;
    p3_in = -2.712;
    p4_in = -16.71;
    p5_in = 15.28;

    mean_in = 770.3;
    std_in = 152.1;


    %Voltage ADC Coefficients:
    p1_v = -0.003031;
    p2_v = 1.093;
    p3_v = 1.661;
    mean_v = 510.4;
    std_v = 338;


    %Initial Position
    lat = lat0;
    lon = lon0;
    h = h0;

    %Initial time and threshold of the timer (time between GS checks)
    timerIni = tic;
    timeThreshold = 5;


    %Ascent Rate Monitor Variables
    prevAlt = 35;
    prevTime = 0;

    ascentRate = 0;

    %Read and Process Data
    while(true)
        if(app.s.BytesAvailable>13000)

            app.SerialBufferLabel.Text = ['Serial Buffer: ',num2str(app.s.BytesAvailable)];

            %Save data with timestamp
            read_Byte = fread(app.s,13000);

            fwrite(binary_file, read_Byte);

            for i=1:(length(read_Byte)-102)
                %Loof for the start of a scientific or GPS packet.
                if((read_Byte(i:i+1)==id_scient)|(read_Byte(i:i+1)==id_gps))
                    %Check if the packet has been completely received.
                   if ((read_Byte(i+100:i+101)==id_scient)|(read_Byte(i+100:i+101)==id_gps))
                        rcvd_packets = rcvd_packets+1;
                        messages(1:100)=read_Byte(i:i+99);

                        %Check if it is a Scientific Packet and parse it
                         if (read_Byte(i:i+1)==id_scient(1:end))
                            packet_num = typecast(uint8(messages(3:4)),'uint16');
```

```
    packet_time = typecast(uint8(messages(27:30)),'uint32');
    packet_time = double(packet_time)/1000;

    packets_sci = packets_sci+1;
    timer_2 = toc(timer_1);
    if ((packets_sci==190)||(timer_2>5))
        packets_sci=0;
        timer_1 = tic;
        %External Temperature Conversion
        temp = typecast(uint8(messages(5:6)),'uint16');
        temp = double(temp);
        temp = (temp-mean_ex)/std_ex;
        temp_ex = p1_ex*temp^4 + p2_ex*temp^3 + p3_ex*temp^2 + p4_ex*temp + p5_ex;


        %Internal Temperature Conversion
        temp = typecast(uint8(messages(7:8)),'uint16');
        temp = double(temp);
        temp = (temp-mean_in)/std_in;
        temp_in = p1_in*temp^4 + p2_in*temp^3 + p3_in*temp^2 + p4_in*temp + p5_in;


        %Extra External Temperature Conversion
        temp = typecast(uint8(messages(9:10)),'uint16');
        temp = double(temp);
        temp = (temp-mean_ex2)/std_ex2;
        temp_ex2 = p1_ex2*temp^4 + p2_ex2*temp^3 + p3_ex2*temp^2 + p4_ex2*temp + p5_ex2
              ;

        %Voltage Monitor
        voltage = typecast(uint8(messages(13:14)),'uint16');
        voltage = double(voltage);
        voltage = (voltage-mean_v)/std_v;
        volt_supply = 3*(p1_v*voltage^2 + p2_v*voltage + p3_v);
        %volt_supply = 3*((3.3/1023)*voltage);
        app.BatteryVoltageLabel.Text =  ['Battery Voltage: ', num2str(volt_supply)];

        app.PacketTimeLabel.Text = ['Packet Time: ', num2str(packet_time)];


        %9DoF Monitor
        accel_Z = typecast(uint8(messages(19:20)),'int16');
        accel_z = double(accel_Z)/1000;

        %app.AccelerometerLabel.Text =  ['Accel. Z: ', num2str(accel_z)];


        %Plot Temperature Sensors Data
        plot(app.TempInt, temp_ex, h/1000, 'b.');
        plot(app.TempInt, temp_in, h/1000, 'r.');
        plot(app.TempInt, temp_ex2, h/1000, 'g.');


        %Plot Accelerometer Data
        plot(app.Voltage, datetime, accel_z,'Marker','.', 'Color','b');

        pause(0.00001);
    end

    %Check if it is a GPS Packet and parse it
 elseif ((read_Byte(i:i+1)==id_gps(1:end)))
    packets_gps = packets_gps+1;
    timer_4 = toc(timer_3);
    %gps_Time = typecast(uint8(messages(27:30)),'uint32');
    %gps_time = double(gps_Time)/1000;

    min_gps = 10;
    if (range>5000)
        min_gps = 13;
    end

    if ((packets_gps==min_gps)||(timer_4>3))
        timer_3 = tic;
        packets_gps=0;

        lat = double(typecast(uint8(messages(5:8)),'int32'))/10000000;
        lon = double(typecast(uint8(messages(9:12)),'int32'))/10000000;
        h = double(typecast(uint8(messages(13:16)),'int32'))/100;
        stat = messages(17);
        numSats = messages(18);
        utcHour = messages(19);
        utcMin = messages(20);
        utcSec = messages(21);
        gps_time = typecast(uint8(messages(27:30)),'uint32');


         %Voltage Monitor
         %voltage = typecast(uint8(messages(35:36)),'uint16');
         %voltage = double(voltage);
         %volt_supply=3*((3.3/1023)*voltage);
         %app.BatteryVoltageLabel.Text =  ['Battery Voltage: ', num2str(volt_supply)];

        newAlt = h;
        newTime = double(gps_time/1000);

        ascentRate = double((newAlt - prevAlt)/(newTime - prevTime));
        %if ((ascentRate>0)&&(ascentRate<15))
            app.AscentRate.Text  = num2str(ascentRate);
        %end
```

```
                                prevAlt = h;
                                prevTime = double(gps_time/1000);

                                if app.PredictionFileButton.Value == 0
                                    %Compute the AZ/EL parameters for the GS and range.
                                    [az,el,range] = geodetic2aer(lat,lon,h,lat0,lon0,h0,app.spheroid);
                                end
                                %Plot GS Location.
                                plot3(app.Location3D, lon0, lat0, h0/1000, 'r*', 'LineWidth',3)
                                plot(app.Location2D, lon0, lat0, 'r*')

                                %Plot Ublox GPS Data
                                plot3(app.Location3D, lon, lat, h/1000,'b*','LineWidth',1)
                                plot(app.Location2D, lon, lat,'b*')

                                %Print Current GPS Data
                                app.gpsLAT.Text  = num2str(lat);
                                app.gpsLONG.Text = num2str(lon);
                                app.gpsALT.Text  = num2str(h);
                                app.gpsFIX.Text = num2str(stat);
                                app.gpsSATS.Text = num2str(numSats);
                                app.gpsHOUR.Text = [num2str(utcHour), ':'];
                                app.gpsMIN.Text = [num2str(utcMin), ':'];
                                app.gpsSEC.Text = num2str(utcSec);


                                app.ElevationGauge.Value=el+str2num(app.ELTuningEditField.Value);
                                app.AzGauge.Value = az-str2num(app.DeclinationEditField.Value);
                                app.sltRange.Text=num2str(range/1000);


                                pause(0.00001);

                                %Send desired pointing to Arduino-Rotor
                                if (app.flag_GS == 1)
                                    app.ConnectingLabel.Text = 'Moving';
                                    app.ConnectingLabel.Visible = 'on';
                                    if app.ManualButton.Value == 1
                                        % no control
                                    elseif app.AzElButton.Value == 1
                                        fprintf(app.serial_GS,'%s\n',['ElAz',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f'),num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                                        elseif app.AzOnlyButton.Value ==1
                                        fprintf(app.serial_GS,'%s\n',['setAz',num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                                         elseif app.ElOnlyButton.Value ==1
                                        fprintf(app.serial_GS,'%s\n',['setEl',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f')]);
                                    end
                                end
                            end
                         end
                        %Compute the number of lost packets in this considered data block
                        if (rcvd_packets == 1)
                            prev_packet_number = double(packet_num);
                        else
                            new_packet_number = double(packet_num);
                        end

                        if (((new_packet_number-prev_packet_number)~=1)&&((new_packet_number-prev_packet_number)~=-65535))
                            if ((new_packet_number-prev_packet_number)>1)
                                lost_packets = lost_packets + (new_packet_number - prev_packet_number - 1);

                            elseif (((new_packet_number-prev_packet_number)<0)&&(rcvd_packets ~= 1))
                                lost_packets = lost_packets + (65535 - prev_packet_number) + new_packet_number;
                            end
                        end

                        prev_packet_number = packet_num;
                    end
                end
            end
            %Do not consider the first and last packet of the data block as packet losses
            if (lost_packets<5)
                lost_packets = 0;
            else
                %Do not consider GPS packets
                lost_packets = lost_packets - 4;
            end

            %Print received and lost packets information
            lost_total = lost_total + double(lost_packets);
            app.ReceivedPackets.Text = ['Received Packets: ',num2str(rcvd_packets)];
            app.LostPackets.Text = ['Lost Packets: ',num2str(lost_total-app.ExtraLossesEditField.Value)];
            app.LossesLabel.Text = ['%Losses: ',num2str(100*(lost_total-app.ExtraLossesEditField.Value)/((lost_total-app.ExtraLossesEditField.Value)+rcvd_packets))];
            lost_packets = 0;

            pause(0.00001);
        end
        %--------------------------------

        %Check timer
        timerCheck = toc(timerIni);

        if ((app.PredictionFileButton.Value == 1)&&(timerCheck>timeThreshold))
           %Reset Timer
```

```
           timerIni = tic;

           if (app.ManualButton2.Value == 1)
               %Select altitude from GUI
               alt = app.AltmSpinner.Value;

                %Altitude during ascent or descent?
                %Grab data accordingly
                if (app.AscentButton.Value == 1)
                    hConC = app.hC(1:app.maxC);
                    distC = abs(hConC-alt);
                    rowC = find(distC == min(distC));

                elseif (app.DescentButton.Value == 1)
                    hConC = app.hC(app.maxC:end);
                    distC = abs(hConC-alt);
                    rowC = find(distC == min(distC)) + (app.maxC-1);
                end
           end

           if (app.AutomaticButton.Value == 1)
                predTime = app.CUSFpredicted(:,5);

                timeNow = datetime - app.startTime;
                vecTime = datevec(timeNow);
                totalSecs = (vecTime(4)*3600) + (vecTime(5)*60) + (vecTime(6));

                distSecs = abs(totalSecs - predTime);
                rowC = find(distSecs == min(distSecs));
           end
            %Grab the data from the selected row. Only for the selected pred. file
            %Compute AZ/El for the rotor controller and range for the GUI

            lat=app.CUSFpredicted(rowC(1),2); % lat
            lon=app.CUSFpredicted(rowC(1),3); % lon
            h=app.CUSFpredicted(rowC(1),4); % alt

            [az,el,range] = geodetic2aer(lat,lon,h,lat0,lon0,h0,app.spheroid);



            %Send desired pointing to Arduino-Rotor
            if (app.flag_GS == 1) %If the ground station is connected
            app.ConnectingLabel.Text = 'Moving';
            app.ConnectingLabel.Visible = 'on';
                if app.ManualButton.Value == 1
                % no control due to movement occuring via GS rotor controller
                elseif app.AzElButton.Value == 1
                    fprintf(app.serial_GS,'%s\n',['ElAz',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f'),num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                    elseif app.AzOnlyButton.Value ==1
                    fprintf(app.serial_GS,'%s\n',['setAz',num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                    elseif app.ElOnlyButton.Value ==1
                    fprintf(app.serial_GS,'%s\n',['setEl',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f')]);
                end
            end

            %Print Current Data from prediction file to the labels and gauges
            app.gpsLAT.Text  = num2str(lat);
            app.gpsLONG.Text = num2str(lon);
            app.gpsALT.Text  = num2str(h);
            app.gpsFIX.Text = "N/A";
            app.gpsSATS.Text = "N/A";
            app.gpsHOUR.Text = "N/A";
            app.gpsMIN.Text = "N/A";
            app.gpsSEC.Text = "N/A";
            app.ElevationGauge.Value=el;
            app.AzGauge.Value = az-str2num(app.DeclinationEditField.Value);
            app.sltRange.Text=num2str(range/1000);

            %Delete previous plots for A and B properties of GUI
            delete(app.A);
            delete(app.B);
            %Set the data for the plots to be the values of LLA for that specific predition file
            app.xdata = lon;
            app.ydata = lat;
            app.zdata = h/1000;
            %Actually plot the prediction trajectory
            app.A = plot(app.Location2D,app.xdata,app.ydata,'b*','LineWidth',1);
            app.B = plot3(app.Location3D,app.xdata,app.ydata,app.zdata,'b*','LineWidth',1);
            %pause(3);
        end
        pause(0.00002);
   end
end

                % Button pushed function: ConnectGSButton
        function ConnectGSButtonPushed(app, event)
            % Initialize Serial Communication with Arduino and MATLAB.
            % The Arduino sends a Char and waits for MATLAB to respond with the proper
            % Char. If no errors, setup ok indication is visible.

            app.flag_GS = 1;

            app.serial_GS = serial(app.GSCOMPortDropDown.Value);


```

```
            set(app.ConnectingLabel,'Visible', 'on');

            %Set serial parameters
            app.serial_GS.InputBufferSize = 300000;
            set(app.serial_GS, 'DataBits', 8);
            set(app.serial_GS, 'StopBits', 1);
            set(app.serial_GS, 'BaudRate', 230400);
            set(app.serial_GS, 'Parity', 'none');

            %Open the serial port
            try
                fopen(app.serial_GS);
            catch err
                fclose(app.serial_GS);
                error('Make sure you select the correct Arduino COM Port.');
            end

            set(app.ConnectGSButton,'Enable','off');
            set(app.DisconnectGSButton,'Enable','on');
            while (app.serial_GS.BytesAvailable == 0)

            end
            a=fscanf(app.serial_GS,'%e');
            fprintf(app.serial_GS,'%s\n','getAz');

            while (app.serial_GS.BytesAvailable == 0)

            end

            app.AzGauge.Value = fscanf(app.serial_GS,'%e');

            fprintf(app.serial_GS,'%s\n','getEl');
            while (app.serial_GS.BytesAvailable == 0)
            end

            app.ElevationGauge.Value = fscanf(app.serial_GS,'%e');

            set(app.ConnectingLabel,'Visible', 'off');

            %After connection allow gps polling
            %set(app.AutoButton,'Enable','on');


           app.GroundStationTestButton.Enable = 'on';
           app.BeginTrackingButton.Enable='on';


        end

        % Button pushed function: DisconnectGSButton
        function DisconnectGSButtonPushed(app, event)


            fclose(app.serial_GS);
            delete(app.serial_GS);
            clear app.serial_GS;
            set(app.ConnectGSButton,'Enable','on')
            set(app.ConnectingLabel,'Visible', 'off');
            set(app.DisconnectGSButton,'Enable','off');
            %set(app.OKLabel,'Visible','off');
            set(app.AutoButton,'Visible','off');


        end

        % Callback function
        function AutoButtonPushed(app, event)
            % function to load current position to gs_lat,lon and alt from gs gps
            fprintf(app.serial_GS,'getLoc');
            location=fgetl(app.serial_GS);
            M =strsplit(location,',');
            while length(M) ~= 6
                fprintf(app.serial_GS,'getLoc');
                location=fgetl(app.serial_GS);
                M =strsplit(location,',');
            end
            if string(M(1))=='lat'
                app.GS_Latitude.Value=str2num(cell2mat(M(2)));
            end
            if string(M(3))=='lon'
                app.GS_Longitude=str2num(cell2mat(M(4)));
            end
            if string(M(5))=='alt'
                app.GS_Altitude=str2num(cell2mat(M(6)));
            end
        end

        % Close request function: UIFigure
        function UIFigureCloseRequest(app, event)
            delete(instrfindall);
            delete(app)

        end

        % Callback function
        function GPS_Selection(app, event)
            %disp("GPS CHANGED!");
        end

```

```
        % Value changed function: GSCOMPortDropDown
        function GSCOMPortDropDownValueChanged(app, event)
            app.GSCOMPortDropDown.Items = cellstr(seriallist);
            app.RxCOMPortDropDown.Items = app.GSCOMPortDropDown.Items;

        end

        % Value changed function: RxCOMPortDropDown
        function RXCOMPortDropDownValueChanged(app, event)
            app.RxCOMPortDropDown.Items = cellstr(seriallist);
            app.GSCOMPortDropDown.Items = app.RxCOMPortDropDown.Items;
        end

        % Callback function
        function CalibrateGSButtonPushed(app, event)
            prompt = ['You are about to perform an initial calibration. Please set the GS-5500 to an azimuth of
                180',char(176), ' then select Next'];
            type = questdlg(prompt,'Initial Calibration','Next','Cancel');
            switch type
                case 'Next'
                    GSCal;
                case 'Cancel'
                    return
            end

        end

        % Callback function: LoadPredictionFileMenu,
        % LoadPredictionfileButton
        function LoadPredictionfileButtonPushed(app, event)

            pAns = questdlg('Which type of prediction path would you like to plot?',...
                'Prediction Path Option',...
                'University of Wyoming','CUSF','Cancel','CUSF');

            switch pAns
                case 'CUSF'
                    [newfile,path] = uigetfile('*.csv','Pediction Path File','flight_path.csv');
                    figure(app.UIFigure);
                    if newfile ~= 0
                        app.predFileFlag = app.predFileFlag + 1;

                        predFile=fullfile(path,newfile);

                        % Predicted path plot from hab-hub.org predictor
                        % http://predict.habhub.org
                        app.CUSFpredicted=csvread(predFile);
                        app.ydata=app.CUSFpredicted(:,2); % lat
                        app.xdata=app.CUSFpredicted(:,3); % lon
                        app.zdata=app.CUSFpredicted(:,4); % alt
                        app.wdata=app.CUSFpredicted(:,5); % time

                        plot3(app.Location3D,app.xdata,app.ydata,app.zdata./1000,app.predFileCol(app.
                            predFileFlag),'LineWidth',2)
                        plot(app.Location2D,app.xdata,app.ydata,app.predFileCol(app.predFileFlag),'LineWidth'
                            ,2)


                    end
                case 'Cancel'

                case 'University of Wyoming'
                    [newfile,path] = uigetfile('*.csv','Pediction Path File','flight_path.csv');
                    figure(app.UIFigure);
                    if newfile ~= 0

                        predFile=fullfile(path,newfile);

                        % Predicted path plot from hab-hub.org predictor
                        % http://predict.habhub.org
                        app.WYpredicted=csvread(predFile,3,1);
                        app.ydata=app.WYpredicted(:,1); % lat
                        app.xdata=app.WYpredicted(:,2); % lon
                        app.zdata=app.WYpredicted(:,3); % alt
                        %plot3(app.Location3D,app.xdata,app.ydata,app.zdata./1000,'r','LineWidth',2)
                        %plot(app.Location2D,app.xdata,app.ydata,'r','LineWidth',2)



                        app.xdata=app.WYpredicted(:,10); % Temperature
                        if min(app.xdata) < app.TempInt.XLim(1)+5
                            app.TempInt.XLim(1) = min(app.xdata) - 15;
                        end

                        plot(app.TempInt,app.xdata,app.zdata./1000,'k');  % Plot temperature prediction


                    end


                    %Prediction File Parameters
                    app.hC = app.CUSFpredicted(:,4); % alt
                    app.maxC = find(app.hC==max(app.hC));
                    app.BeginTrackingButton.Enable='on';



                    %                 case 'Create New CUSF'
                    %                     web('http://predict.habhub.org','-new','-noaddressbox', '-notoolbar')
```

```
;
                    %                     uiwait(msgbox('Opening HabHub.org Prediction tool. Save the file in csv format. Then Press OK.',...
                    %                         'Get Prediction File'));
                    %                     [newfile,path] = uigetfile('*.csv','Pediction Path File','flight_path.csv');
                    %                     if newfile ~= 0
                    %
                    %
                    %                         predFile=fullfile(path,newfile);
                    %
                    %                         % Predicted path plot from hab-hub.org predictor
                    %                         % http://predict.habhub.org
                    %                         predicted=csvread(predFile);
                    %                         app.ydata=predicted(:,2); % lat
                    %                         app.xdata=predicted(:,3); % lon
                    %                         app.zdata=predicted(:,4); % alt
                    %                         plot3(app.Location3D,app.xdata,app.ydata,app.zdata./1000,'y','LineWidth',2)
                    %                         plot(app.Location2D,app.xdata,app.ydata,'y','LineWidth',2)
                    %
                    %                         plot3(app.Location3D, lon0, lat0, h0/1000, 'y*', 'LineWidth',1)
                    %                         plot(app.Location2D, lon0, lat0, 'y*')
                    %                     end

            end
        end

                % Callback function
        function PredictionFileDropDownValueChanged(app, event)
            value = app.PredictionFileDropDown.Value;
            if strcmp(value,'CUSF')
                app.AltmSpinner.Limits(2) = max(app.CUSFpredicted(:,4));

            elseif strcmp(value,'Wyoming')
                app.AltmSpinner.Limits(2) = max(app.WYpredicted(:,3));

            end

        end

        % Menu selected function: RefreshCOMPortsMenu
        function RefreshCOMPortsMenuSelected(app, event)
            app.RxCOMPortDropDown.Items = cellstr(seriallist);
            app.GSCOMPortDropDown.Items = app.RxCOMPortDropDown.Items;
        end

        % Button pushed function: LaunchTimeButton
        function LaunchTimeButtonPushed(app, event)
            app.startTime = datetime;
            app.LaunchTimeLabel.Text = ['Launch Time: ',datestr(app.startTime, 'HH:MM:SS')];
        end

        % Button pushed function: GroundStationTestButton
        function GroundStationTestButtonPushed(app, event)
          while(true)
            pause(3);

            lat0=app.myGSCoord(1);
            lon0=app.myGSCoord(2);
            h0=app.myGSCoord(3);
%
            alt = app.AltmSpinner.Value;

            if (app.AscentButton.Value == 1)
               hConC = app.hC(1:app.maxC);
               distC = abs(hConC-alt);
               rowC = find(distC == min(distC));

           elseif (app.DescentButton.Value == 1)
               hConC = app.hC(app.maxC:end);
               distC = abs(hConC-alt);
               rowC = find(distC == min(distC)) + (app.maxC-1);
           end

           if (app.ManualButton2.Value == 1)
               autoFlag = 0;
               %Grab the data from the selected row. Only for the selected pred. file
               %Compute AZ/El for the rotor controller and range for the GUI

               lat=app.CUSFpredicted(rowC(1),2); % lat
               lon=app.CUSFpredicted(rowC(1),3); % lon
               h=app.CUSFpredicted(rowC(1),4); % alt
               [az,el,range] = geodetic2aer(lat,lon,h,lat0,lon0,h0,app.spheroid);

               if (app.flag_GS == 1) %If the ground station is connected
                    app.ConnectingLabel.Text = 'Moving';
                    app.ConnectingLabel.Visible = 'on';
                    if app.ManualButton.Value == 1
                        % no control due to movement occuring via GS rotor controller
                        elseif app.AzElButton.Value == 1
                            fprintf(app.serial_GS,'%s\n',['ElAz',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f'),num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                        elseif app.AzOnlyButton.Value ==1
                            fprintf(app.serial_GS,'%s\n',['setAz',num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                        elseif app.ElOnlyButton.Value ==1
                            fprintf(app.serial_GS,'%s\n',['setEl',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f')]);
```

```
                    end
                end

                %Print Current Data from prediction file to the labels and gauges
                app.gpsLAT.Text  = num2str(lat);
                app.gpsLONG.Text = num2str(lon);
                app.gpsALT.Text  = num2str(h);
                app.gpsFIX.Text = "N/A";
                app.gpsSATS.Text = "N/A";
                app.gpsHOUR.Text = "N/A";
                app.gpsMIN.Text = "N/A";
                app.gpsSEC.Text = "N/A";
                app.ElevationGauge.Value=el;
                app.AzGauge.Value = az-str2num(app.DeclinationEditField.Value);
                app.sltRange.Text=num2str(range/1000);

                %Delete previous plots for A and B properties of GUI
                delete(app.A);
                delete(app.B);
                %Set the data for the plots to be the values of LLA for that specific predition file
                app.xdata = lon;
                app.ydata = lat;
                app.zdata = h/1000;
                %Actually plot the prediction trajectory
                app.A = plot(app.Location2D,app.xdata,app.ydata,'b*','LineWidth',1);
                app.B = plot3(app.Location3D,app.xdata,app.ydata,app.zdata,'b*','LineWidth',1);
            end

            if (app.AutomaticButton.Value == 1)
                if autoFlag == 0
                     for i = rowC:length(app.wdata)
                        pause(3);
                        %Grab the data from the selected row. Only for the selected pred. fil
                        %Compute AZ/El for the rotor controller and range for the GUI
                        lat=app.CUSFpredicted(i,2); % lat
                        lon=app.CUSFpredicted(i,3); % lon
                        h=app.CUSFpredicted(i,4); % alt
                        [az,el,range] = geodetic2aer(lat,lon,h,lat0,lon0,h0,app.spheroid);

                        if (app.flag_GS == 1) %If the ground station is connected
                            app.ConnectingLabel.Text = 'Moving';
                            app.ConnectingLabel.Visible = 'on';
                            if app.ManualButton.Value == 1
                                % no control due to movement occuring via GS rotor controller
                            elseif app.AzElButton.Value == 1
                                fprintf(app.serial_GS,'%s\n',['ElAz',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f'),num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                            elseif app.AzOnlyButton.Value ==1
                                fprintf(app.serial_GS,'%s\n',['setAz',num2str(az-str2num(app.DeclinationEditField.Value), '%03.0f')]);
                            elseif app.ElOnlyButton.Value ==1
                                fprintf(app.serial_GS,'%s\n',['setEl',num2str(el+str2num(app.ELTuningEditField.Value),'%03.0f')]);
                            end
                        end

                       %Print Current Data from prediction file to the labels and gauges
                       app.gpsLAT.Text  = num2str(lat);
                       app.gpsLONG.Text = num2str(lon);
                       app.gpsALT.Text  = num2str(h);
                       app.gpsFIX.Text = "N/A";
                       app.gpsSATS.Text = "N/A";
                       app.gpsHOUR.Text = "N/A";
                       app.gpsMIN.Text = "N/A";
                       app.gpsSEC.Text = "N/A";
                       app.ElevationGauge.Value=el;
                       app.AzGauge.Value = az-str2num(app.DeclinationEditField.Value);
                       app.sltRange.Text=num2str(range/1000);

                       %Delete previous plots for A and B properties of GUI
                       delete(app.A);
                       delete(app.B);
                       %Set the data for the plots to be the values of LLA for that specific predition file
                       app.xdata = lon;
                       app.ydata = lat;
                       app.zdata = h/1000;
                       %Actually plot the prediction trajectory
                       app.A = plot(app.Location2D,app.xdata,app.ydata,'b*','LineWidth',1);
                       app.B = plot3(app.Location3D,app.xdata,app.ydata,app.zdata,'b*','LineWidth',1);

                       if (i==length(app.wdata))
                           autoFlag = 1;
                       end
                    end
                end
            end
         end
        end

        % Button pushed function: ReproduceFlightButton
        function ReproduceFlightButtonPushed(app, event)
            [newfile,path] = uigetfile('*.bin','Create Data File','data.bin');
            if newfile == 0
                return;
            end
            filename=fullfile(path,newfile);

            s1=fopen(filename,'r+');
```

```
lat0=app.myGSCoord(1);
lon0=app.myGSCoord(2);
h0=app.myGSCoord(3);


id_scient=[160,177]';
id_gps=[192,209]';
messages=zeros(1,100);
rcvd_packets = 0;
packets_sci = 0;
packets_gps = 0;
lost_packets = 0;
lost_total = 0;
packet_num = 0;
new_packet_number = 0;
range = 0;
timer_1 = tic;

%External Thermistor Coefficients:
p1_ex = 13.6;
p2_ex = -6.838;
p3_ex = 20.3;
p4_ex = -14.81;


mean_ex = 423.8;
std_ex = 358.5;

%Internal Thermistor Coefficients:
p1_in = -5.2;
p2_in = -9.875;
p3_in = -24.22;
p4_in = 19.94;

mean_in = 742.8;
std_in = 224.8;



%Initial Position
lat = lat0;
lon = lon0;
h = h0;

%Ascent Rate Monitor Variables
prevAlt = 35;
prevTime = 0;

ascentRate = 0;


min_gps = 10*4;
min_sci = 10*125;

read_Byte = fread(s1);
       for i=1:(length(read_Byte)-102)
            %Loof for the start of a scientific or GPS packet.
             if((read_Byte(i:i+1)==id_scient)|(read_Byte(i:i+1)==id_gps))
                %Check if the packet has been completely received.
                if ((read_Byte(i+100:i+101)==id_scient)|(read_Byte(i+100:i+101)==id_gps))
                    rcvd_packets = rcvd_packets+1;
                    messages(1:100)=read_Byte(i:i+99);
                    packet_num = typecast(uint8(messages(3:4)),'uint16');
                    packet_num = double(packet_num);

                    if (rcvd_packets == 1)
                        prev_packet_number = packet_num;
                    end
                    %Check if it is a Scientific Packet and parse it
                     if (read_Byte(i:i+1)==id_scient)
                        packets_sci = packets_sci+1;
                        timer_2 = toc(timer_1);
                        if (packets_sci==min_sci)
                            packets_sci=0;
                            %External Temperature Conversion
                            temp = typecast(uint8(messages(5:6)),'uint16');
                            temp = double(temp);
                            temp = (temp-mean_ex)/std_ex;
                            temp_ex = p1_ex*temp^3 + p2_ex*temp^2 + p3_ex*temp + p4_ex;


                            %Internal Temperature Conversion
                            temp = typecast(uint8(messages(7:8)),'uint16');
                            temp = double(temp);
                            temp = (temp-mean_in)/std_in;
                            temp_in = p1_in*temp^3 + p2_in*temp^2 + p3_in*temp + p4_in;


                            %Voltage Monitor
                            voltage = typecast(uint8(messages(13:14)),'uint16');
                            voltage = double(voltage);
                            volt_supply=3*((3.3/1023)*voltage);
                            app.BatteryVoltageLabel.Text =  ['Battery Voltage: ', num2str(
                                 volt_supply)];

                            %9DoF Monitor
                            accel_X = typecast(uint8(messages(37:38)),'int16');
                            accel_x = double(accel_X)/1000;
```

```
                    accel_Y = typecast(uint8(messages(39:40)),'int16');
                    accel_y = double(accel_Y)/1000;
                    accel_Z = typecast(uint8(messages(41:42)),'int16');
                    accel_z = double(accel_Z)/1000;

                    %app.AccelerometerLabel.Text =  ['Accel. Z: ', num2str(accel_z)];


                    %Plot Temperature Sensors Data
                    plot(app.TempInt, temp_ex, h/1000, 'b.');
                    plot(app.TempInt, temp_in, h/1000, 'r.');


                    %Plot Acceleromete Data
                    %plot(app.Voltage, datetime, volt_supply,'Marker','.', 'Color','r');
                    %plot(app.Voltage, datetime, accel_x,'Marker','.', 'Color','r');
                    %plot(app.Voltage, datetime, accel_y,'Marker','.', 'Color','g');
                    plot(app.Voltage, datetime, accel_z, 'Marker','.', 'Color','b');


                    %Print received and lost packets information
                    lost_total = lost_total + double(lost_packets);
                    app.ReceivedPackets.Text = ['Received Packets: ',num2str(rcvd_packets)
                          ];
                    app.LostPackets.Text = ['Lost Packets: ',num2str(lost_total)];
                    app.LossesLabel.Text = ['%Losses: ',num2str(100*(lost_total/(lost_total
                          +rcvd_packets)))];
                    lost_packets = 0;


                    pause(0.001);
                end

                %Check if it is a GPS Packet and parse it
            elseif ((read_Byte(i:i+1)==id_gps))
                packets_gps = packets_gps+1;

                if (packets_gps==min_gps)
                    packets_gps=0;

                    lat = double(typecast(uint8(messages(5:8)),'int32'))/10000000;
                    lon = double(typecast(uint8(messages(9:12)),'int32'))/10000000;
                    h = double(typecast(uint8(messages(13:16)),'int32'));
                    stat = messages(17);
                    numSats = messages(18);
                    utcHour = messages(19);
                    utcMin = messages(20);
                    utcSec = messages(21);
                    packetTime = double(typecast(uint8(messages(27:30)),'uint32'));

                    newAlt = h;
                    newTime = double(packetTime/1000);

                    ascentRate = double((newAlt - prevAlt)/(newTime - prevTime));
                    app.AscentRate.Text  = num2str(ascentRate);

                    prevAlt = h;
                    prevTime = double(packetTime/1000);


                    %Compute the AZ/EL parameters for the GS and range.
                    [az,el,range] = geodetic2aer(lat,lon,h,lat0,lon0,h0,app.spheroid);

                    %Plot GS Location.
                    plot3(app.Location3D, lon0, lat0, h0/1000, 'r*', 'LineWidth',3)
                    plot(app.Location2D, lon0, lat0, 'r*')

                    %Plot Ublox GPS Data
                    plot3(app.Location3D, lon, lat, h/1000,'b*','LineWidth',1)
                    plot(app.Location2D, lon, lat,'b*')

                    %Print Current GPS Data
                    app.gpsLAT.Text  = num2str(lat);
                    app.gpsLONG.Text = num2str(lon);
                    app.gpsALT.Text  = num2str(h);
                    app.gpsFIX.Text = num2str(stat);
                    app.gpsSATS.Text = num2str(numSats);
                    app.gpsHOUR.Text = [num2str(utcHour), ':'];
                    app.gpsMIN.Text = [num2str(utcMin), ':'];
                    app.gpsSEC.Text = num2str(utcSec);


                    app.ElevationGauge.Value=el+str2num(app.ELTuningEditField.Value);
                    app.AzGauge.Value = az-str2num(app.DeclinationEditField.Value);
                    app.sltRange.Text=num2str(range/1000);

                    pause(0.001);

                end
             end

             %Compute the number of lost packets in this considered data block

            new_packet_number = packet_num;

            if (((new_packet_number-prev_packet_number)~=1)&&((new_packet_number-
                  prev_packet_number)~=-65535))
                if ((new_packet_number-prev_packet_number)>1)
                    lost_packets = lost_packets + (new_packet_number - prev_packet_number -
```

```
                                                1);
                                        end

                                        if ((new_packet_number-prev_packet_number)<0)
                                            lost_packets = lost_packets + (65535 - prev_packet_number) +
                                                 new_packet_number;
                                        end
                                    end
                                    prev_packet_number = packet_num;
                                end

                            end
                        end
        end

         % Button pushed function: RestartRxPortButton
        function RestartRxPortButtonPushed(app, event)
            %Serial for the radio communication or file
            fclose(app.s);

            app.s = serial(app.RxCOMPortDropDown.Value);

            %Set serial parameters
            app.s.InputBufferSize = 1000000;
            set(app.s, 'DataBits', 8);
            set(app.s, 'StopBits', 1);
            set(app.s, 'BaudRate', 230400);
            set(app.s, 'Parity', 'none');

             %Open the serial port
            try
                fopen(app.s);
            catch err
                fclose(app.s);
                warndlg('Make sure you select the correct Radio COM Port.');
            end
        end
    end

 % App initialization and construction
 [...]
  methods (Access = public)
        % Construct app
        function app = MURI_HAB_GUI_v14PL2_Mobile_GS

            % Create and configure components
            createComponents(app)

            % Register the app with App Designer
            registerApp(app, app.UIFigure)

            % Execute the startup function
            runStartupFcn(app, startupFcn)

            if nargout == 0
                clear app
            end
        end
        % Code that executes before app deletion
        function delete(app)
            % Delete UIFigure when app is deleted
            delete(app.UIFigure)
        end
    end
end
```

Appendix D

Thermistors Calibration

The thermistors calibration is mainly based on two different parts: the temperature range adjustment and the ADC-temperature fitting.

D.1 Temperature Range Adjustment

The thermistor needs power to get a ”temperature” reading. The temperature reading is actually a voltage value that the ADC of the microcontroller used will read. The voltage will decrease or increase, depending on how the voltage divider is built and the temperature change.

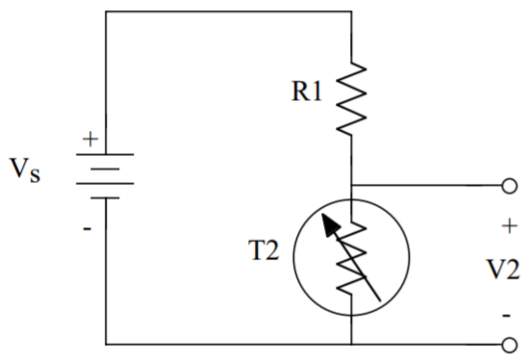


Figure D.1: Thermistor Calibration - Voltage Divider

The previous figure shows a simple voltage divider used to measure the change in resistance of the thermistor, T_2. Considering that the same current flows through R_1 and T_2, the voltage V_2 can be computed as:

$$V2 = \frac{T_2 * V_s}{R_1 + T_2} \tag{D.1}$$

The thermistors considered for the payloads are Negative Temperature Coefficient (NTC) thermistors, which means that the resulting resistance will decrease while the temperature increases, and therefore the voltage will decrease increase as well, if the thermistor position in the voltage divider is the one considered in Figure D.1. The resistance at room temperature -normally defined as R_{25}-, is a key point to calibrate them, because the resistors considered for the voltage divider will have to consider this parameter for a better temperature range fit. For a 5KOhm R_{25} thermistor, a 5KOhm resistor for the voltage divider would be enough for room temperatures of payload internal temperature [0 °C, 50 °C]. However, considering the Z curves characteristics, for lower temperature ranges -payload external temperature- a multiple of 5KOhm would be needed. For this design, a set of resistors of a total 50KOhm resistance is considered.

The voltage V_2 is used to fit/calibrate the real temperature around the thermistors and the ADC counts (voltage) from the voltage divider.

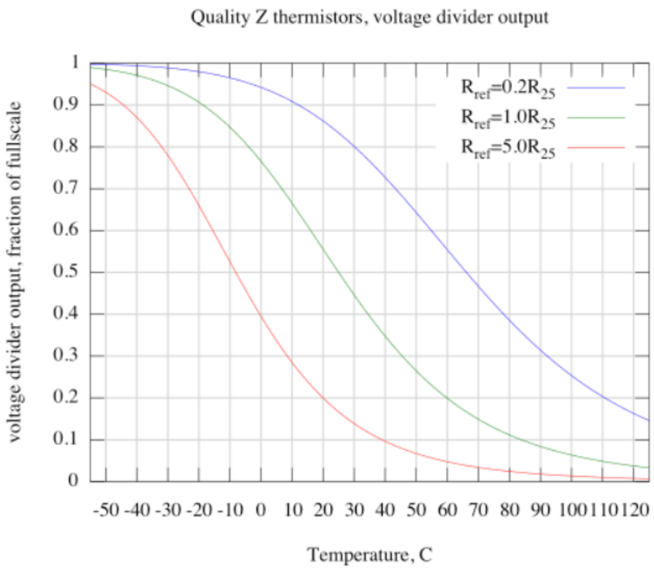


Figure D.2: Thermistor Calibration - Z Curves

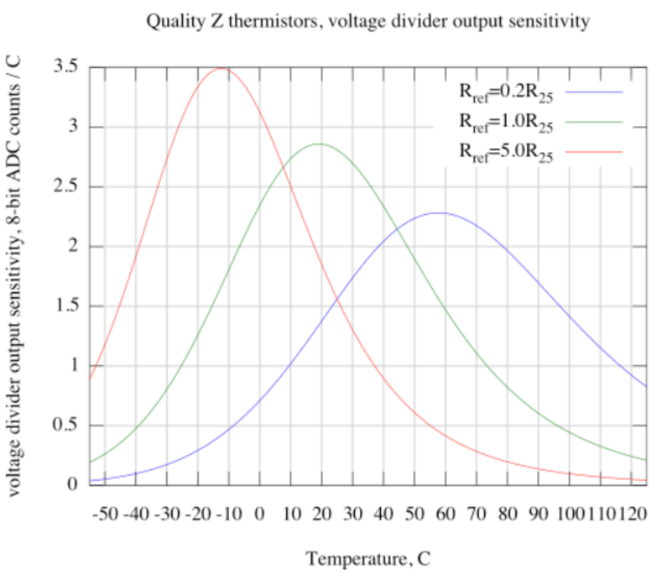


Figure D.3: Thermistor Calibration - Voltage Divider Sensibility

D.2 ADC-Temperature Fitting

To do the fitting between the ADC and the actual temperature, the temperature chamber is used. A temperature profile of 2 hours is used to simulate the temperature changes that the thermistors will be experiencing during the flight. The profile will start at around 55-60°C in order to calibrate the internal one, and the temperature inside the chamber will decrease in 1.5h to -70°C. After that, it will increase again to -55°C and then come back to room temperature.

To calibrate all the thermistors at the same time, the temperature of the chamber is recorded at the same time that the ADC counts for each thermistors are recorded as well. To do that, the microcontroller is programmed to output the ADC readings at a certain rate. The microcontroller is connected via USB to the same laptop that the temperature chamber will be connected as well. With a MATLAB program, whenever the microcontroller outputs ADC readings, the temperature of the chamber is read and all the results are printed in the MATLAB workspace for monitoring purposes, and they are saved in a .TXT file using a pre-defined format.

Figure D.4: Temperature chamber calibration controls and thermistors being calibrated.

These are the microcontroller and MATLAB codes used for the calibration process:

- Microcontroller Code

```
#include <ADC.h>

//ANALOG PINS DEFINITION
#define TEMP_EXT_H A9
#define TEMP_INT A8
#define TEMP_EXT_L A7
#define VOLTAGE A6

int tempExt_h;                     //Upper Range External Temperature sensor.
int tempExt_l;                     //Lower Range External Temperature sensor.
int tempInt;                       //Internal Temperature sensor.

byte temp[6];
ADC *adc = new ADC();

void setup() {
 Serial.begin(9600);
 analogReadResolution(10);
 adc->setReference(ADC_REFERENCE::REF_3V3, ADC_0);
 adc->setConversionSpeed(ADC_CONVERSION_SPEED::LOW_SPEED); // change the conversion speed
}

void loop() {
  delay(2000);

  tempExt_h = analogRead(TEMP_EXT_H);
  tempExt_l = analogRead(TEMP_EXT_L);
  tempInt = analogRead(TEMP_INT);

  temp[0] = tempExt_h;
  temp[1] = tempExt_h >> 8;

  temp[2] = tempExt_l;
  temp[3] = tempExt_l >> 8;

  temp[4] = tempExt_l;
  temp[5] = tempExt_l >> 8;

  Serial.write(temp, 6);

  Serial.print("External Temperature High: "); Serial.println(tempExt_h);
  Serial.print("External Temperature Low: "); Serial.println(tempExt_l);
  Serial5.print("Internal Temperature: "); Serial.println(tempInt); Serial.println(" ");
  }
```

- MATLAB Code

```matlab
% Temperature chamber and Arduino Boards - Thermistors readings
% Noemi Miguelez, 2019

%% Setup
clc;
clear all;
close all;
fclose('all');
delete(instrfind);

% Measurement duration
duration = 2;    %hours

% Time between measurements defined by the boards (keep same time step between them).

%% Configure File

% Generate dated file name
date_time=fix(clock);
date_time_str=sprintf('%04d%02d%02d_%02d%02d',date_time(1),date_time(2),date_time(3),date_time(4),date_time(5));
file_str=sprintf('%s',mfilename);
text_str=sprintf('%s_%s.txt',file_str,date_time_str);

% Open file for recording data
record_file=fopen(text_str,'w');

%% Create objects and establish connections
TemperatureChamber=modbus('serialrtu','COM24');

%-----------------------------------------------------
%TEENSY/ARDUINO BOARD #1
%Serial for the radio communication or file
board1 = serial('COM29');

%Set serial parameters
board1.InputBufferSize = 20;
set(board1, 'DataBits', 8);
set(board1, 'StopBits', 1);
set(board1, 'BaudRate', 9600);
set(board1, 'Parity', 'none');

%Open the serial port
try
    fopen(board1);
catch err
    fclose(board1);
    warndlg('Connection error with board 1.');
end

tic;

%% Monitor temperature profile
while toc<(duration*3600)
   %Consider the measurements when both boards have data available.
   if (board1.BytesAvailable> 0 )
        readData1 = fread(board1,6);
        temp_ext_h = typecast(uint8(readData1(1:2)),'uint16');
        temp_ext_h = double(temp_ext_h);

        temp_int = typecast(uint8(readData1(3:4)),'uint16');
        temp_int = double(temp_int);

        temp_ext_l = typecast(uint8(readData1(5:6)),'uint16');
        temp_ext_l = double(temp_ext_l);

        % Reading chamber temperature
        chamber_temp=read(TemperatureChamber,'holdingregs',101)/10;
        if chamber_temp>1000
            chamber_temp=chamber_temp-6553.5;
        end

        % Writing data to file
        fprintf(record_file,'%s\t%s\t%s\t%s\t%s\r\n',num2str(toc),num2str(chamber_temp),num2str(temp_ext_h),
              num2str(temp_ext_l), num2str(temp_int));

        % Printing temperatures to terminal for monitoring
        disp([num2str(toc),': ',num2str(chamber_temp),' C, T.Ext_H #1: ',num2str(temp_ext_h),', T.Ext_L: ',
              num2str(temp_ext_l), '. T.Int: ', num2str(temp_int),'.']);
   end
end

%% Cleaning up
close all;
fclose('all');
delete(instrfind);
```

The data recorded from the ADC in counts is fitted to the actual temperature chamber values. The resulting coefficients are used to convert from ADC counts - sent from the payload- to actual temperature - used by the GUI to plot the data for monitoring purposes, and during the data post-processing to analyze the launch results-.

To do that, the following MATLAB code can be used to configure the temperature range for the calibration of each thermistor separately:

```
clc;
clear all;
close all;
fclose('all');
delete(instrfind);

[FileName,PathName] = uigetfile({'*.dat;*.mat'},'File Selector');
data = load(FileName);

%----------EXTERNAL THERMISTOR UPPER RANGE------------
x_exH = data(:,3);
y_exH = data(:,2);
minTemp = -30;
maxTemp = 30;

range = find((y_exH>minTemp)&(y_exH<maxTemp));
x_exH = x_exH(range);
y_exH = y_exH(range);

f_extH = fit(x_exH, y_exH,'poly5','Normalize','on','Robust','Bisquare')

figure;
plot(x_exH, y_exH,'o')
title("Teensy - Upper Range External Thermistor Fit");
hold on
plot(x_exH, f_extH(x_exH), 'x');
xlabel('Counts');
ylabel('Temperature');


%--------EXTERNAL THERMISTOR LOWER RANGE---------
x_exL = data(:,4);
y_exL = data(:,2);
minTemp = -65;
maxTemp = -20;

range = find((y_exL>minTemp)&(y_exL<maxTemp));
x_exL = x_exL(range);
y_exL = y_exL(range);

f_extL = fit(x_exL, y_exL,'poly5','Normalize','on','Robust','Bisquare')

figure;
plot(x_exL, y_exL,'o')
title("Teensy - Lower Range External Thermistor Fit");
hold on
plot(x_exL, f_extL(x_exL), 'x');
xlabel('Counts');
ylabel('Temperature');


%--------INTERNAL THERMISTOR------
x_in = data(:,5);
y_in = data(:,2);
minTemp = -10;
maxTemp = 50;

range = find((y_in>minTemp)&(y_in<maxTemp));
x_in = x_in(range);
y_in = y_in(range);

f_inter = fit(x_in, y_in,'poly5','Normalize','on','Robust','Bisquare')
figure;
plot(x_in, y_in,'o')
title("Teensy - Internal Thermistor Fit");
hold on
plot(x_in, f_inter(x_in), 'x');
xlabel('Counts');
ylabel('Temperature');
```

The obtained calibration coefficients will be valid only for the thermistors used with the same ADC pins of the microcontroller used for the calibration procedure.

It is important to introduce the new coefficients to the GUI and the post-processing scripts used for that payload launch and analysis.

Appendix E

Transceiver Configuration

This appendix presents the XBee PRO SX modules considered for each segment, as well as the configuration of these modules for a multiple ground station tracking scenario.

To configure these radios, the XCTU platform [6] is used. With this software, both boards can be configured at the same time and some communication tests can be performed to check the link.

The ground station and the payload transceivers have almost the same configuration. Only the "Node Identifier" parameter is different in order to identify which configuration is supposed to be used for the GS and which one for the payload. It is prepared this way to distinguish payload and ground station transceivers configuration, in case a different communications setup is preferred.

To be able to configure the payload surface mount chip, the following board is used to connect it do a computer with the configuration software. As it can be seen in Figure E.1, this board includes a USB 2.0 B connection that will be used to communicate with the configuration software. If needed, the module also includes external pins to test the communication between the ground station and the payload modules, as well as indicator LEDs for power, TX and RX checking. A group of three LEDs that work as received signal strength indicator [RSSI] is also included in this board. The communication between GS and payload boards can be tested with XCTU using this interface board; however, it is suggested to run these tests with only 20 dBm output power, since this board cannot handle the 30 dBm configuration.

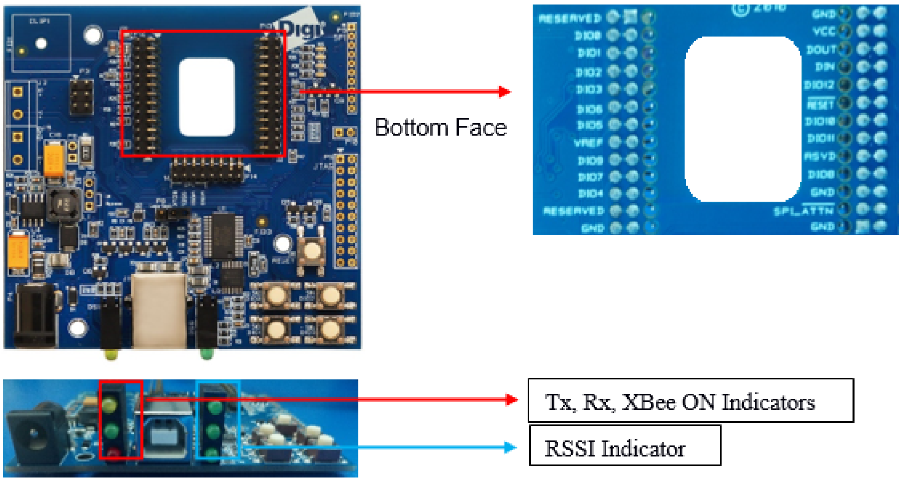


Figure E.1: Transceiver Interface Board for Surface Mount Modules Configuration.

Once the board is connected, it will be detected as a 'COM' port.

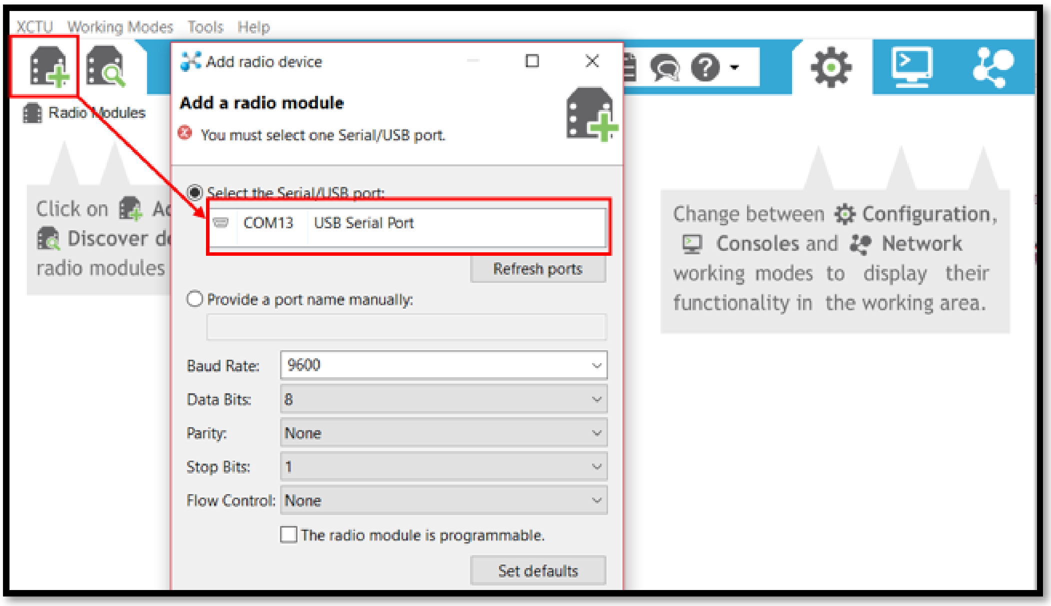


Figure E.2: XCTU - Add a Radio Module.

If it is the first time that the radio module is added to the XCTU -sometimes even after the first configuration-, XCTU will ask you to push the reset button of the board to identify the module or it will inform you and do it automatically. After that, the board-transceiver will be connected to XCTU as it can be seed in Figure E.3.

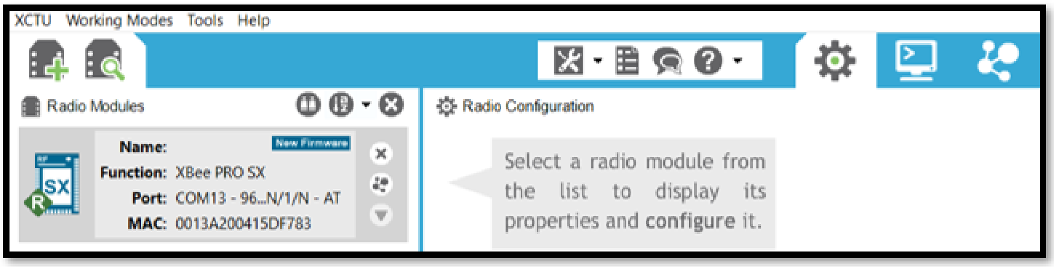


Figure E.3: XCTU - XBee Module Connected/Attached.

By clicking on top of the desired module, a list of all its configured parameters will be presented on the right side of the XCTU panel.

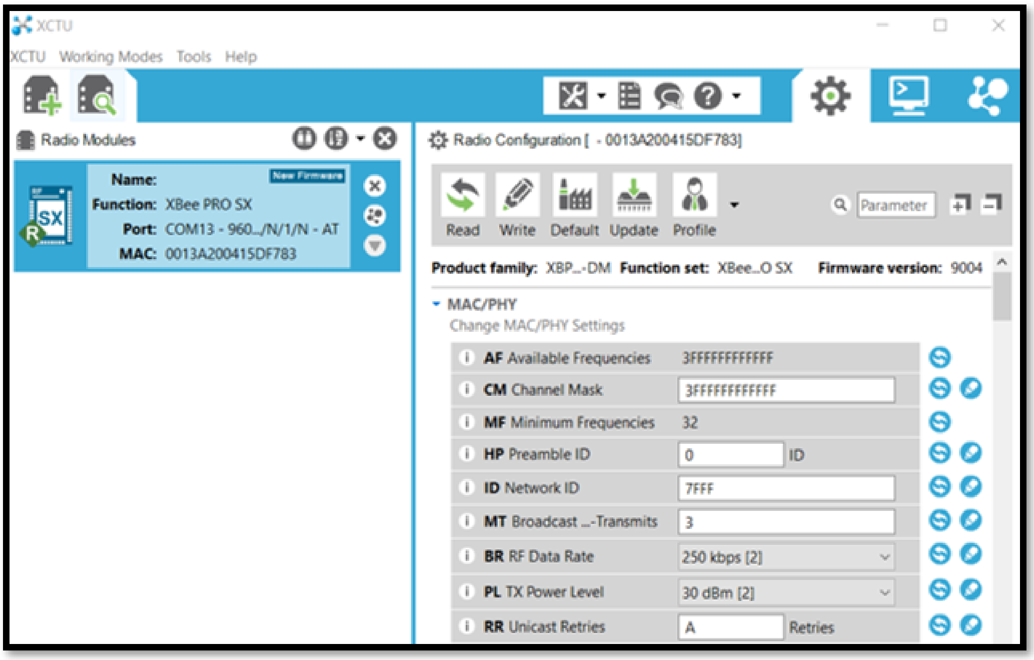


Figure E.4: XCTU - Radio Configuration View.

Each parameter has either one or two blue buttons on their right side, to read/refresh the parameter value from the transceiver or to read and write the value of this parameter, respectively:

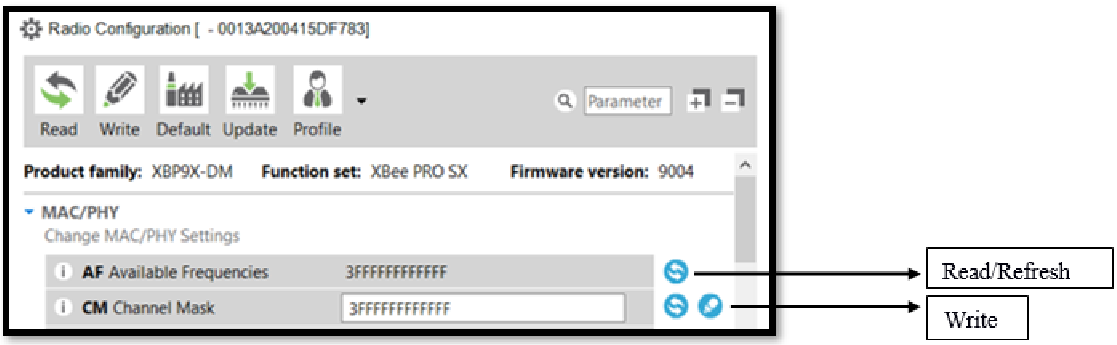


Figure E.5: XCTU - Read/Write Configuration Parameters.

The left side of each parameters contains a button for information about them:

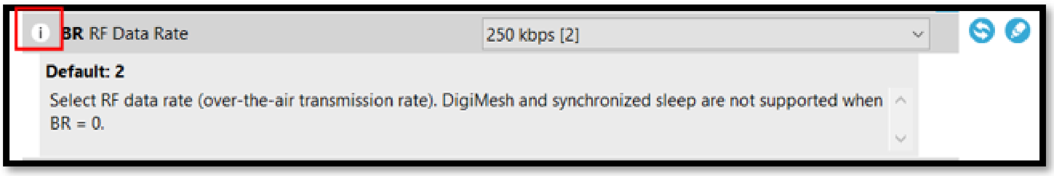


Figure E.6: XCTU - Parameters Information.

For both GS and payload, the following "MAC/PHY" parameters are used:

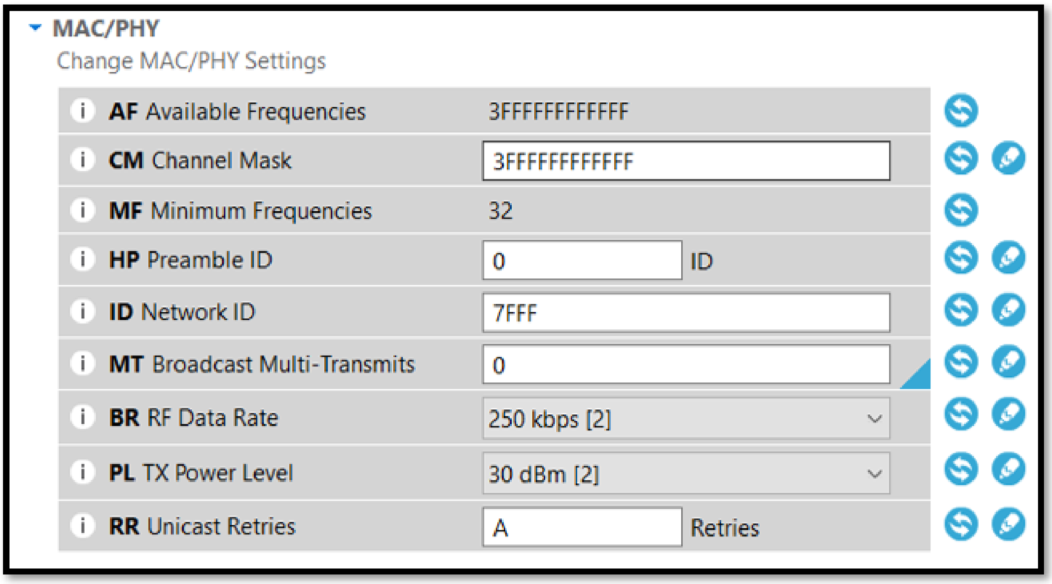


Figure E.7: XCTU - MAC/PHY Parameters Configuration.

- The preamble and network IDs shall match for the radios to be able to communicate with each other.

- In this case, it is specified that the radio should not do additional broadcast retransmissions (to ensure that it is received).

- The RF data rate is configured to be the maximum possible [250 kbps], which it is not the actual data throughput of the communications link.

- The TX power is set to 1W [30 dBm].

For both GS and payload configuration, the following "Network" parameters are used:

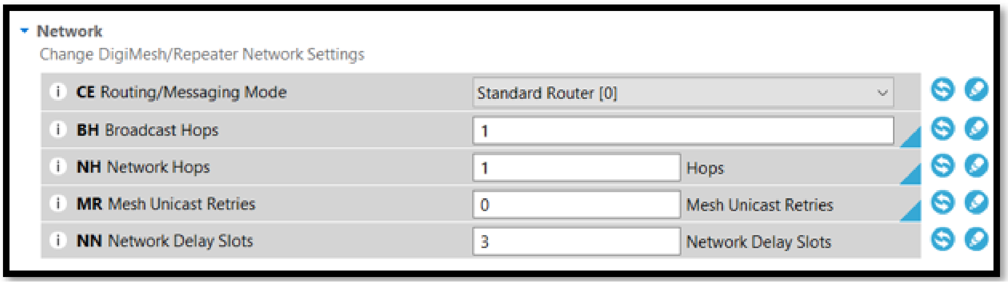


Figure E.8: XCTU - Network Parameters Configuration.

- The number of broadcast and network hops is 1, which represents the maximum number of transmissions hops.

- The mesh unicast retries is 0, to ensure that no acknowledgements are expected if working in unicast mode.

For the GS, the "Addressing" configuration is the following one:

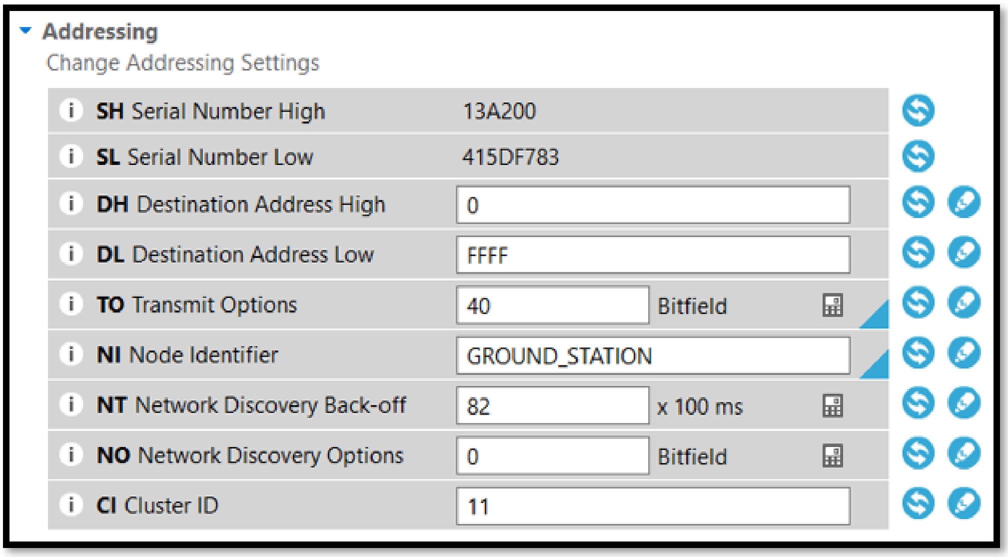


Figure E.9: XCTU - Ground Station Addressing Parameters Configuration.

- The destination address is set to 0x000000000000FFFF [DH: 0, DL: FFFF] because it is the broadcasting address.

- The transmit option is set to 40, which represents the point-to-point/multipoint configuration.

- The node identifier is specified as GROUND_STATION.

For the payload, the following “Addressing” parameters configuration is used:

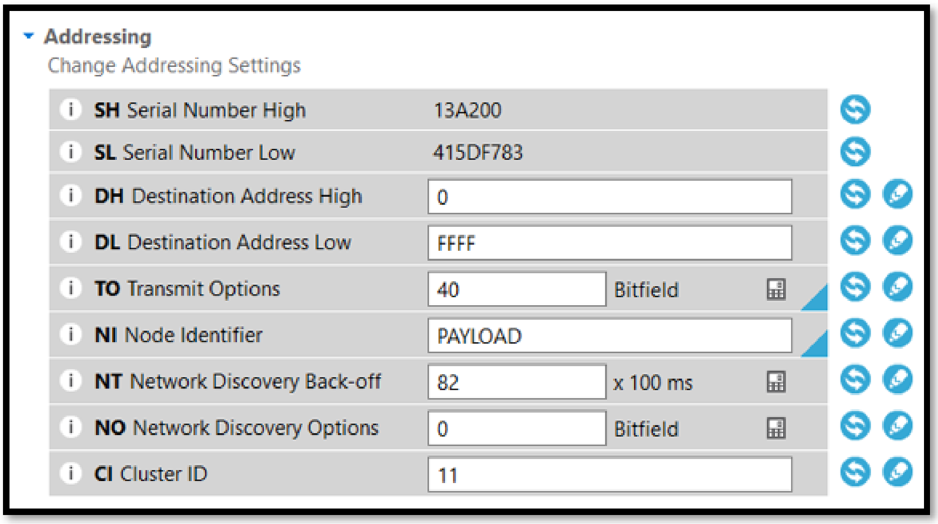


Figure E.10: XCTU - Payload Addressing Parameters Configuration.

- The node identifier is specified as PAYLOAD.

For both GS and payload, the “Serial Interfacing” parameters configuration is the following one:

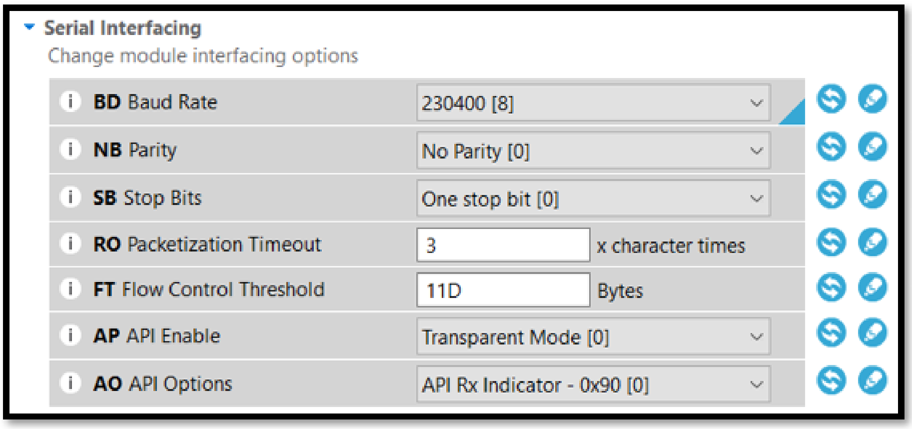


Figure E.11: XCTU - Serial Interfacing Parameters Configuration.

-The baud rate is set to 230k4 bps, which will be used for interfacing between the transceiver module and the microcontroller UART Tx/Rx lines or the USB serial communications with the GS GUI.

-No parity is used in this serial interfacing.

-Only one stop bit is configured.

-The API Enable is set to Transparent Mode.

-The Flow Control Threshold value is configured as default, but it can be changed if CTS/RTS lines are used for flow control purposes. The CTS will be de-asserted if FT bytes are in the UART receive buffer. It is important to configure this value considering the size of the data packets of the payload. CTS should be asserted with enough margin to put the next data packet in the transceiver transmission buffer.

The rest of blocks of configuration parameters are not used for this communications link setup.

The configuration profiles can be saved for next modules configurations. To apply a configuration profile to a new transceiver module, the "Profile – Apply Configuration Profile" buttons shall be used:

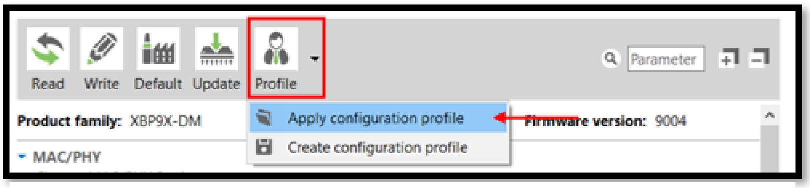


Figure E.12: XCTU - Configuration Profile Application.

Once the radios are configured, the XCTU serial consoles can be used for testing purposes:

1) Open the serial console view.

2) Open the connection with the selected radio.

3) In Tx mode, create the packet to be sent.

4) Specify the desired transmit interval. Specify the number of times that the packet will be sent or transmit infinite number of packets (Loop Infinitely) and start the transmission sequence.

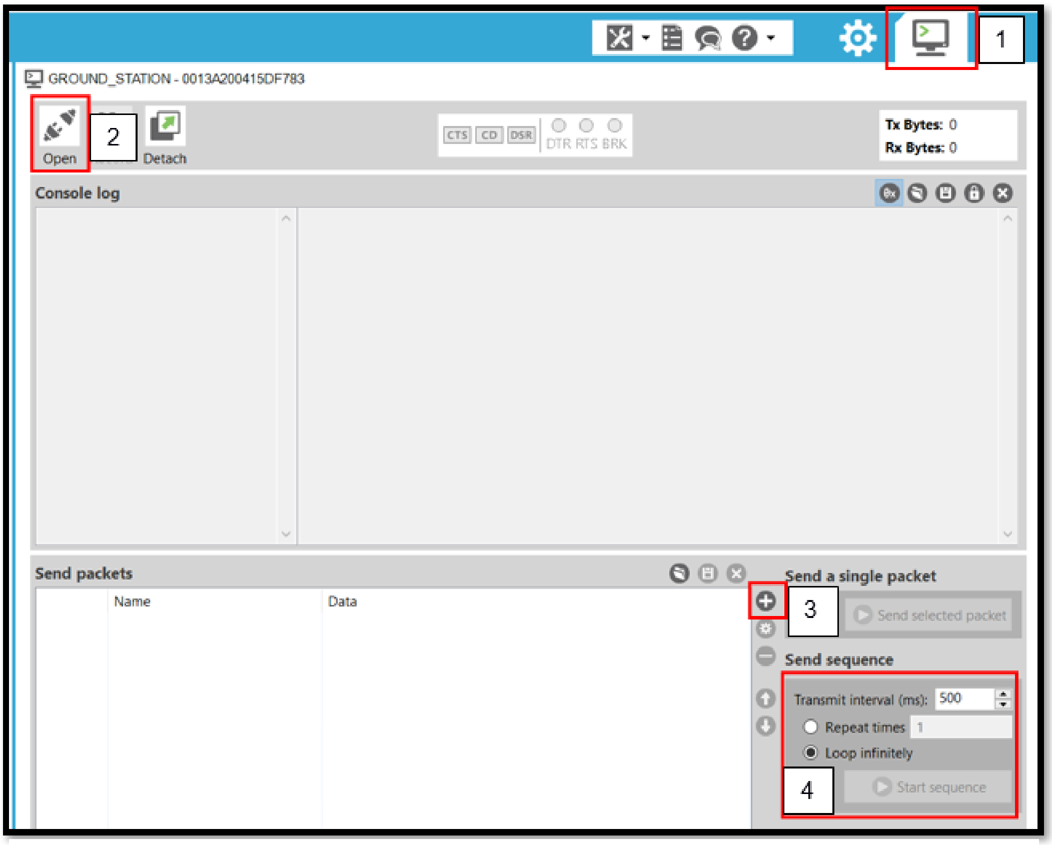


Figure E.13: XCTU - Serial Console View.

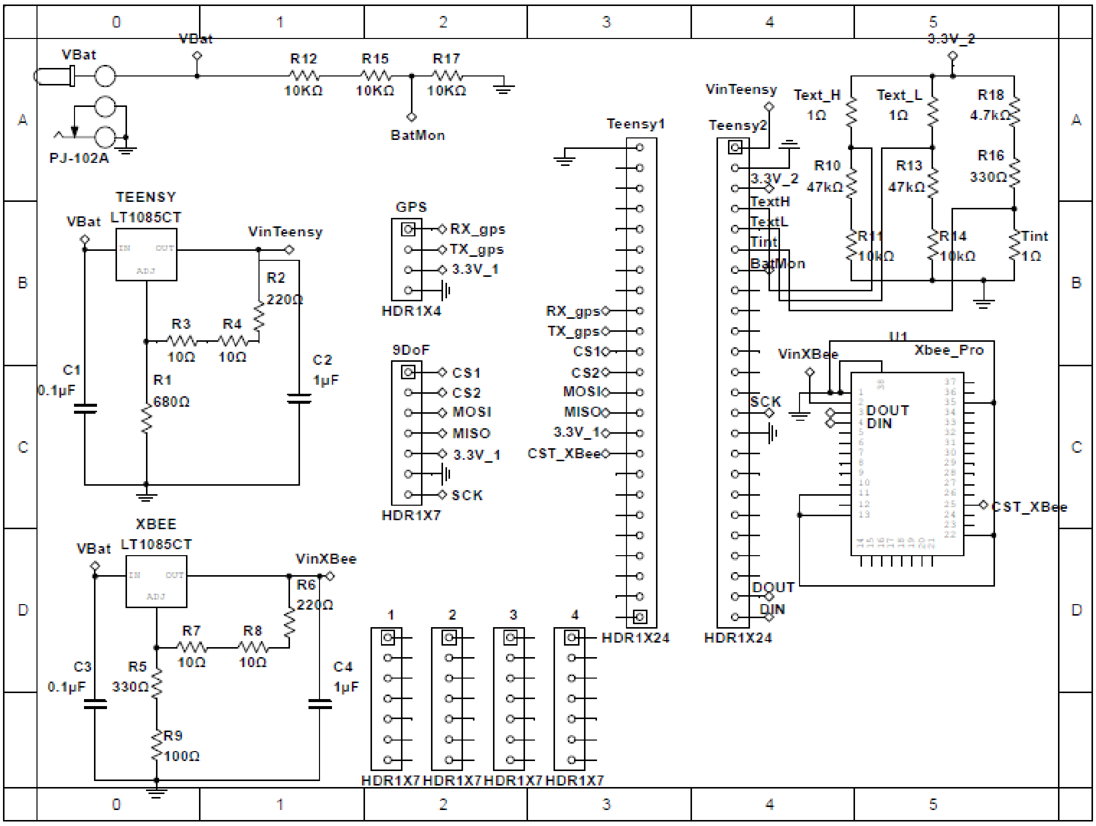
Once the transmission sequence is started, the number of Tx Bytes increases, and the console log shows the transmitted packet in blue.

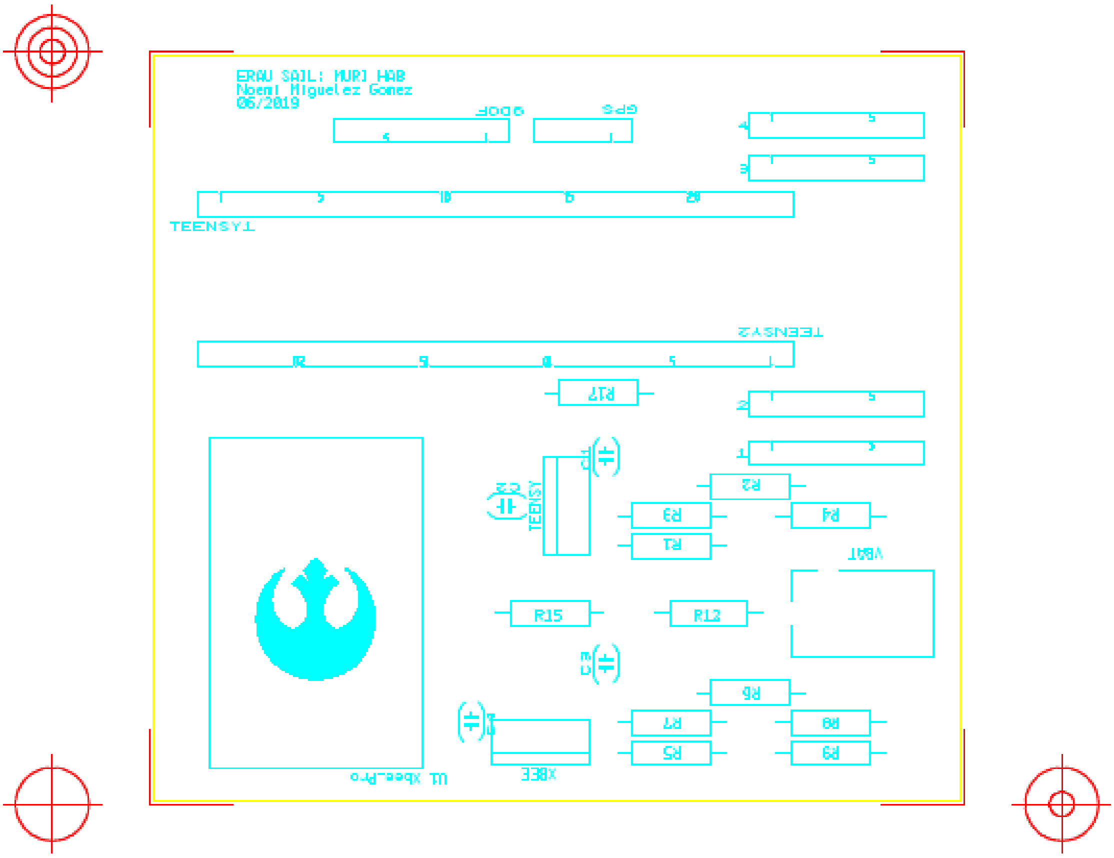
If a receiver radio is configured and attached to the serial console, the previously configured packets will be printed in their console log in red, and the number of Rx Bytes will increase.

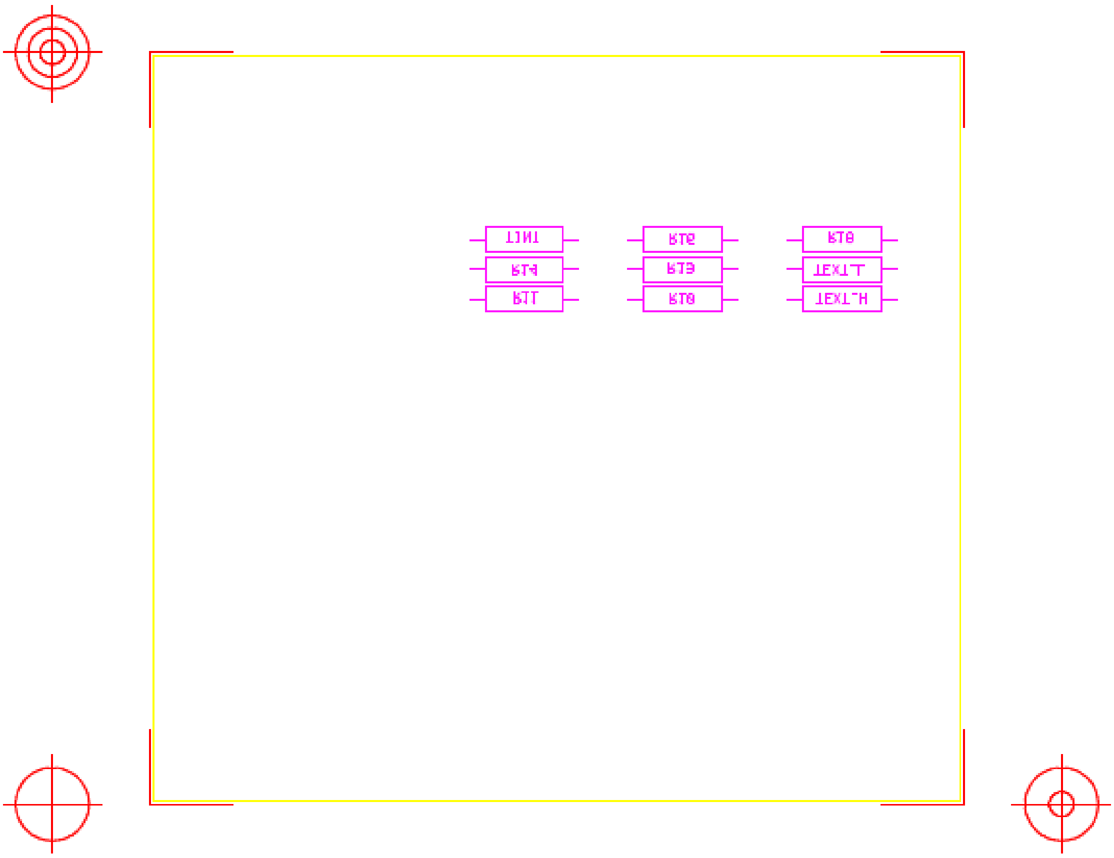
Appendix F

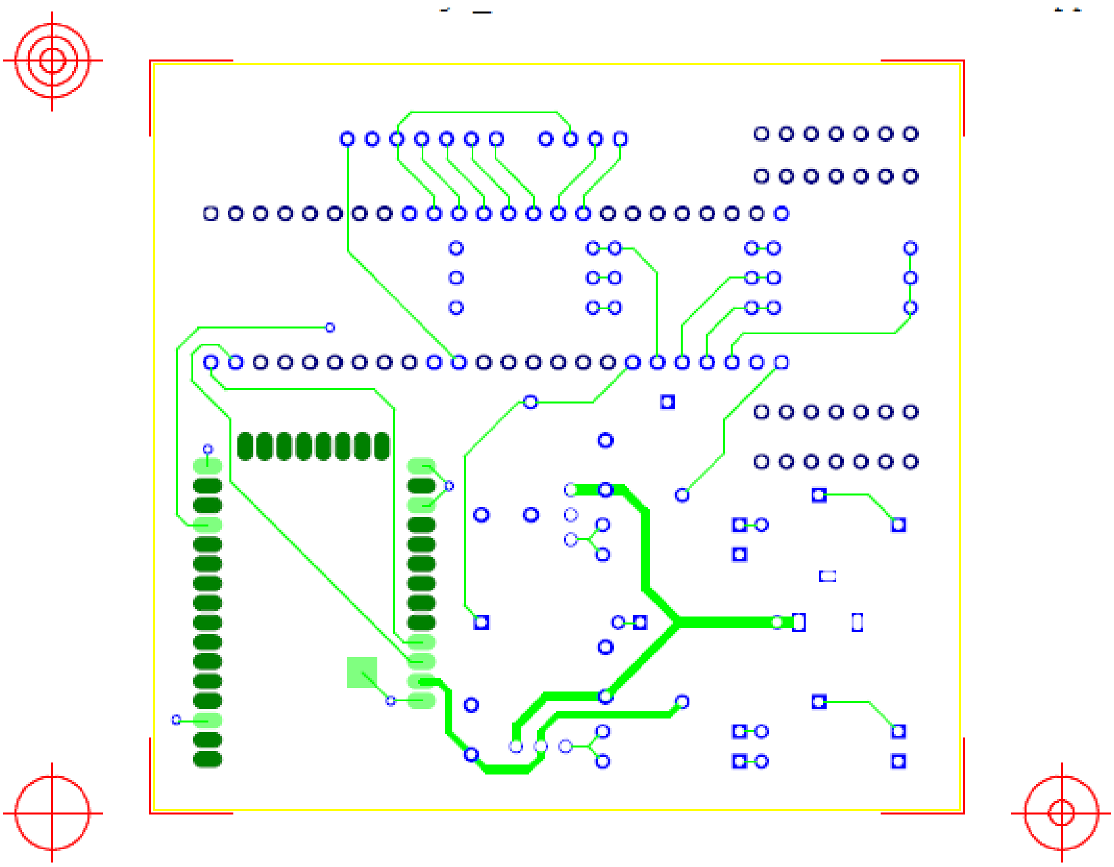
Printed Circuit Board Designs

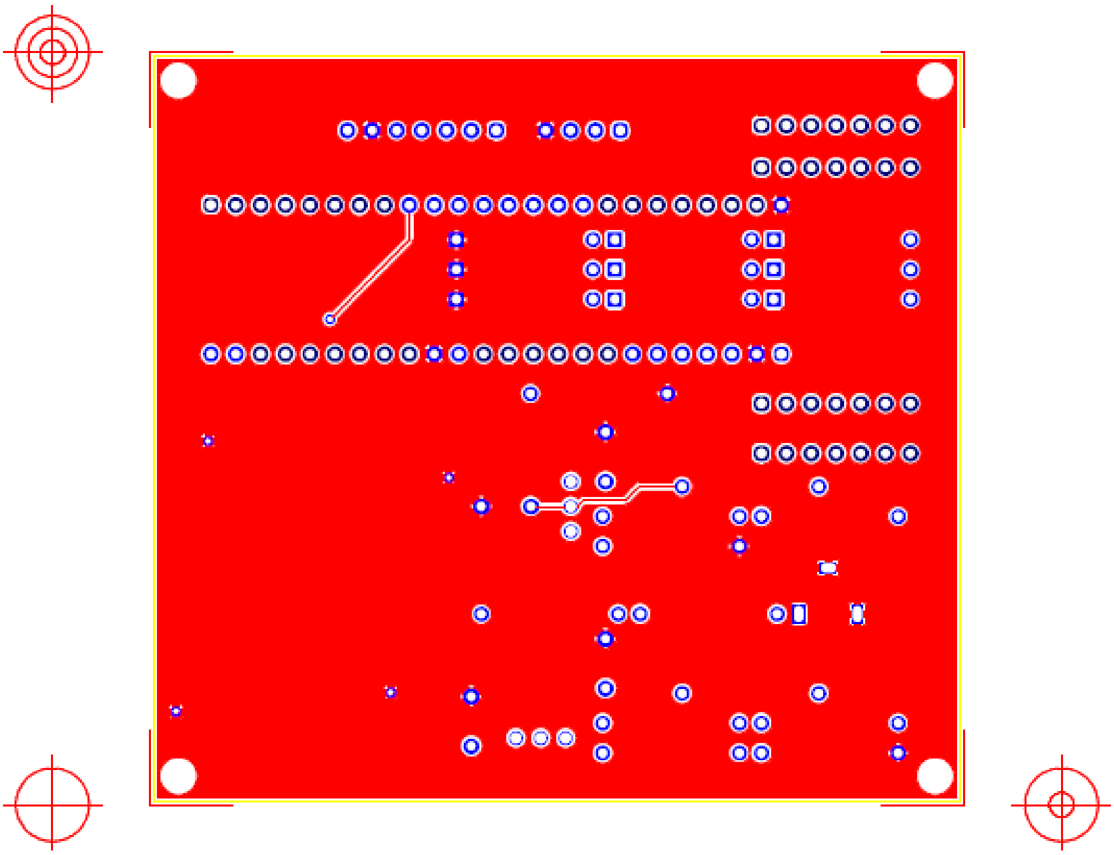
F.1 Payload



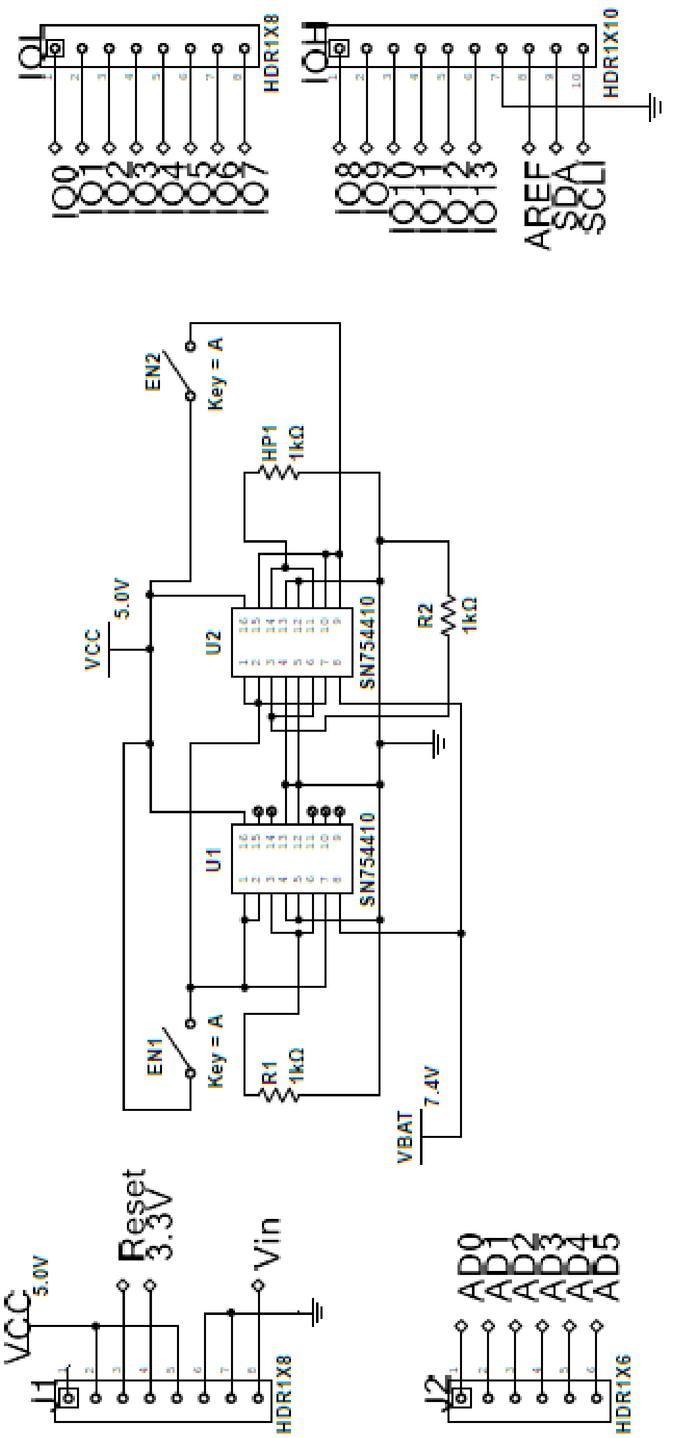


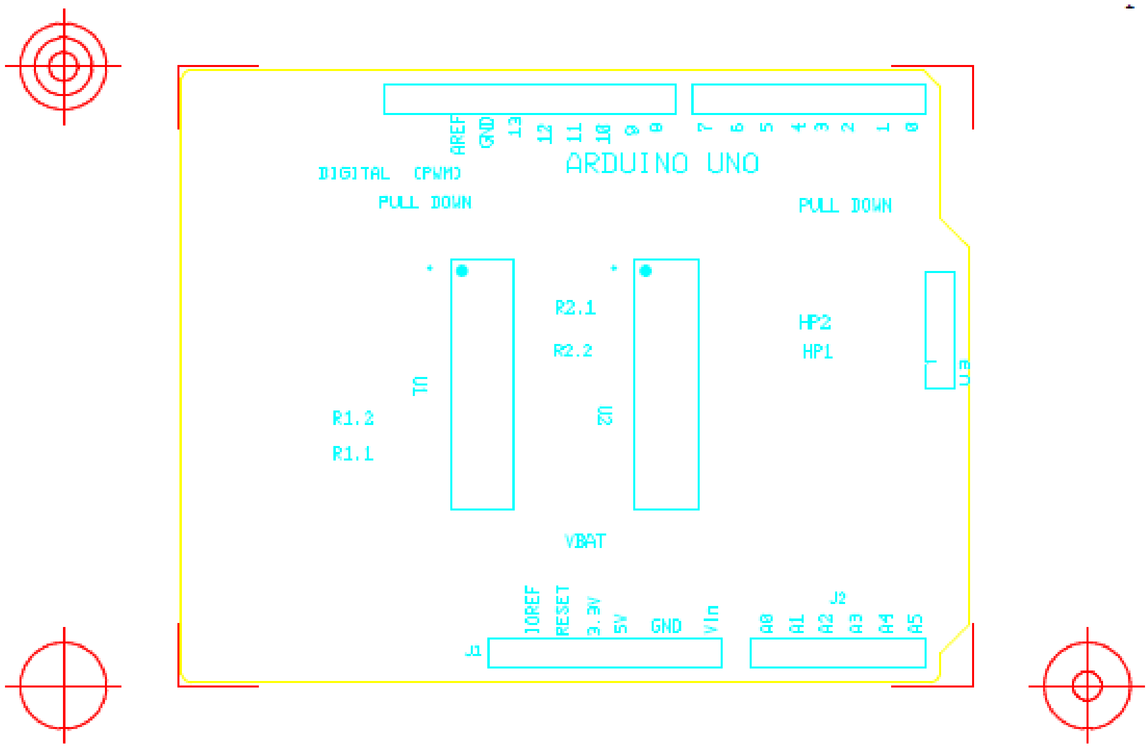


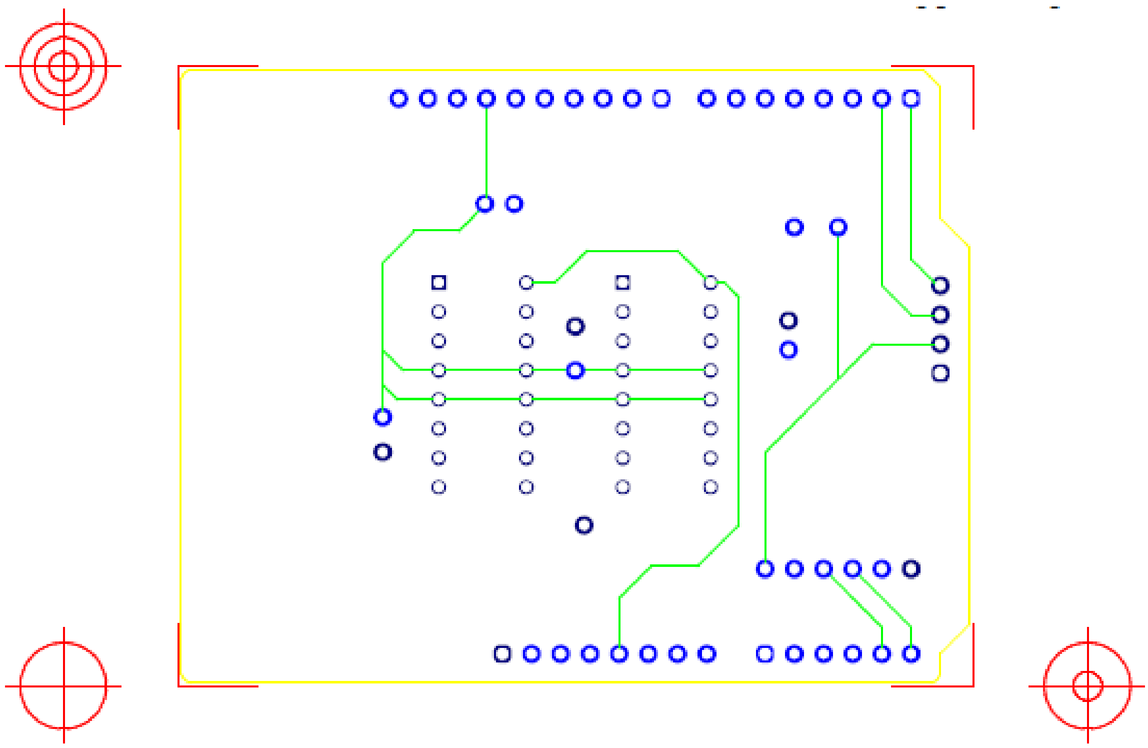


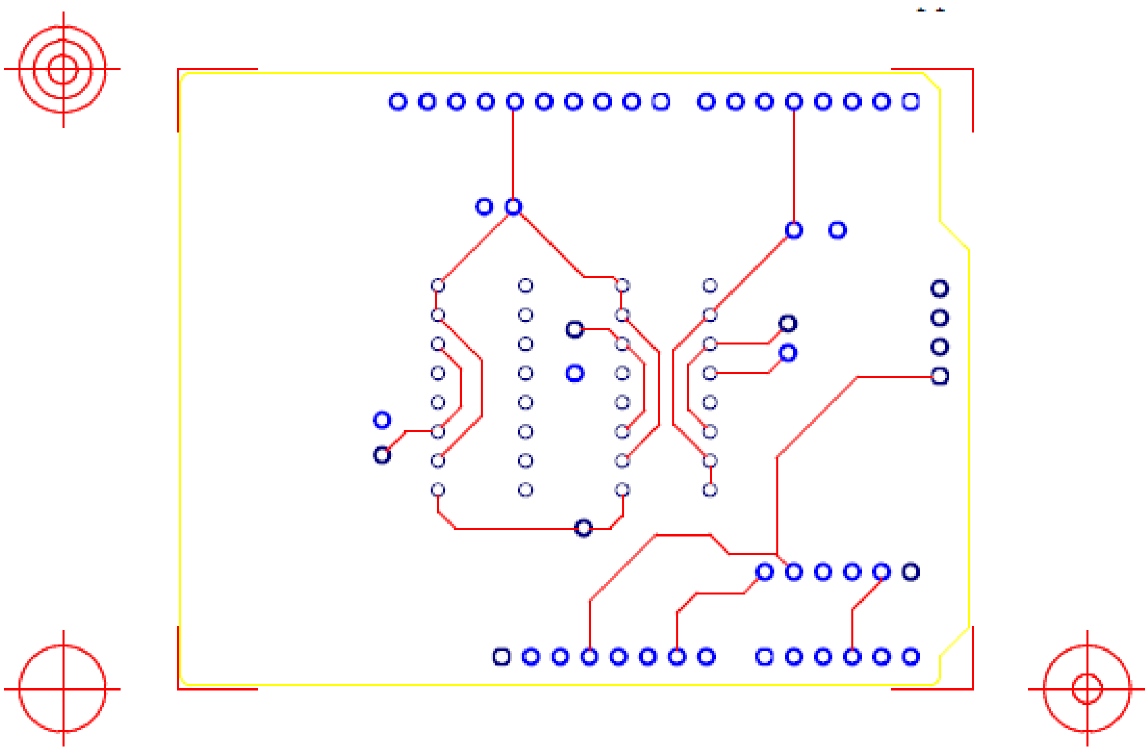


F.2 Controlled Descent Unit

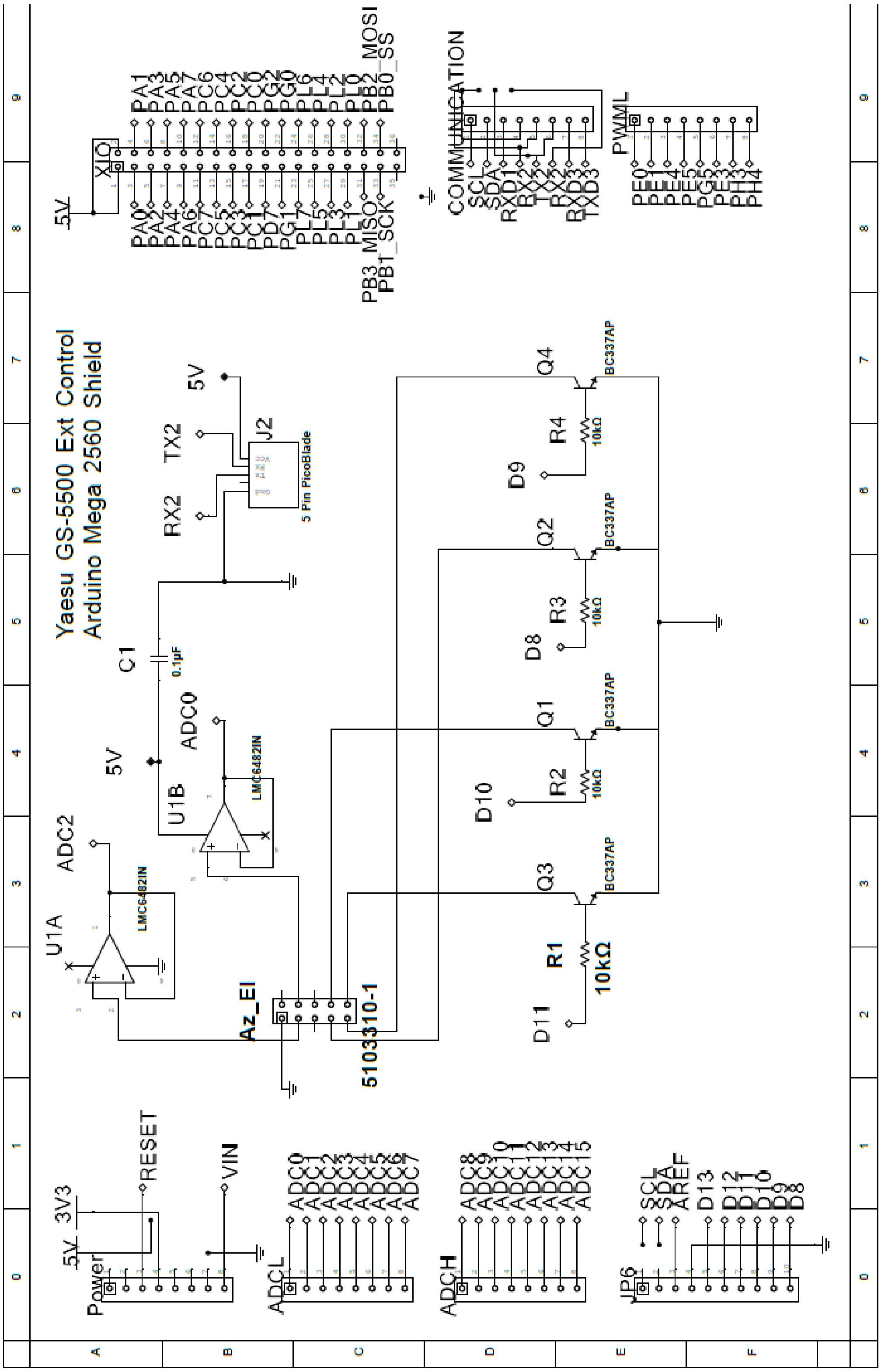


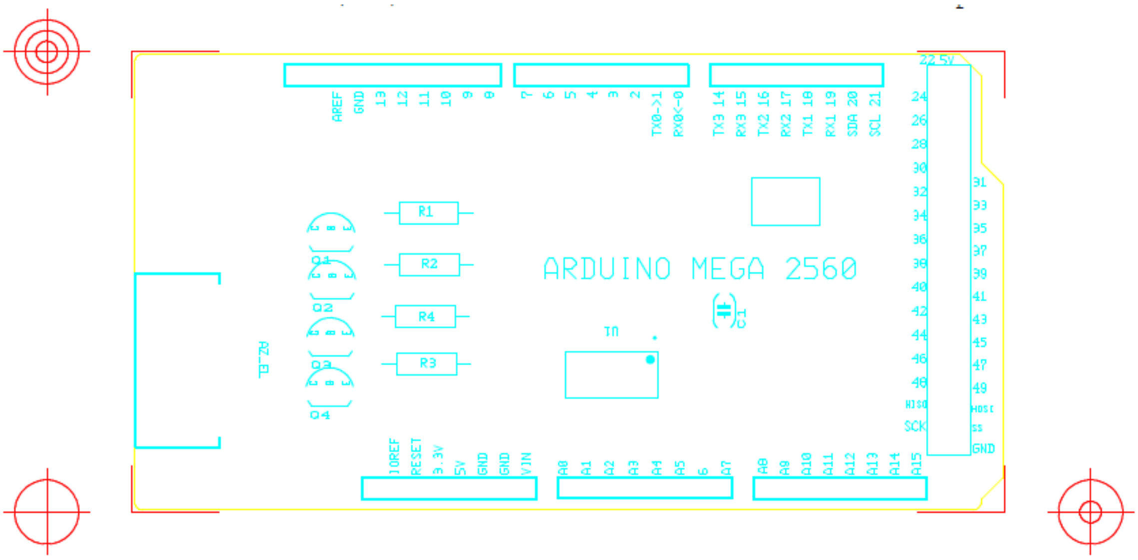

AREF
GND
13
12
11
10
9
8
7
6
5
4
3
2
1
0
DIGITAL (PWM)
ARDUINO UNO
PULL DOWN
PULL DOWN
R2.1
R2.2
HP2
HP1
U1
U2
U3
R1.2
R1.1
VBAT
J2
IOREF
RESET
3.3V
5V
GND
Vin
A0
A1
A2
A3
A4
A5
J1

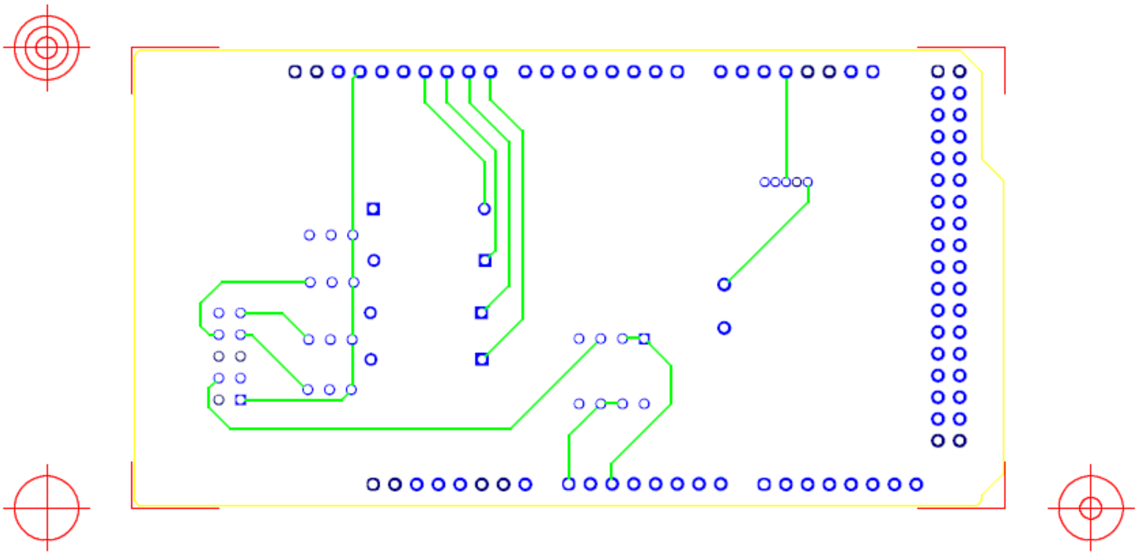


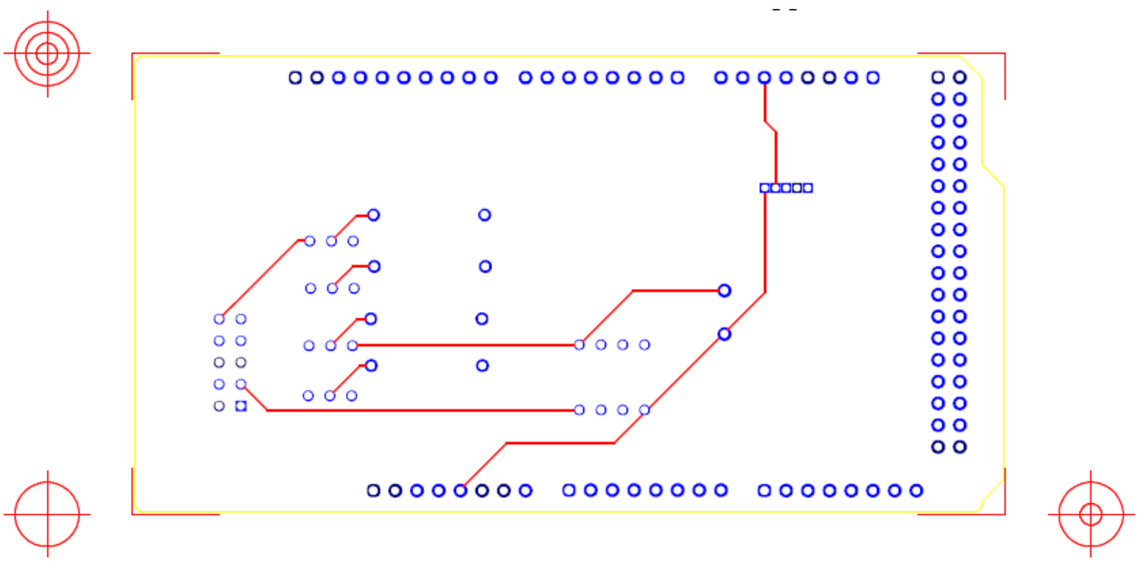


F.3 Ground Station




ARDUINO MEGA 2560
R1
R2
R4
R3
AREF
GND
IOREF
RESET
3.3V
5V
GND
GND
VIN
SCK
GND





Appendix G

Payload Movement Simulator

The sensor used to get information about the payload movements was the LSM9DS1, a nine degrees of freedom (9DoF) motion-sensing system in a single chip. It contains a 3-axis accelerometer, 3-axis gyroscope, and a 3-axis magnetometer. The data can be accessed through I2C or SPI communication, also used to configure the different scales and ranges of the aforementioned sensors:

- **Accelerometer:** it measures the payload acceleration in g's, with a scale that can be set to ±2, 4, 8 or 16 g.
- **Gyroscope:** it measures the angular velocity in degrees per second (DPS) of the payload with a scale that can be set to ± 245, 500 or 2000 DPS.

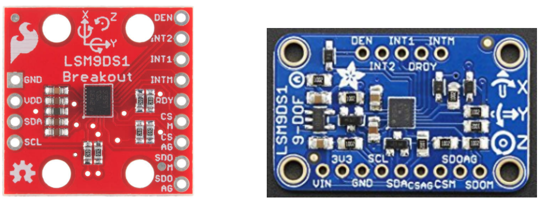


Figure G.1: 9 Degrees of Freedom - LSM8DS1 (L) Sparkfun, (R) Adafruit Modules

A system consisting of an Arduino UNO board and a LSM9DS1 sensor was created to simulate and understand the payload movements during a HAB launch:

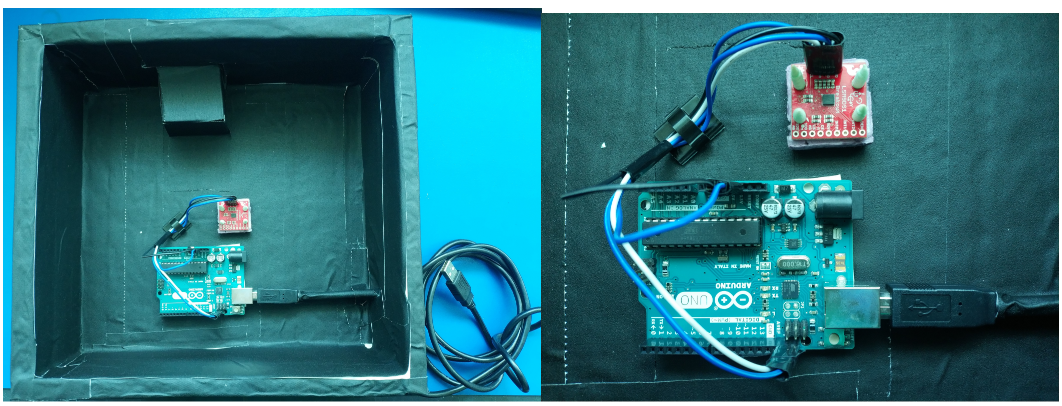


Figure G.2: LSM8DS1 Sensor: (L) Sparkfun, (R) Adafruit Modules

As it can be seen in the previous figures, the hardware is placed inside a box similar to the ones used during the HAB launches, with only one cable connection required to get the data. For this simulator, the I2C connection to the sensor was chosen, requiring only 4 connections to cover the system's power supply and data transfer.

The system box can be hanging from a certain altitude to be dropped to simulate controlled descent unit cases, balloon bursts, double and single balloon configurations lifting the payload, among others. On the other hand, the whole box can be placed in a controlled environment, where the three axis are controlled with motors moving with a certain acceleration and at different angles in order to calibrate the sensors and to analyse the movements during the launch with more precision.

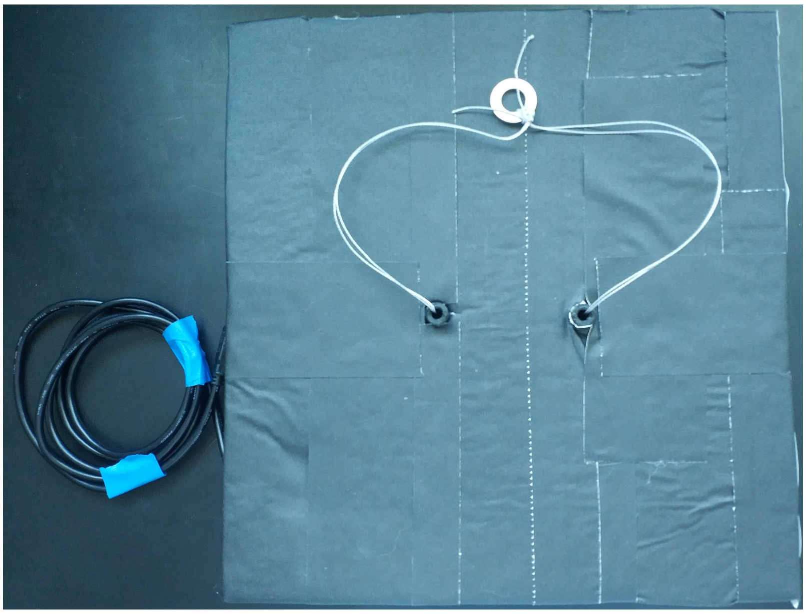


Figure G.3: Movement Simulator System Box

As aforementioned, only one cable connection to this system is required. The Arduino serial cable is directly connected to a computer, where two different code modes can be used to get and plot the sensor data in real-time:

- **Arduino Mode:** using the Arduino IDE, the embedded serial plotter can be used to plot the desired signals in real-time with the specified data points per second.
- **MATLAB Mode:** running a second MATLAB code, the accelerometer and the gyroscope data can be plotted separately.

The Arduino code used for both modes is presented below:

```
/*****************************************************************
 * NAME: 9DoF_plotter.ino
 * AUTHOR: Noemi Miguelez Gomez
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Movement Plotter.
 *
 *
 * DEVELOPMENT HISTORY:
 *   Date     Author  Version            Description Of Change
 * --------  ------  -------  ------------------------------------
 * 07/17/2019   NMG      1.1     Code adapted to MATLAB/Arduino and
                                 LSM9DS1.
 ****************************************************************/

#include <Wire.h>
#include <SPI.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>  // not used in this demo but required!

// i2c
Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1();
byte imuPacket[12];
sensors_event_t a, m, g, temp;

int accel_x;
int accel_y;
int accel_z;
int gyro_x;
int gyro_y;
int gyro_z;

int measPS = 50; //Sensor sampling rate.


void setupSensor()
{
  // 1.) Set the accelerometer range
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_4G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_8G);
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_16G);

  // 3.) Setup the gyroscope
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_245DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_500DPS);
  lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_2000DPS);
}

void setup()
{
  Serial.begin(115200);
  while (!Serial) {
    delay(1);
  }
    if (!lsm.begin())
  {
    while (1);
  }
  setupSensor();
}

void loop()
{
  lsm.getEvent(&a, &m, &g, &temp);

  accel_x = a.acceleration.x;
  accel_y = a.acceleration.y;
  accel_z = a.acceleration.z;

  gyro_x = g.gyro.x;
  gyro_y = g.gyro.y;
  gyro_z = g.gyro.z;

  /**********ARDUINO MODE***********/
  //Select/Uncomment the signals to plot on the Serial Plotter.
  Serial.print(float(accel_x)/1000); Serial.print(" ");

  Serial.print(float(accel_y)/1000); Serial.print(" ");

  Serial.println(float(accel_z)/1000); Serial.print(" ");

//  Serial.print(float(gyro_x)/1000); Serial.print(" ");
//
//  Serial.print(float(gyro_y)/1000); Serial.print(" ");
//
//  Serial.print(float(gyro_z)/1000); Serial.print(" ");

  Serial.println("uT");

 /***********MATLAB MODE***********/
//  imuPacket[0] = accel_x;
//  imuPacket[1] = accel_x >> 8;
//
//  imuPacket[2] = accel_y;
//  imuPacket[3] = accel_y >> 8;
//
//  imuPacket[4] = accel_z;
//  imuPacket[5] = accel_z >> 8;
```

```
//
//   imuPacket[6] = gyro_x;
//   imuPacket[7] = gyro_x >> 8;
//
//   imuPacket[8] = gyro_y;
//   imuPacket[9] = gyro_y >> 8;
//
//   imuPacket[10] = gyro_z;
//   imuPacket[11] = gyro_z >> 8;
//
//   Serial.write(imuPacket, 12);
 /*************************************/

   delay(1000/measPS);
}
```

The expected results from the Arduino plotter mode can be seen in the next figures:

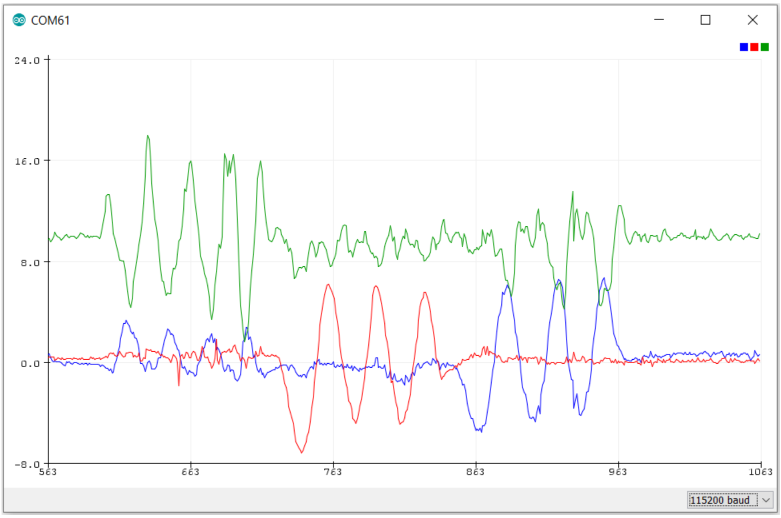


Figure G.4: Arduino IDE - Movement Simulator Acceleration Data.

For the previous case, the box was moved vertically at the beginning, then horizontally in Y direction, horizontally in X direction and not moving at all at the end. The data plotted presents the expected values, were the colors are automatically assigned by the IDE in order of printing: Blue=X, Red=Y, Green=Z.

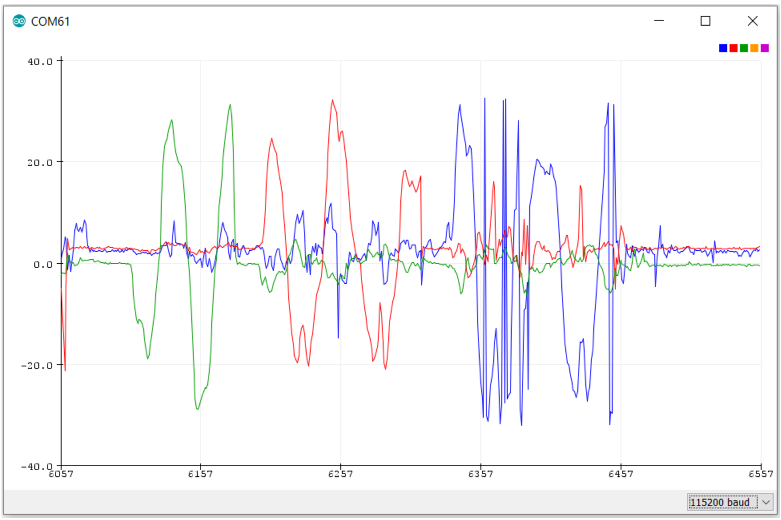


Figure G.5: Arduino IDE - Movement Simulator Angular Velocity Data.

To obtain the gyroscope data, the center of the payload was intended to be in the same point, while it was rotated in the different axis. We can see in this case a movement in Z, Y and X axis, in order, and no movement at the end.

For the MATLAB mode, the code used is the following one:

```
clear all;
close all;
fclose('all');
delete(instrfind);
%% Create objects and establish connections
duration = 10; %[mins]

%Serial for the board comms
board = serial('COM61');
%Set serial parameters
board.InputBufferSize = 500000;
set(board, 'DataBits', 8);
set(board, 'StopBits', 1);
set(board, 'BaudRate', 115200);
set(board, 'Parity', 'none');

%Open the serial port
try
    fopen(board);
catch err
    fclose(board);
    warndlg('Board connection error.');
end

figure
h1 = animatedline('Color', 'r');
h2 = animatedline('Color', 'g');
h3 = animatedline('Color', 'b');
ax1 = gca;
ax1.YGrid = 'on';
ax1.YLim = [-20 20];
xlabel('Time')
ylabel('Acceleration [m/s^2]')
legend('X', 'Y', 'Z');

figure
h4 = animatedline('Color', 'r');
h5 = animatedline('Color', 'g');
h6 = animatedline('Color', 'b');
ax2 = gca;
ax2.YGrid = 'on';
ax2.YLim = [-40 40];
startTime = datetime('now');
xlabel('Time')
ylabel('Angular Speed [dps]')
legend('X', 'Y', 'Z');

tic
pause(1);
while toc<(duration*60)
   if (board.BytesAvailable>60)
            readData = fread(board,60);
            %Accelerometer Monitor
            accel_X = typecast(uint8(readData(1:2)),'int16');
            accel_x = double(accel_X)/1000;

            accel_Y = typecast(uint8(readData(3:4)),'int16');
            accel_y = double(accel_Y)/1000;

            accel_Z = typecast(uint8(readData(5:6)),'int16');
            accel_z = double(accel_Z)/1000;

            %Gyroscope Monitor
            gyro_X = typecast(uint8(readData(7:8)),'int16');
            gyro_x = double(gyro_X)/1000;

            gyro_Y = typecast(uint8(readData(9:10)),'int16');
            gyro_y = double(gyro_Y)/1000;

            gyro_Z = typecast(uint8(readData(11:12)),'int16');
            gyro_z = double(gyro_Z)/1000;


            % Get current time
            t =  datetime('now') - startTime;
            % Add points to animation
            addpoints(h1,datenum(t),accel_x)
            addpoints(h2,datenum(t),accel_y)
            addpoints(h3,datenum(t),accel_z)
            % Update axes
            ax1.XLim = datenum([t-seconds(30) t]);
            datetick('x','keeplimits')

            % Get current time
            t =  datetime('now') - startTime;
            % Add points to animation
            addpoints(h4,datenum(t),gyro_x)
            addpoints(h5,datenum(t),gyro_y)
```

```
            addpoints(h6,datenum(t),gyro_z)
            % Update axes
            ax2.XLim = datenum([t-seconds(30) t]);
            datetick('x','keeplimits')
            drawnow
    end
end
```

The expected results from this mode are presented in the following figures:

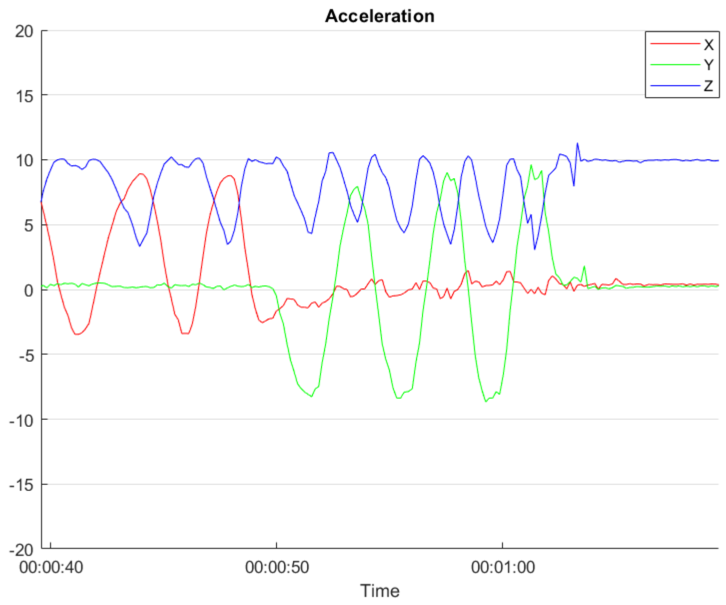


Figure G.6: MATLAB - Movement Simulator Acceleration Data.

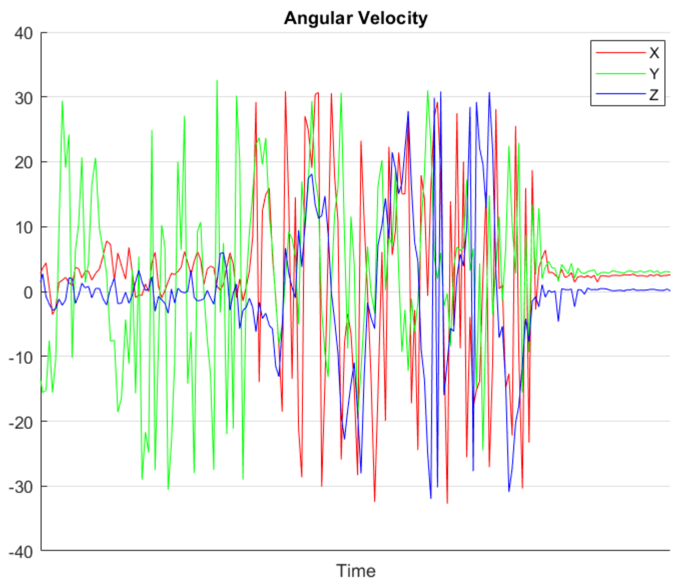


Figure G.7: MATLAB - Movement Simulator Angular Velocity Data.

In this case, the accelerometer and the gyroscope data can be plotted for the same movement in two different plots, which can result more useful than a single plot for the Arduino IDE with all the data on it.

Appendix H

Payload Codes and Flow Diagrams

H.1 Payload With Internal CDU

```
/*************************************************************
 * NAME: HAB_Transmitter.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Transmitter.
 *
 * DEVELOPMENT HISTORY:
 *   Date     Author  Version            Description Of Change
 * --------  ------  -------  -----------------------------------
 * 02/23/2018   NMG     1.1     Scientific packets included and
                                sync.
 * 03/02/2018   NMG     1.2     GPS sensors included.
 * 03/20/2018   NMG     1.3     Cutting System Included.
 * 06/04/2018   NMG     1.4     SD Card System Included.
 * 06/08/2018   NMG     1.5     Watchdog Timer Included.
 * 07/20/2018   NMG     2.1     Data Packet and SD File Changes.
 * 08/09/2018   NMG     2.2     Scientific Packets With GPS Info.
 * 10/10/2018   NMG     3       Packets Structure Changes.
 * 01/12/2018   NMG     3.1     Cutting System I2C-SPI Changes.
 * 01/15/2018   NMG     4       Code Adapted to Teensy 3.5.
 * 01/15/2018   NMG     4.1     Code Restructured.
 *************************************************************/

#include <NMEAGPS.h>
#include <GPSport.h>
#include <Streamers.h>
#include <EEPROM.h>
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <avr/wdt.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>


/********************DATA BACKUP*************************/
int32_t dataTimeTh[7] = {3600000, 7200000, 10800000, 14400000, 18000000, 21600000, 25200000};
int32_t dataAltTh[7] = {10000, 20000, 30000, 40000, 30000, 20000, 10000};

int dataCount;

File flightData;
String fileN;
String fileName;
int sdFlag;
int32_t timeSD;
unsigned int fileNum;
unsigned int address;
int sdCount;

/***************CUTTING SYSTEM******************/
//ALTITUDE THRESHOLD FOR CUTTING SYSTEM
#define CUTTING_THRES 30000
#define CUTTING_TIME 12000
#define CUT_ENABLE 39
volatile int cutting_flag;
volatile int cutting_finished;
int altCnt;


/*******************************GPS SENSORS**********************************/
/******SERIAL DEFINITIONS*******
* RADIO_TX           Serial_0 *
* GPS UBlox External  Serial_3 *
```

```
*******************************/

static NMEAGPS  uBloxEX;   //uBlox GPS
static gps_fix  uBloxEXFix;

/*****************RADIO PACKETS AND SENSORS VARIABLES*************************/
byte id_sci[2] = {0xA0, 0xB1}; //Identifier for scientific data packet.
byte id_gps[2] = {0xC0, 0xD1};

byte sciPacket[100];           //Scientific data packet byte array.
byte gpsPacket[100];           //GPS data packet byte array.
unsigned int packet_number;             //Packet number/counter.

//ANALOG PINS DEFINITION
#define TEMP_EXT A9
#define TEMP_INT A8
#define VOLTAGE A12

int tempExt;                   //External Temperature.
int tempInt;                   //Internal Temperature.
int voltage;                   //Voltage Monitor (VBat).

//9DoF PINS DEFINITION
#define LSM9DS1_MISO 50
#define LSM9DS1_MOSI 51
#define LSM9DS1_SCK 52
#define LSM9DS1_XGCS 43
#define LSM9DS1_MCS 45

Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1(LSM9DS1_XGCS, LSM9DS1_MCS);
sensors_event_t a, m, g, temp;
int accel_x;
int accel_y;
int accel_z;

int gyro_x;
int gyro_y;
int gyro_z;

/********************GPS DATA*********************/
int32_t lat;     //Latitude

int32_t lon;     //Longitude

int32_t alt1;    //Altitude x.0
int16_t alt2;    //Altitude 0.x

byte stat;       //Status

uint8_t numSats; //Number of Satellites in View
uint8_t utcHour; //UTC Time - Hour
uint8_t utcMin;  //UTC Time - Minutes
uint8_t utcSec;  //UTC Time - Seconds

/*****************TIMERS*************************/
unsigned long time_ref; //Reference Time (computed after a packet transmission)
unsigned long time_gps; //Last GPS Time (updated once a GPS packet is transmitted)

unsigned long time_cutting1;
unsigned long time_cutting2;

unsigned long time_packet;


void setupSensor()
{
  // 1.) Set the accelerometer range
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_4G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_8G);
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_16G);

  // 2.) Set the magnetometer sensitivity
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_4GAUSS);
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_8GAUSS);
  //lsm.setupMag(lsm.LSM9DS1_MAGGAIN_12GAUSS);
  lsm.setupMag(lsm.LSM9DS1_MAGGAIN_16GAUSS);

  // 3.) Setup the gyroscope
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_245DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_500DPS);
  lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_2000DPS);
}

void setup() {
  wdt_disable();
  /************IMU SENSOR SETUP**********/

  lsm.begin();
  setupSensor();

  /************RADIO COMMS INI**********/
  Serial.begin(230400);

  /************SD CARD**************/
  sdCount = 0;
  pinMode(41, OUTPUT);
  digitalWrite(41, HIGH);
  while ((!SD.begin(41))&&(sdCount < 3)){
```

```
    sdCount = sdCount + 1;
    delay(1000);
  }

  address = 0;
  fileNum = EEPROM.read(address);
  fileNum = fileNum + 1;
  EEPROM.write(address, fileNum);

  fileN =  "data";
  String m = fileN + fileNum;
  String ext = ".txt";
  fileName = m + ext;

  flightData = SD.open(fileName, FILE_WRITE);
  timeSD = millis();
  dataCount = 0;

  /************GPS SERIALS**********/
  gpsuBloxExt.begin(9600);

  /*********CUTTING SYSTEM*********/
  pinMode(CUT_ENABLE, OUTPUT);
  digitalWrite(CUT_ENABLE, LOW);

  time_cutting1 = 0;
  time_cutting2 = 0;
  cutting_flag = 0;
  cutting_finished = 0;
  packet_number=0;

  /*********WATCHDOG TIMER*********/
  wdt_enable(WDTO_500MS);
}


void send_sci_packet(int id)
{
    packet_number++;
    sciPacket[2] = packet_number;
    sciPacket[3] = packet_number >> 8;

    if (id==3){
      //prepare_fix_packet();  //CUTTING SYSTEM

      sciPacket[0] = id_gps[0];
      sciPacket[1] = id_gps[1];

      sciPacket[4] = lat;
      sciPacket[5] = lat >> 8;
      sciPacket[6] = lat >> 16;
      sciPacket[7] = lat >> 24;

      sciPacket[8] = lon;
      sciPacket[9] = lon >> 8;
      sciPacket[10] = lon >> 16;
      sciPacket[11] = lon >> 24;

      sciPacket[12] = alt1;
      sciPacket[13] = alt1 >> 8;
      sciPacket[14] = alt1 >> 16;
      sciPacket[15] = alt1 >> 24;

      sciPacket[16] = stat;

      sciPacket[17] = numSats;

      sciPacket[18] = utcHour;
      sciPacket[19] = utcMin;
      sciPacket[20] = utcSec;

      sciPacket[21] = alt2;
      sciPacket[22] = alt2 >> 8;

      tempExt = analogRead(TEMP_EXT);
      sciPacket[23] = tempExt;
      sciPacket[24] = tempExt >> 8;

      time_packet = millis();
      sciPacket[26] = time_packet;
      sciPacket[27] = time_packet >> 8;
      sciPacket[28] = time_packet >> 16;
      sciPacket[29] = time_packet >> 24;
    }
    else {
      sciPacket[0] = id_sci[0];
      sciPacket[1] = id_sci[1];

      tempExt = analogRead(TEMP_EXT);
      sciPacket[4] = tempExt;
      sciPacket[5] = tempExt >> 8;

      tempInt = analogRead(TEMP_INT);
      sciPacket[6] = tempInt;
      sciPacket[7] = tempInt  >> 8;

      tempExt = analogRead(TEMP_EXT);
      tempInt = analogRead(TEMP_INT);
      sciPacket[8] = tempExt;
```

```
      sciPacket[9] = tempExt >> 8;
      sciPacket[10] = tempInt;
      sciPacket[11] = tempInt >> 8;

      voltage = analogRead(VOLTAGE);
      sciPacket[12] = voltage;
      sciPacket[13] = voltage >> 8;


      lsm.getEvent(&a, &m, &g, &temp);
      accel_x = a.acceleration.x;

      sciPacket[14] = accel_x;
      sciPacket[15] = accel_x >> 8;

      accel_y = a.acceleration.y;
      sciPacket[16] = accel_y;
      sciPacket[17] = accel_y >> 8;

      accel_z = a.acceleration.z;
      sciPacket[18] = accel_z;
      sciPacket[19] = accel_z >> 8;

      gyro_x = g.gyro.x;
      sciPacket[20] = gyro_x;
      sciPacket[21] = gyro_x >> 8;

      gyro_y = g.gyro.y;
      sciPacket[22] = gyro_y;
      sciPacket[23] = gyro_y >> 8;

      gyro_z = g.gyro.z;
      sciPacket[24] = gyro_z;
      sciPacket[25] = gyro_z >> 8;

      time_packet = millis();
      sciPacket[26] = time_packet;
      sciPacket[27] = time_packet >> 8;
      sciPacket[28] = time_packet >> 16;
      sciPacket[29] = time_packet >> 24;
    }

    for(int i=0; i<3; i++)
    {
        tempExt = analogRead(TEMP_EXT);
        sciPacket[30+22*i] = tempExt;
        sciPacket[30+22*i+1] = tempExt >> 8;

        tempInt = analogRead(TEMP_EXT);
        sciPacket[30+22*i+2] = tempInt;
        sciPacket[30+22*i+3] = tempInt  >> 8;

        voltage = analogRead(VOLTAGE);
        sciPacket[30+22*i+4] = voltage;
        sciPacket[30+22*i+5] = voltage >> 8;

        accel_x = a.acceleration.x;
        sciPacket[30+22*i+6] = accel_x;
        sciPacket[30+22*i+7] = accel_x >> 8;

        accel_y = a.acceleration.y;
        sciPacket[30+22*i+8] = accel_y;
        sciPacket[30+22*i+9] = accel_y >> 8;

        accel_z = a.acceleration.z;
        sciPacket[30+22*i+10] = accel_z;
        sciPacket[30+22*i+11] = accel_z >> 8;

        gyro_x = g.gyro.x;
        sciPacket[30+22*i+12] = gyro_x;
        sciPacket[30+22*i+13] = gyro_x >> 8;

        gyro_y = g.gyro.y;
        sciPacket[30+22*i+14] = gyro_y;
        sciPacket[30+22*i+15] = gyro_y >> 8;

        gyro_z = g.gyro.z;
        sciPacket[30+22*i+16] = gyro_z;
        sciPacket[30+22*i+17] = gyro_z >> 8;

        tempExt = analogRead(TEMP_EXT);
        sciPacket[[30+22*i+18] = tempExt;
        sciPacket[[30+22*i+19] = tempExt >> 8;

        tempExt = analogRead(TEMP_EXT);
        sciPacket[[30+22*i+20] = tempExt;
        sciPacket[[30+22*i+21] = tempExt >> 8;
    }

    Serial.write(sciPacket, 100);
    flightData.write(sciPacket, 30);
    delay(2);
}


static void prepare_fix_packet(){
  if (cutting_flag == 0){
    if (alt1 > CUTTING_THRES){
      digitalWrite(CUT_ENABLE, HIGH);
```

```
      time_cutting1 = millis();
      cutting_flag = 1;
    }
  }

  if ((cutting_flag == 1)&&(cutting_finished == 0)){
    time_cutting2 = millis();
    if ((time_cutting2-time_cutting1)> CUTTING_TIME){
     digitalWrite(CUT_ENABLE, LOW);
     cutting_finished = 1;
    }
  }
}



void loop() {
  int id = 0;
  while (uBloxEX.available( gpsuBloxExt )) {
    uBloxEXFix = uBloxEX.read();
    if (uBloxEXFix.valid.location) {
      lat = uBloxEXFix.latitudeL();
      lon = uBloxEXFix.longitudeL();
      alt1 = uBloxEXFix.alt.whole;
      alt2 = uBloxEXFix.alt.frac;
      stat = uBloxEXFix.status;
      numSats = uBloxEXFix.satellites;
      utcHour = uBloxEXFix.dateTime.hours;
      utcMin = uBloxEXFix.dateTime.minutes;
      utcSec = uBloxEXFix.dateTime.seconds;

      id = 3;
    }
  }
    send_sci_packet(id);
    wdt_reset();

  if (flightData) {
    if (sdFlag == 0){
      if (((millis() - timeSD) > dataTimeTh[dataCount])||(alt1 > dataAltTh[dataCount])){
        dataCount = dataCount + 1;
        flightData.flush();
      }
      if (dataCount == 7){
        sdFlag = 1;
      }
      wdt_reset();
    }
    if (sdFlag == 1){
      flightData.close();
      sdFlag = 2;
      wdt_reset();
    }
  }
}
```

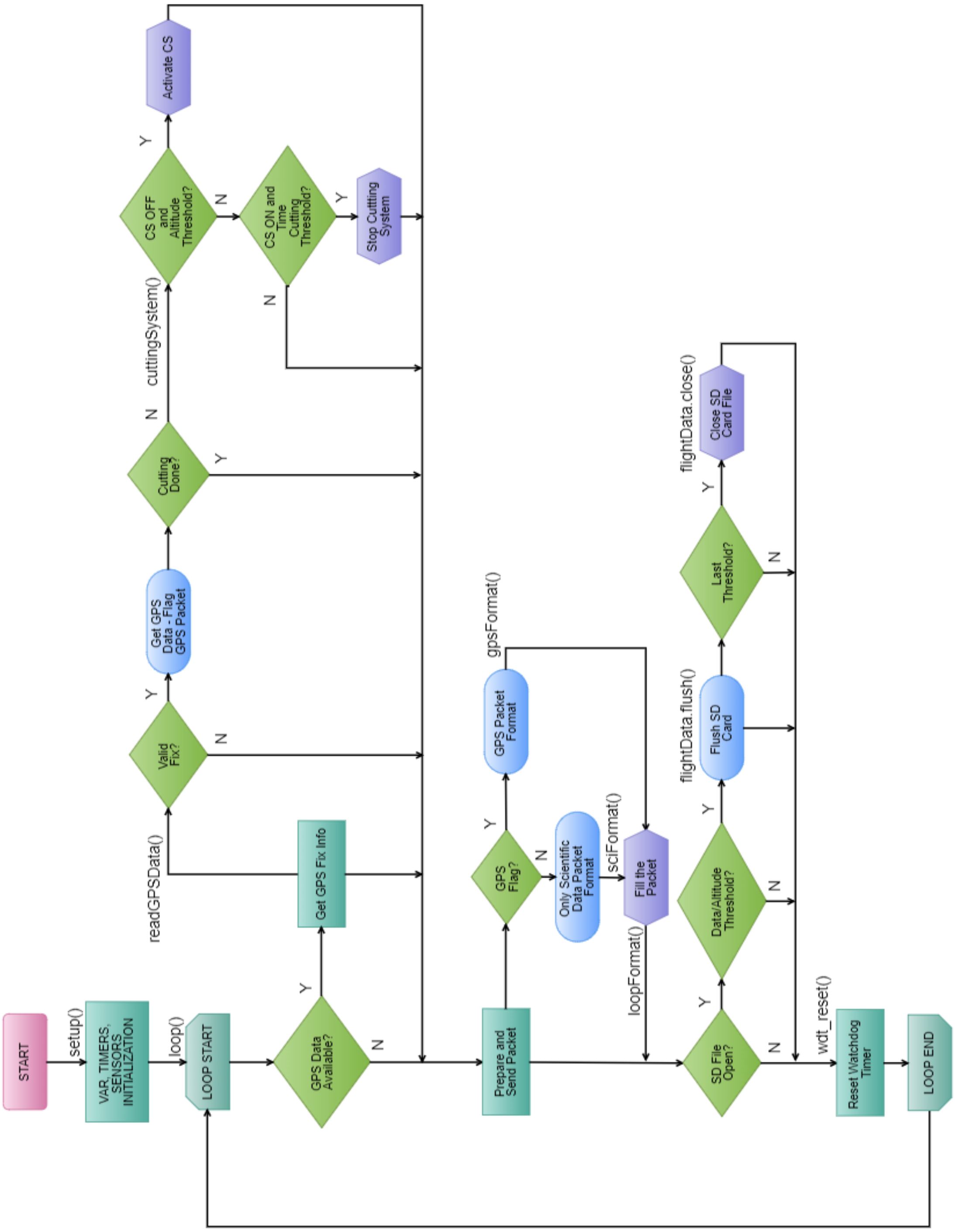


Figure H.1: Design 1-2 Software Flow Diagram - Payload with Internal Cutting System for a Controlled Descent.

H.2 Payload with Retransmissions

```
/*****************************************************************
 * NAME: HAB_Transmitter.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Transmitter.
 *
 * DEVELOPMENT HISTORY:
 *   Date     Author  Version            Description Of Change
 * --------  ------  -------  ------------------------------------
 * 02/23/2018   NMG     1.1     Scientific packets included and
                                sync.
 * 03/02/2018   NMG     1.2     GPS sensors included.
 * 03/20/2018   NMG     1.3     Cutting System Included.
 * 06/04/2018   NMG     1.4     SD Card System Included.
 * 06/08/2018   NMG     1.5     Watchdog Timer Included.
 * 07/20/2018   NMG     2.1     Data Packet and SD File Changes.
 * 08/09/2018   NMG     2.2     Scientific Packets With GPS Info.
 * 10/10/2018   NMG     3       Packets Structure Changes.
 * 01/12/2018   NMG     3.1     Cutting System I2C-SPI Changes.
 * 01/15/2018   NMG     4       Code Adapted to Teensy 3.5.
 * 01/15/2018   NMG     4.1     Code Restructured.
 * 03/09/2018   NMG     4.2     Re-send data implementation.
 * 03/13/2018   NMG     4.3     External Watchdog Timer Included.
 * 05/10/2019   NMG     4.4     GPS packets not considered.
 * 07/02/2019   NMG     4.5     Teensy ADC Reference - Flow Control.
 ****************************************************************/

#include <NMEAGPS.h>
#include <GPSport.h>
#include <Streamers.h>
#include <EEPROM.h>
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <avr/wdt.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>
#include <ADC.h>
ADC *adc = new ADC();

/******SERIAL DEFINITIONS*******
* RADIO_TX            Serial_0 *
* GPS UBlox External  Serial_3 *
********************************/

/************************DATA BACKUP***************************/
File flightData;
String ext = ".bin";
String fileN;
String fileName1;
String fileName2;
int sdFlag;
int32_t timeSD;
unsigned int fileNum;
unsigned int address;
int sdCount;
const int chipSelect = BUILTIN_SDCARD;

/*********************CUTTING SYSTEM***************************/
////ALTITUDE THRESHOLD FOR CUTTING SYSTEM
//#define CUTTING_THRES 30000
//#define CUTTING_TIME 12000
//volatile int cutting_flag;
//volatile int cutting_finished;


/*************************GPS SENSORS**************************/
static NMEAGPS  uBloxEX;    //uBlox GPS
static gps_fix  uBloxEXFix;
int id;

int32_t lat1;      //Latitude
int32_t lon;      //Longitude
int32_t alt;      //Altitude
int32_t highAlt;
byte stat;        //Status
uint8_t numSats; //Number of Satellites in View
uint8_t utcHour; //UTC Time - Hour
uint8_t utcMin;  //UTC Time - Minutes
uint8_t utcSec;  //UTC Time - Seconds

/**************DATA PACKETS AND SENSORS VARIABLES***************/
byte id_sci[2] = {0xA0, 0xB1}; //Identifier for only scientific data packet.
byte id_gps[2] = {0xC0, 0xD1}; //Identifier for packet with GPS data.

byte sciPacket[100];           //Scientific data packet byte array.
unsigned int packet_number;    //Packet number/counter.
unsigned long time_packet;     //Packet Timestamp.

//ANALOG PINS DEFINITION
//PAYLOAD PCB
//#define TEMP_EXT_H A9
//#define TEMP_EXT_L A8
//#define TEMP_INT A7
#define CTS_PIN 24
```

```
//--------------------

////PAYLOAD 2
#define TEMP_EXT_H A9
#define TEMP_INT A8
#define TEMP_EXT_L A7
//--------------------
#define VOLTAGE A6

int xbeePower;
int tempExt_l;                    //External Temperature.
int tempExt_h;
int tempInt;                    //Internal Temperature.
int voltage;                    //Voltage Monitor (VBat).

//9DoF PINS DEFINITION
#define LSM9DS1_XGCS 10
#define LSM9DS1_MCS 9

//9DoF VARIABLES DEFINITION
Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1(LSM9DS1_XGCS, LSM9DS1_MCS);
sensors_event_t a, m, g, temp;
int accel_x;
int accel_y;
int accel_z;

int gyro_x;
int gyro_y;
int gyro_z;


int altFlag;
int altCnt;
unsigned long altDscntCnt;
int kmPos;
int altTest;
int testFlag;
int altEEPROM;

void setupSensor()
{
  // 1.) Set the accelerometer range
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_4G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_8G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_16G);

  // 2.) Setup the gyroscope
  lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_245DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_500DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_2000DPS);
}

void setup() {
  /********************IMU SENSOR SETUP******************/

  lsm.begin();
  setupSensor();

  analogReadResolution(10);
  //adc->setAveraging(8);
  adc->setReference(ADC_REFERENCE::REF_3V3, ADC_0);
  //adc->setConversionSpeed(ADC_CONVERSION_SPEED::LOW_SPEED);
  Serial5.attachCts(24);
  pinMode(24, INPUT);

  /********************RADIO COMMS INI*******************/
  Serial5.begin(230400);
  id = 0;
  /***********************SD CARD***********************/
 sdCount = 0;
  while ((!SD.begin(chipSelect)) && (sdCount < 3)) {
    sdCount = sdCount + 1;
    delay(1000);
  }
  //Read and set file number and name
  address = 0;
  fileNum = EEPROM.read(address);
  fileNum = fileNum + 1;
  EEPROM.write(address, fileNum);
  fileN =  "data";
  String m1 = fileN + fileNum;
  fileName1 = m1 + ext;

  //Open file
  flightData = SD.open(fileName1.c_str(), FILE_WRITE);
  timeSD = millis();

  //Prepare the next file for the descent
  fileN =  "dscnt";
  String m2 = fileN + fileNum;
  fileName2 = m2 + ext;


  /********************GPS SERIALS*******************/
  gpsuBloxExt.begin(9600);
  packet_number = 0;
  highAlt = 0;
  altFlag = 0;
```

```
  altTest = 0;
  altCnt = 0;
  altDscntCnt = 0;
  testFlag = 0;
  alt = 0;

/********************WATCHDOG TIMER*****************/
   noInterrupts();
   WDOG_UNLOCK = WDOG_UNLOCK_SEQ1;
   WDOG_UNLOCK = WDOG_UNLOCK_SEQ2;
   delayMicroseconds(1);


   WDOG_TOVALH = 0x006d;
   WDOG_TOVALL = 0xdd00;

   WDOG_PRESC  = 0x400;

   WDOG_STCTRLH |= WDOG_STCTRLH_ALLOWUPDATE |
   WDOG_STCTRLH_WDOGEN | WDOG_STCTRLH_WAITEN |
   WDOG_STCTRLH_STOPEN | WDOG_STCTRLH_CLKSRC;
   interrupts();
}


void send_sci_packet(int id)
{
    if (id==3){

      sciPacket[0] = id_gps[0];
      sciPacket[1] = id_gps[1];

      sciPacket[2] = packet_number;
      sciPacket[3] = packet_number >> 8;

      sciPacket[4] = lat1;
      sciPacket[5] = lat1 >> 8;
      sciPacket[6] = lat1 >> 16;
      sciPacket[7] = lat1 >> 24;

      sciPacket[8] = lon;
      sciPacket[9] = lon >> 8;
      sciPacket[10] = lon >> 16;
      sciPacket[11] = lon >> 24;

      sciPacket[12] = alt;
      sciPacket[13] = alt >> 8;
      sciPacket[14] = alt >> 16;
      sciPacket[15] = alt >> 24;

      sciPacket[16] = stat;

      sciPacket[17] = numSats;

      sciPacket[18] = utcHour;
      sciPacket[19] = utcMin;
      sciPacket[20] = utcSec;

      /*******EXTRA TEMP DATA*******/
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[21] = tempExt_h;
      sciPacket[22] = tempExt_h >> 8;

      tempExt_l = analogRead(TEMP_EXT_L);
      sciPacket[23] = tempExt_l;
      sciPacket[24] = tempExt_l >> 8;

    }
    else {
      sciPacket[0] = id_sci[0];
      sciPacket[1] = id_sci[1];

      packet_number++;
      sciPacket[2] = packet_number;
      sciPacket[3] = packet_number >> 8;

      adc->setConversionSpeed(ADC_CONVERSION_SPEED::LOW_SPEED);
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[4] = tempExt_h;
      sciPacket[5] = tempExt_h >> 8;

      tempInt = analogRead(TEMP_INT);
      sciPacket[6] = tempInt;
      sciPacket[7] = tempInt  >> 8;


      tempExt_l = analogRead(TEMP_EXT_L);
      sciPacket[8] = tempExt_l;
      sciPacket[9] = tempExt_l >> 8;

      adc->setConversionSpeed(ADC_CONVERSION_SPEED::MED_SPEED);
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[10] = tempExt_h;
      sciPacket[11] = tempExt_h >> 8;

      voltage = analogRead(VOLTAGE);
      sciPacket[12] = voltage;
      sciPacket[13] = voltage >> 8;

```

```
    lsm.getEvent(&a, &m, &g, &temp);

    accel_x = a.acceleration.x;
    sciPacket[14] = accel_x;
    sciPacket[15] = accel_x >> 8;

    accel_y = a.acceleration.y;
    sciPacket[16] = accel_y;
    sciPacket[17] = accel_y >> 8;

    accel_z = a.acceleration.z;
    sciPacket[18] = accel_z;
    sciPacket[19] = accel_z >> 8;

    gyro_x = g.gyro.x;
    sciPacket[20] = gyro_x;
    sciPacket[21] = gyro_x >> 8;

    gyro_y = g.gyro.y;
    sciPacket[22] = gyro_y;
    sciPacket[23] = gyro_y >> 8;

    gyro_z = g.gyro.z;
    sciPacket[24] = gyro_z;
    sciPacket[25] = gyro_z >> 8;
  }

  for(int i=0; i<3; i++)
  {
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[30+22*i] = tempExt_h;
      sciPacket[30+22*i+1] = tempExt_h >> 8;

      tempExt_l = analogRead(TEMP_EXT_L);
      sciPacket[30+22*i+2] = tempExt_l;
      sciPacket[30+22*i+3] = tempExt_l  >> 8;

      voltage = analogRead(VOLTAGE);
      sciPacket[30+22*i+4] = voltage;
      sciPacket[30+22*i+5] = voltage >> 8;

      accel_x = a.acceleration.x;
      sciPacket[30+22*i+6] = accel_x;
      sciPacket[30+22*i+7] = accel_x >> 8;

      accel_y = a.acceleration.y;
      sciPacket[30+22*i+8] = accel_y;
      sciPacket[30+22*i+9] = accel_y >> 8;

      accel_z = a.acceleration.z;
      sciPacket[30+22*i+10] = accel_z;
      sciPacket[30+22*i+11] = accel_z >> 8;

      gyro_x = g.gyro.x;
      sciPacket[30+22*i+12] = gyro_x;
      sciPacket[30+22*i+13] = gyro_x >> 8;

      gyro_y = g.gyro.y;
      sciPacket[30+22*i+14] = gyro_y;
      sciPacket[30+22*i+15] = gyro_y >> 8;

      gyro_z = g.gyro.z;
      sciPacket[30+22*i+16] = gyro_z;
      sciPacket[30+22*i+17] = gyro_z >> 8;

      /**EXTRA EXTERNAL TEMP READINGS**/
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[30+22*i+18] = tempExt_h;
      sciPacket[30+22*i+19] = tempExt_h >> 8;

      tempExt_l = analogRead(TEMP_EXT_L);
      sciPacket[30+22*i+20] = tempExt_l;
      sciPacket[30+22*i+21] = tempExt_l >> 8;
  }
      /**EXTRA TEMP READINGS**/
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[96] = tempExt_h;
      sciPacket[97] = tempExt_h >> 8;

      tempInt = analogRead(TEMP_INT);
      sciPacket[98] = tempInt;
      sciPacket[99] = tempInt >> 8;

      while (digitalRead(CTS_PIN)==1){
        delayMicroseconds(1);
      }
       /****PACKET TIMESTAMP****/
      time_packet = millis();
      sciPacket[26] = time_packet;
      sciPacket[27] = time_packet >> 8;
      sciPacket[28] = time_packet >> 16;
      sciPacket[29] = time_packet >> 24;
      Serial5.write(sciPacket, 100);
      //delay(3);
}


void loop() {
```

```
    while (uBloxEX.available( gpsuBloxExt )) {
      uBloxEXFix = uBloxEX.read();
      if (uBloxEXFix.valid.location) {
        lat1 = uBloxEXFix.latitudeL();          // Scaled by 10,000,000
        lon = uBloxEXFix.longitudeL();         // Scaled by 10,000,000
        alt = uBloxEXFix.altitude_cm();
        stat = uBloxEXFix.status;
        numSats = uBloxEX.sat_count;
        utcHour = uBloxEXFix.dateTime.hours;
        utcMin = uBloxEXFix.dateTime.minutes;
        utcSec = uBloxEXFix.dateTime.seconds;

        send_sci_packet(3);
      }
    }
    if (altFlag < 2){
        send_sci_packet(0);
        flightData.write(sciPacket, 100);

        if (alt > highAlt){
          highAlt = alt;
          altDscntCnt = millis();
        }
        else if ((altFlag == 0)&&((highAlt-10000-alt) > 0)){  //Change to 10000 after lab tests!
          if ((millis() - altDscntCnt) > 120000) {
            flightData.close();
            flightData = SD.open(fileName2.c_str(), FILE_WRITE);
            timeSD = millis();
            kmPos = flightData.position();
            altFlag = 1;
          }
        }
        else if ((altFlag == 1)&&(alt < 2000000)){ //ALTITUDE IN CM!
              altFlag = 2;
          }
    }

    else if (altFlag == 2){
      flightData.seek(kmPos);
      altFlag = 3;
    }
    else if(altFlag == 3){
      if (flightData.available()>= 100){
        flightData.read(sciPacket, 100);
        while (digitalRead(CTS_PIN)==1){
          delayMicroseconds(1);
        }
        Serial5.write(sciPacket, 100);
        //delay(5);
      }
      else {
        altFlag = 2;
      }

      if (!flightData){
        altFlag = -1;
      }
    }

    /*****************FLUSH SD CARD IF REQUIRED*************/
    if ((flightData)&&(altFlag != 3)&&((millis() - timeSD) > 60000)){
        timeSD = millis();
        flightData.flush();
    }

    noInterrupts();
    WDOG_REFRESH = 0xA602;
    WDOG_REFRESH = 0xB480;
    interrupts();
}
```

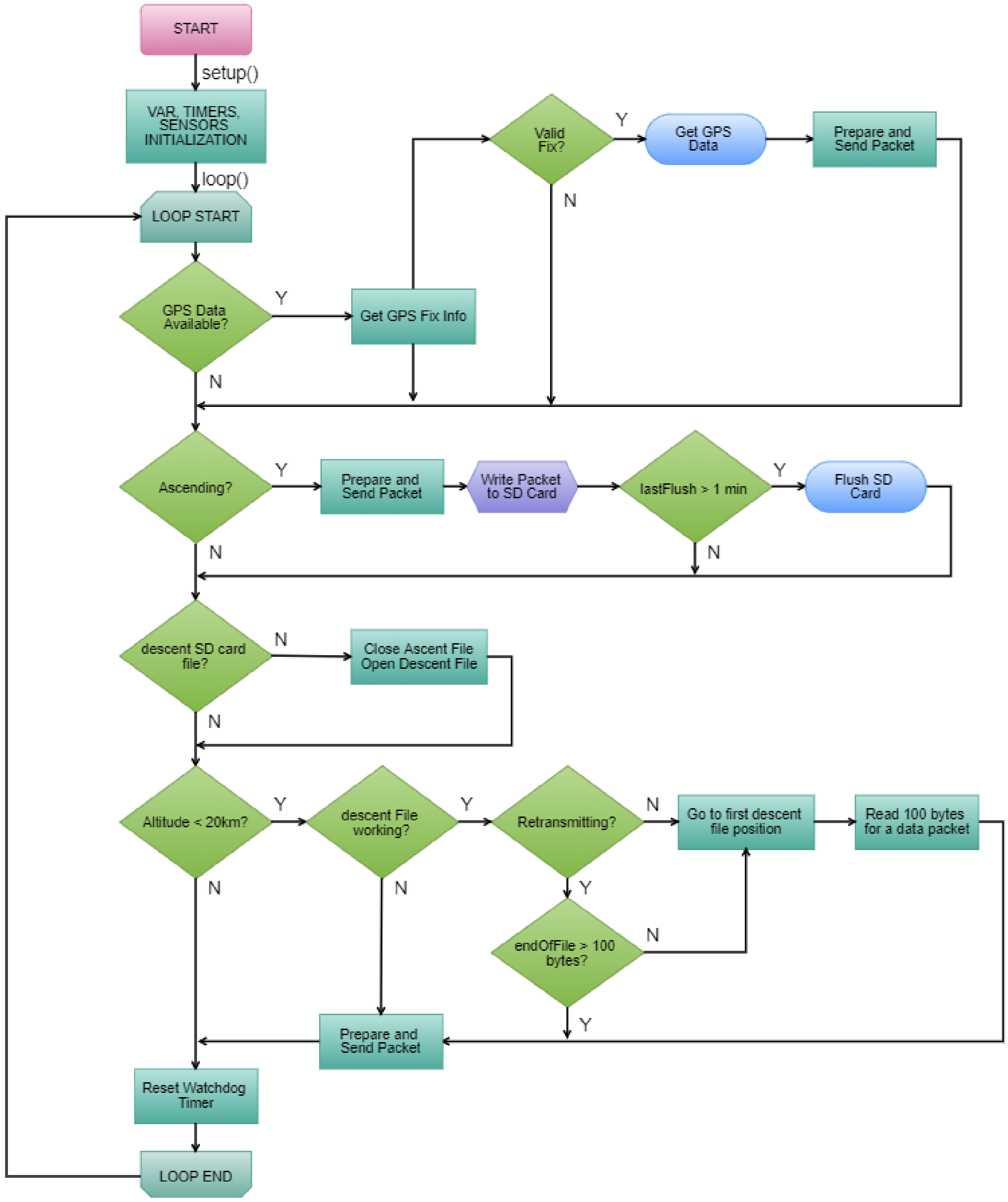


Figure H.2: Design 4 Software Flow Diagram - Payload with Retransmissions.

H.3 Payload - Bluetooth Commands to CDU

```
/*****************************************************************
 * NAME: HAB_Transmitter.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Transmitter.
 *
 * DEVELOPMENT HISTORY:
 *   Date     Author   Version            Description Of Change
 * --------  ------   -------   --------------------------------------
 * 02/23/2018   NMG      1.1     Scientific packets included and sync.
 * 03/02/2018   NMG      1.2     GPS sensors included.
 * 03/20/2018   NMG      1.3     Cutting System Included.
 * 06/04/2018   NMG      1.4     SD Card System Included.
 * 06/08/2018   NMG      1.5     Watchdog Timer Included.
 * 07/20/2018   NMG      2.1     Data Packet and SD File Changes.
 * 08/09/2018   NMG      2.2     Scientific Packets With GPS Info.
 * 10/10/2018   NMG      3       Packets Structure Changes.
 * 01/12/2018   NMG      3.1     Cutting System I2C-SPI Changes.
 * 01/15/2018   NMG      4       Code Adapted to Teensy 3.5.
 * 01/15/2018   NMG      4.1     Code Restructured.
 * 03/09/2018   NMG      4.2     Re-send data implementation.
 * 03/13/2018   NMG      4.3     External Watchdog Timer Included.
 * 05/10/2019   NMG      4.4     GPS packets not considered.
 * 07/02/2019   NMG      4.5     Teensy ADC Reference - Flow Control.
 * 10/11/2019   NMG      5       Bluetooth Communication with CDU.
 *****************************************************************/
/******SERIAL DEFINITIONS*******
* RADIO_TX            Serial5  *
* GPS UBlox           Serial3  *
* Bluetooth           Serial1  *
*******************************/

#include <NMEAGPS.h>
#include <GPSport.h>
#include <Streamers.h>
#include <EEPROM.h>
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <avr/wdt.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>
#include <ADC.h>

/**************DATA BACKUP******************/
File flightData;
String ext = ".bin";
String fileN;
String fileName1;
String fileName2;
int sdFlag;
int32_t timeSD;
unsigned int fileNum;
unsigned int address;
int sdCount;
const int chipSelect = BUILTIN_SDCARD;

/**************GPS SENSORS******************/
static NMEAGPS  uBloxEX;   //uBlox GPS
static gps_fix  uBloxEXFix;
int id;

int32_t lat1;      //Latitude
int32_t lon;      //Longitude
int32_t alt;      //Altitude
int32_t highAlt;
byte stat;        //Status
uint8_t numSats; //Number of Satellites in View
uint8_t utcHour; //UTC Time - Hour
uint8_t utcMin;  //UTC Time - Minutes
uint8_t utcSec;  //UTC Time - Seconds

/*********DATA PACKETS AND SENSORS VARIABLES**********/
ADC *adc = new ADC();

byte id_sci[2] = {0xA0, 0xB1}; //Identifier for only scientific data packet.
byte id_gps[2] = {0xC0, 0xD1}; //Identifier for packet with GPS data.

byte sciPacket[100];            //Scientific data packet byte array.
unsigned int packet_number;     //Packet number/counter.
unsigned long time_packet;      //Packet Timestamp.

//ANALOG PINS DEFINITION
#define TEMP_EXT_H A9
#define TEMP_EXT_L A8
#define TEMP_INT A7
#define VOLTAGE A6

int tempExt_l;                  //External Temperature - Low Range.
int tempExt_h;                  //External Temperature - High Range.
int tempInt;                    //Internal Temperature.
int voltage;                    //Voltage Monitor (VBat).

//9DoF PINS DEFINITION
#define LSM9DS1_XGCS 10
```

```
#define LSM9DS1_MCS 9

//9DoF VARIABLES DEFINITION
Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1(LSM9DS1_XGCS, LSM9DS1_MCS);
sensors_event_t a, m, g, temp;
int accel_x;
int accel_y;
int accel_z;

int gyro_x;
int gyro_y;
int gyro_z;


/**************CONTROLLED DESCENT************/
/********TEST***********/
int altTest;
int testFlag;
int top;
int topFlag;

/********CODES********/
byte OPCLcode = 0xAA;
byte OPENcode = 0xBB;
byte CLOSEcode = 0xCC;
byte CUTcode = 0xDD;
byte CHECKcode = 0xEE;
/*********************/

#define ALT_THRES 2500000        //CM! - Altitude threshold to open the valve.
#define DSCNT_THRES_MIN -3.5     //Descent rate at which the valve will be closed - Min value.
#define DSCNT_THRES_MAX -2       //Descent rate at which the valve will be closed - Max value.
#define DSCNT_MEAN 60            //Seconds considered to compute the average descent rate (/[Nav.Rate]).


byte code, temp1, temp2;
int16_t temperature;

int altFlag;
int altCnt;
unsigned long altDscntCnt;
int kmPos;
int descentFlag;
unsigned long timeFin;
unsigned long timePrev;
unsigned long timeAlt;
int dscntCnt;
double descentRate;
double descentSum;
double descentAverage;
int32_t prevAlt;
int opclFlag;
int valveStatus;


void setupSensor()
{
  // 1.) Set the accelerometer range
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_4G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_8G);
  //lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_16G);

  // 2.) Setup the gyroscope
  lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_245DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_500DPS);
  //lsm.setupGyro(lsm.LSM9DS1_GYROSCALE_2000DPS);
}

void setup() {
  //Serial.begin(230400); //Serial Monitor

  /************IMU SENSOR SETUP**********/
  lsm.begin();
  setupSensor();

  analogReadResolution(10);
  //adc->setAveraging(8);
  adc->setReference(ADC_REFERENCE::REF_3V3, ADC_0);
  //adc->setConversionSpeed(ADC_CONVERSION_SPEED::LOW_SPEED);

  /************RADIO COMMS INI**********/
  Serial5.begin(230400);
  id = 0;

  /************SD CARD**************/
 sdCount = 0;
  while (!SD.begin(chipSelect)) {
    delay(200);
  }
  //Read and set file number and name
  address = 0;
  fileNum = EEPROM.read(address);
  fileNum = fileNum + 1;
  EEPROM.write(address, fileNum);
  fileN =  "data";
  String m1 = fileN + fileNum;
  fileName1 = m1 + ext;

```

```
  //Open file
  flightData = SD.open(fileName1.c_str(), FILE_WRITE);
  timeSD = millis();

  //Prepare the next file for the descent
  fileN =  "dscnt";
  String m2 = fileN + fileNum;
  fileName2 = m2 + ext;


  /************GPS SERIALS**********/
  gpsuBloxExt.begin(9600);
  packet_number = 0;
  highAlt = 0;
  altFlag = 0;
  altTest = 0;
  altCnt = 0;
  altDscntCnt = 0;
  testFlag = 0;
  alt = 0;
  descentFlag = 0;

/************BLUETOOTH SETUP**********/
   Serial1.begin(38400);
   timeFin = millis();

/*********WATCHDOG TIMER*********/
   noInterrupts();
   WDOG_UNLOCK = WDOG_UNLOCK_SEQ1;
   WDOG_UNLOCK = WDOG_UNLOCK_SEQ2;
   delayMicroseconds(1);

   WDOG_TOVALH = 0x006d;
   WDOG_TOVALL = 0xdd00;
   WDOG_PRESC  = 0x400;

   WDOG_STCTRLH |= WDOG_STCTRLH_ALLOWUPDATE |
   WDOG_STCTRLH_WDOGEN | WDOG_STCTRLH_WAITEN |
   WDOG_STCTRLH_STOPEN | WDOG_STCTRLH_CLKSRC;
   interrupts();
}


void send_sci_packet(int id)
{
    if (id==3){

      sciPacket[0] = id_gps[0];
      sciPacket[1] = id_gps[1];

      sciPacket[2] = packet_number;
      sciPacket[3] = packet_number >> 8;

      sciPacket[4] = lat1;
      sciPacket[5] = lat1 >> 8;
      sciPacket[6] = lat1 >> 16;
      sciPacket[7] = lat1 >> 24;

      sciPacket[8] = lon;
      sciPacket[9] = lon >> 8;
      sciPacket[10] = lon >> 16;
      sciPacket[11] = lon >> 24;

      sciPacket[12] = alt;
      sciPacket[13] = alt >> 8;
      sciPacket[14] = alt >> 16;
      sciPacket[15] = alt >> 24;

      sciPacket[16] = stat;

      sciPacket[17] = numSats;

      sciPacket[18] = utcHour;
      sciPacket[19] = utcMin;
      sciPacket[20] = utcSec;

      /*******CDU DATA*******/
      sciPacket[21] = temp1;
      sciPacket[22] = temp2;
      sciPacket[23] = code;
      sciPacket[24] = valveStatus;

    }
    else {
      sciPacket[0] = id_sci[0];
      sciPacket[1] = id_sci[1];

      packet_number++;
      sciPacket[2] = packet_number;
      sciPacket[3] = packet_number >> 8;

      adc->setConversionSpeed(ADC_CONVERSION_SPEED::LOW_SPEED);
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[4] = tempExt_h;
      sciPacket[5] = tempExt_h >> 8;

      tempInt = analogRead(TEMP_INT);
      sciPacket[6] = tempInt;
      sciPacket[7] = tempInt  >> 8;
```

```

      tempExt_l = analogRead(TEMP_EXT_L);
      sciPacket[8] = tempExt_l;
      sciPacket[9] = tempExt_l >> 8;

      adc->setConversionSpeed(ADC_CONVERSION_SPEED::MED_SPEED);
      tempExt_h = analogRead(TEMP_EXT_H);
      sciPacket[10] = tempExt_h;
      sciPacket[11] = tempExt_h >> 8;

      voltage = analogRead(VOLTAGE);
      sciPacket[12] = voltage;
      sciPacket[13] = voltage >> 8;


      lsm.getEvent(&a, &m, &g, &temp);

      accel_x = a.acceleration.x;
      sciPacket[14] = accel_x;
      sciPacket[15] = accel_x >> 8;

      accel_y = a.acceleration.y;
      sciPacket[16] = accel_y;
      sciPacket[17] = accel_y >> 8;

      accel_z = a.acceleration.z;
      sciPacket[18] = accel_z;
      sciPacket[19] = accel_z >> 8;

      gyro_x = g.gyro.x;
      sciPacket[20] = gyro_x;
      sciPacket[21] = gyro_x >> 8;

      gyro_y = g.gyro.y;
      sciPacket[22] = gyro_y;
      sciPacket[23] = gyro_y >> 8;

      gyro_z = g.gyro.z;
      sciPacket[24] = gyro_z;
      sciPacket[25] = gyro_z >> 8;
    }

    for(int i=0; i<3; i++)
    {
        tempExt_h = analogRead(TEMP_EXT_H);
        sciPacket[30+22*i] = tempExt_h;
        sciPacket[30+22*i+1] = tempExt_h >> 8;

        tempExt_l = analogRead(TEMP_EXT_L);
        sciPacket[30+22*i+2] = tempExt_l;
        sciPacket[30+22*i+3] = tempExt_l  >> 8;

        voltage = analogRead(VOLTAGE);
        sciPacket[30+22*i+4] = voltage;
        sciPacket[30+22*i+5] = voltage >> 8;

        accel_x = a.acceleration.x;
        sciPacket[30+22*i+6] = accel_x;
        sciPacket[30+22*i+7] = accel_x >> 8;

        accel_y = a.acceleration.y;
        sciPacket[30+22*i+8] = accel_y;
        sciPacket[30+22*i+9] = accel_y >> 8;

        accel_z = a.acceleration.z;
        sciPacket[30+22*i+10] = accel_z;
        sciPacket[30+22*i+11] = accel_z >> 8;

        gyro_x = g.gyro.x;
        sciPacket[30+22*i+12] = gyro_x;
        sciPacket[30+22*i+13] = gyro_x >> 8;

        gyro_y = g.gyro.y;
        sciPacket[30+22*i+14] = gyro_y;
        sciPacket[30+22*i+15] = gyro_y >> 8;

        gyro_z = g.gyro.z;
        sciPacket[30+22*i+16] = gyro_z;
        sciPacket[30+22*i+17] = gyro_z >> 8;

        /**EXTRA EXTERNAL TEMP READINGS**/
        tempExt_h = analogRead(TEMP_EXT_H);
        sciPacket[30+22*i+18] = tempExt_h;
        sciPacket[30+22*i+19] = tempExt_h >> 8;

        tempExt_l = analogRead(TEMP_EXT_L);
        sciPacket[30+22*i+20] = tempExt_l;
        sciPacket[30+22*i+21] = tempExt_l >> 8;
    }
        /**EXTRA TEMP READINGS**/
        tempExt_h = analogRead(TEMP_EXT_H);
        sciPacket[96] = tempExt_h;
        sciPacket[97] = tempExt_h >> 8;

        tempInt = analogRead(TEMP_INT);
        sciPacket[98] = tempInt;
        sciPacket[99] = tempInt >> 8;

         /****PACKET TIMESTAMP****/
```

```
        time_packet = millis();
        sciPacket[26] = time_packet;
        sciPacket[27] = time_packet >> 8;
        sciPacket[28] = time_packet >> 16;
        sciPacket[29] = time_packet >> 24;

        /*******CDU DATA*******/
        sciPacket[30] = temp1;
        sciPacket[31] = temp2;
        sciPacket[32] = code;
        sciPacket[33] = valveStatus;

       Serial5.write(sciPacket, 100);
       delay(3);
}



static void descentSystem()
{
  if (descentFlag == 1)
  {
      if (dscntCnt < (DSCNT_MEAN))
      {
        descentSum += descentRate;
        dscntCnt +=1;
      }
      else
      {
        descentAverage = descentSum/(DSCNT_MEAN);
        //Serial.print("Average Descent Rate: "); Serial.println(descentAverage);
        dscntCnt = 0;
        descentSum = 0;
        if((descentAverage < DSCNT_THRES_MAX)&&(descentAverage > DSCNT_THRES_MIN))
        {
           descentFlag = 2;
        }
      }
   }
}


void CDUSystem()
{
  if ((alt<ALT_THRES)&&(descentFlag != 2))
  {
      Serial1.write(CHECKcode);
      //Serial.print("Sending Check Code.");
  }

  else if((alt>ALT_THRES)&&(opclFlag<15)){
      Serial1.write(OPCLcode);
      //Serial.print("Sending Open/Close Code.");
      opclFlag +=1;
  }

  else if((descentFlag != 2)&&(opclFlag==15)){
      Serial1.write(OPENcode);
      //Serial.print("Sending Open Code.");
  }

  else if(descentFlag == 2){
      Serial1.write(CLOSEcode);
      //Serial.print("Sending Close Code.");
  }
  noInterrupts();
  WDOG_REFRESH = 0xA602;
  WDOG_REFRESH = 0xB480;
  interrupts();
}


void loop()
{
  while (uBloxEX.available( gpsuBloxExt )) {
    uBloxEXFix = uBloxEX.read();
        if (uBloxEXFix.valid.location) {
          lat1 = uBloxEXFix.latitudeL();         // Scaled by 10,000,000
          lon = uBloxEXFix.longitudeL();        // Scaled by 10,000,000
          alt = uBloxEXFix.altitude_cm();
          stat = uBloxEXFix.status;
          numSats = uBloxEX.sat_count;
          utcHour = uBloxEXFix.dateTime.hours;
          utcMin = uBloxEXFix.dateTime.minutes;
          utcSec = uBloxEXFix.dateTime.seconds;

/**********************TEST***********************
//    if (altFlag < 4)
//    {
//      if((alt<3000000)&&(topFlag == 0)){
//        alt = alt + 50000;
//      }
//      else{
//        topFlag = 1;
//      }
//
//      if(topFlag == 1){
//        alt = alt - 300;
//      }
```

```
*****************************************************/

        timeAlt = millis();

        float timeLast = float((timeAlt-timePrev))/1000;
        float altLast = float((alt-prevAlt))/100;

        //Serial.print("Time Diff: "); Serial.println(timeLast);
        //Serial.print("Altitude Diff: "); Serial.println(altLast);
        descentRate = altLast/timeLast;
        //Serial.print("Altitude: "); Serial.print(alt); Serial.print(", Ascent/Descent Rate: "); Serial.
             println(descentRate);


        prevAlt = alt;
        timePrev = timeAlt;

        descentSystem();
        send_sci_packet(3);
      }
    }

    if (altFlag < 2)
    {
        send_sci_packet(0);
        flightData.write(sciPacket, 100);

        if (alt > highAlt){
          highAlt = alt;
          altDscntCnt = millis();
        }
        else if ((altFlag == 0)&&((highAlt-10000-alt) > 0)){
          if ((millis() - altDscntCnt) > 120000) {
            flightData.close();
            flightData = SD.open(fileName2.c_str(), FILE_WRITE);
            timeSD = millis();
            kmPos = flightData.position();
            altFlag = 1;
            descentFlag = 1;
          }
        }
        else if ((altFlag == 1)&&(alt < 2000000)){ //ALTITUDE IN CM!
              altFlag = 2;
          }
    }

    else if (altFlag == 2){
      flightData.seek(kmPos);
      altFlag = 3;
    }
    else if(altFlag == 3){
      if (flightData.available()>= 100){
        flightData.read(sciPacket, 100);
        Serial5.write(sciPacket, 100);
        delay(5);
      }
      else {
        altFlag = 2;
      }

      if (!flightData){
        altFlag = -1;
      }
    }

    /********FLUSH SD CARD IF REQUIRED********/
    if ((flightData)&&(altFlag != 3)&&((millis() - timeSD) > 60000)){
        timeSD = millis();
        flightData.flush();
    }

  /********CDU CHECKS-ACTIONS********/
  if ((millis()-timeFin)>3000)
  {
    CDUSystem();
    while(Serial1.available() > 0)
    {
      byte c2 = Serial1.read();
      if((c2==CHECKcode)||(c2==OPCLcode)||(c2==OPENcode)||(c2==CLOSEcode))
      {
        code = c2;
        temp1 = Serial1.read();
        temp2 = Serial1.read();
        //Serial.print(temp1); Serial.print(" "); Serial.println(temp2);
        temperature = (int16_t) (temp1 + (temp2<<8));
        //Serial.print("Received Code: "); Serial.print(c2,HEX);
        //Serial.print(", Temperature: "); Serial.println(temperature);
      }
    }
    timeFin = millis();
  }
    noInterrupts();
    WDOG_REFRESH = 0xA602;
    WDOG_REFRESH = 0xB480;
    interrupts();
}
```

H.4 External CDU - Cutting Thread

```
/*****************************************************************
 * NAME: controlledDescentUnit_v1.4.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Controlled Descent Unit.
 *
 * DEVELOPMENT HISTORY:
 *   Date      Author  Version              Description Of Change
 * --------  ------  -------   -----------------------------------------------
 * 11/20/2018  NMG      1.1      Controlled descent implementation based on time
 *                               and altitude.
 * 03/27/2019  NMG      1.2      Time threshold - Resets Consideration.
 * 04/07/2019  NMG      1.3      Heating Pad - Temperature Sensor Addition.
 * 06/27/2019  NMG      1.4      Heating Pad - Temperature Range
 *                               Calibrated/Adjusted.
 *************************************************************/

#include <NMEAGPS.h>
#include <GPSport.h>
#include <math.h>
#include <EEPROM.h>
#include <Wire.h>
#include "SparkFunTMP102.h"

//const int ALERT_PIN = A3;

TMP102 sensor0(0x48);
float temperature;
boolean alertPinState, alertRegisterState;

/*******************GPS SENSOR******************/
//Altitude
int32_t alt;
int32_t altIni;
int32_t altFini;

//Library Variables Declaration
static NMEAGPS  uBlox;       //uBlox Sensor
static gps_fix  uBloxFix;    //Fix/Sentence to be parsed



/***************CUTTING SYSTEM******************/
#define ALT_THRES 33000
#define CUTTING_TIME 12000 //Time that the system will be activated [ms].
#define TIME_THRES 8000000 //Initial time threshold [ms].
#define CUT_ENABLE 13
#define PAD_ENABLE 4
volatile int cuttingOn;
volatile int cuttingDone;
unsigned long timeCutStart;
unsigned long timeCutting;
unsigned long flightTime;
unsigned long timeThreshold;
unsigned long timerEEPROM;
unsigned long timeEEPROM;
int ascentRate;
int address;
int n;

void setup()
{
  Serial.begin(9600);
  /***************CUTTING SYSTEM******************/
   pinMode(CUT_ENABLE, OUTPUT);
  digitalWrite(CUT_ENABLE, LOW);

  pinMode(PAD_ENABLE, OUTPUT);
  digitalWrite(PAD_ENABLE, LOW);

  pinMode(ALERT_PIN,INPUT);
  sensor0.begin();

  // set the number of consecutive faults before triggering alarm.
  // 0-3: 0:1 fault, 1:2 faults, 2:4 faults, 3:6 faults.
  sensor0.setFault(2);  // Trigger alarm immediately

  // set the polarity of the Alarm. (0:Active LOW, 1:Active HIGH).
  sensor0.setAlertPolarity(0); // Active Low

  // set the sensor in Comparator Mode (0) or Interrupt Mode (1).
  sensor0.setAlertMode(0); // Comparator Mode.

  // set the Conversion Rate (how quickly the sensor gets a new reading)
  //0-3: 0:0.25Hz, 1:1Hz, 2:4Hz, 3:8Hz
  sensor0.setConversionRate(1);

  //set Extended Mode.
  //0:12-bit Temperature(-55C to +128C) 1:13-bit Temperature(-55C to +150C)
  sensor0.setExtendedMode(0);

  //set T_HIGH, the upper limit to trigger the alert on
  sensor0.setHighTempC(0); // set T_HIGH in C

  //set T_LOW, the lower limit to shut turn off the alert
```

```
  sensor0.setLowTempC(-10); // set T_LOW in C

  timeCutStart = 0;
  timeCutting = 0;
  cuttingOn = 0;
  cuttingDone = 0;
  n = 0;
  timeEEPROM = 0;
  timeThreshold = TIME_THRES;

  address = 4;
  timeEEPROM = EEPROM.read(address); //[mins]
  //Serial.println(timeEEPROM);
  timeThreshold = timeThreshold -  (60000*timeEEPROM);//[ms]
  //Serial.println(timeThreshold);

  delay(2000);
  timerEEPROM = millis();
  //Serial.println(timerEEPROM);
}


static void cuttingSystem()
{
  flightTime = millis();
  if (cuttingOn == 0)
  {
    if ((flightTime > timeThreshold)||(alt > ALT_THRES))
    {
      //Serial.println("CUTTING!!");
      digitalWrite(ENABLE, HIGH);
      delay(12000);
      cuttingOn = 1;
    }
  }
  else if (cuttingOn == 1)
  {
    while(true){
      digitalWrite(ENABLE, LOW);
      delay(30000);
      digitalWrite(ENABLE, HIGH);
      delay(12000);
    }
  }
}


void loop()
{
  while (uBlox.available( Serial ))
  {
    uBloxFix = uBlox.read();
    if (uBloxFix.valid.location)
    {
      alt = uBloxFix.alt.whole;
      //Serial.println(alt);
    }
  }
  flightTime = millis();
  if ((flightTime-timerEEPROM) > 60000) {
    timerEEPROM = millis();
    timeEEPROM = timeEEPROM + 1;
    EEPROM.write(address, (timeEEPROM));
  }
  sensor0.wakeup();
  temperature = sensor0.readTempC();
  //Serial.println(temperature);

  // Check for Alert
  alertRegisterState = sensor0.alert();   // read the Alert from register

  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.print("\tAlert Pin: ");
  Serial.println(alertPinState);

  digitalWrite(PAD_ENABLE, alertRegisterState);
  cuttingSystem();
  //delay(2000);
}
```

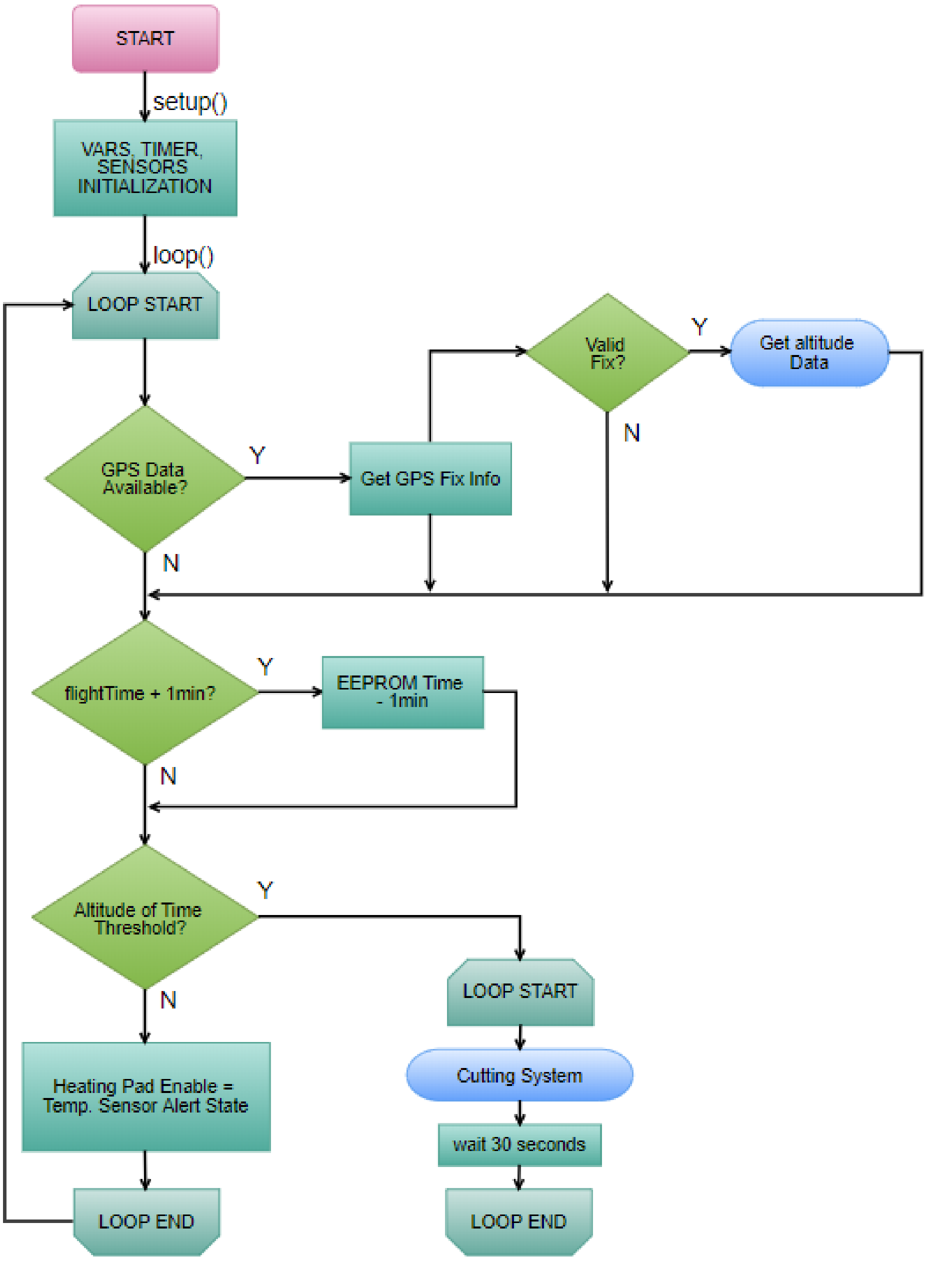


Figure H.3: Design 3 Software Flow Diagram - External Controlled Descent Unit Block Diagram (Cap and Cutting Thread Systems).

H.5 External CDU - Valve System

```
/**********************************************************
 * NAME: controlledDescentUnit.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Controlled Descent Unit.
 *
 * AUTHORS: Noemi Miguelez Gomez, Julio Cesar Guardado
 *
 * DEVELOPMENT HISTORY:
 *   Date      Author  Version              Description Of Change
 * --------   ------  -------   ----------------------------------------------
 * 11/20/2018  NMG      1.1      Controlled descent implementation based on time
 *                               and altitude.
 * 03/27/2019  NMG      1.2      Time threshold - Resets Consideration.
 * 04/07/2019  NMG      1.3      Heating Pad - Temperature Sensor Addition.
 * 06/27/2019  NMG      1.4      Heating Pad - Temperature Range Adjusted.
 * 07/29/2019  JCG      2.0      Valve System - Servo Motor Calibration
 * 07/31/2019  NMG      2.1      Valve System - Altitude/Ascent Rate Logic
 *********************************************************/

#include <NMEAGPS.h>
#include <GPSport.h>
#include <math.h>
#include <EEPROM.h>
#include <Wire.h>
#include <Servo.h>
#include "SparkFunTMP102.h"

TMP102 sensor0(0x48);
Servo myservo;
float temperature;
boolean alertPinState, alertRegisterState;

/********************GPS SENSOR******************/
//Altitude
int32_t alt;
int32_t altIni;
int32_t altFini;

//Library Variables Declaration
static NMEAGPS  uBlox;       //uBlox Sensor
static gps_fix  uBloxFix;    //Fix/Sentence to be parsed

/***************VALVE SYSTEM******************/
#define ALT_THRES 2500000        //CM! - Altitude threshold to open the valve.
#define POS_OPEN 180             //angle where valve is open [deg]
#define POS_CLOSED 0             //angle where valve is closed [deg]
#define PAD_ENABLE 4
#define DSCNT_THRES_MIN -3     //CM! - Descent rate at which the valve will be closed - Min value.
#define DSCNT_THRES_MAX -1.5     //CM! - Descent rate at which the valve will be closed - Max value.
#define MAX_ALT_THRES 30         //Seconds after reaching altitude threshold to consider it valid [for GPS
     errors].
#define DSCNT_MEAN 30            //Seconds considered to compute the average descent rate.

volatile int valveFlag;
unsigned long posOpen;
unsigned long posClosed;
unsigned long altDscntCnt;
unsigned long altMax;
int altCnt;
unsigned long timePrev;
unsigned long timeAlt;
int dscntCnt;
double descentRate;
double descentSum;
double descentAverage;
int prevAlt;
int address;
int n;
int top;

void setup()
{
  Serial.begin(9600);
  /*********VALVE SYSTEM*********/
  myservo.attach(9);       //Set servo PWM pin to pin 9

  pinMode(PAD_ENABLE, OUTPUT);
  digitalWrite(PAD_ENABLE, LOW);

  //pinMode(ALERT_PIN,INPUT);
  sensor0.begin();

  // Set the number of consecutive faults before activate pin.
  // 0-3: 0:1 fault, 1:2 faults, 2:4 faults, 3:6 faults.
  sensor0.setFault(2);

  // Set the polarity of the Alarm. (0:Active LOW, 1:Active HIGH).
  sensor0.setAlertPolarity(0); // Active Low

  // Set the sensor in Comparator Mode (0) or Interrupt Mode (1).
  sensor0.setAlertMode(0); // Comparator Mode.

  // Set the Conversion Rate (how quickly the sensor gets a new reading)
  //0-3: 0:0.25Hz, 1:1Hz, 2:4Hz, 3:8Hz
```

```
  sensor0.setConversionRate(1);

  //Set Mode.
  //0:12-bit Temperature(-55C to +128C) 1:13-bit Temperature(-55C to +150C)
  sensor0.setExtendedMode(0);

  //Set the upper limit to turn off the alert
  sensor0.setHighTempC(0); // set T_HIGH in C

  //Set the lower limit to turn on the alert
  sensor0.setLowTempC(-10); // set T_LOW in C

  valveFlag = -1;
  posOpen = POS_OPEN;
  posClosed = POS_CLOSED;
  n = 0;
  myservo.write(posClosed);
  delay(2000);
}

static void valveSystem()
{
  if (valveFlag < 2)
  {
    if (valveFlag == 0)
    {
        //Serial.println("Opening valve...");
        myservo.write(posOpen);
        valveFlag = 1;
        prevAlt = alt;
        timePrev = millis();
    }
    else if (valveFlag == 1)
    {
      float timeLast = float((timeAlt-timePrev))/1000;
      descentRate = ((alt-prevAlt)/(timeLast))/100;
      prevAlt = alt;
      timePrev = timeAlt;

      if (dscntCnt < (DSCNT_MEAN*2)) //2Hz
      {
        descentSum += descentRate;
        dscntCnt +=1;
      }
      else
      {
        descentAverage = descentSum/(DSCNT_MEAN*2);
        //Serial.print("Average Descent Rate: "); Serial.println(descentAverage);
        dscntCnt = 0;
        descentSum = 0;
        if((descentAverage < DSCNT_THRES_MAX)&&(descentAverage > DSCNT_THRES_MIN))
        {
          //Serial.println("Closing valve...");
          myservo.write(posClosed);
          valveFlag = 2;
        }
      }
     }
   }
}

void loop()
{
  while (uBlox.available( Serial ))
  {
    uBloxFix = uBlox.read();
    if (uBloxFix.valid.location)
    {
      alt = uBloxFix.altitude_cm();
      //Serial.println(alt);
      timeAlt = millis();
    }
  }

      if ((alt>ALT_THRES)&&(valveFlag==-1))
      {
          altCnt +=1;
          if ((altCnt>MAX_ALT_THRES*2))
          {
            valveFlag = 0;
          }
      }
      else
      {
          altCnt = 0;
      }

  sensor0.wakeup();
  temperature = sensor0.readTempC();


  alertRegisterState = sensor0.alert();   // read the Alert from register

  digitalWrite(PAD_ENABLE, alertRegisterState);
  valveSystem();

}
```

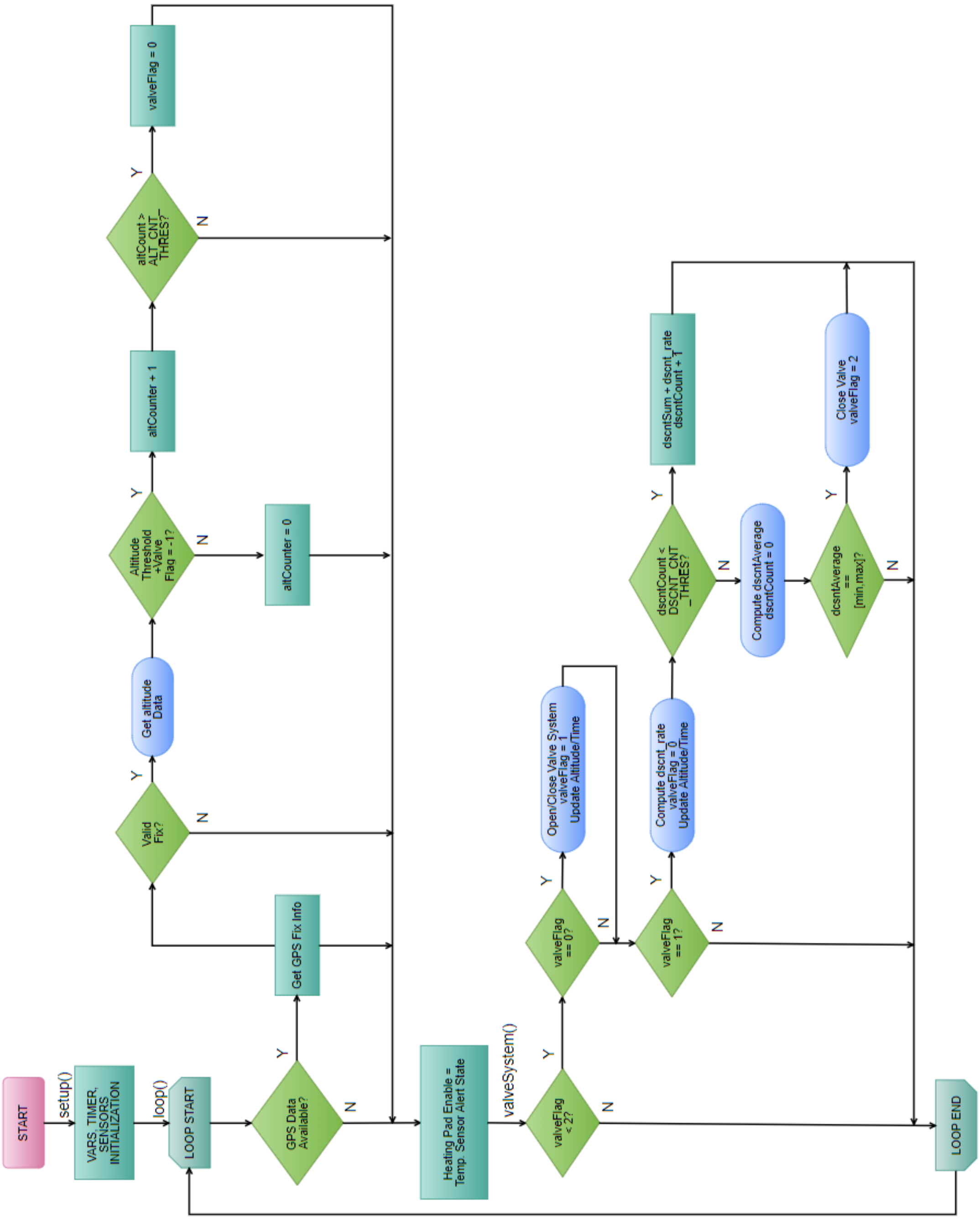


Figure H.4: Design 4 Software Flow Diagram - CDU Only Valve System

H.6 External CDU - Valve System - Bluetooth

```
/****************************************************************
 * NAME: controlledDescentUnit.ino
 *
 * PURPOSE: AFOSR-MURI HIGH ALTITUDE BALLOON - Controlled Descent Unit.
 *
 * AUTHORS: Noemi Miguelez Gomez, Julio Cesar Guardado
 *
 * DEVELOPMENT HISTORY:
 *   Date      Author  Version              Description Of Change
 * --------   ------  -------   --------------------------------
 * 11/20/2018  NMG      1.1      Controlled descent implementation based on time and altitude.
 * 03/27/2019  NMG      1.2      Time threshold - Resets Consideration.
 * 04/07/2019  NMG      1.3      Heating Pad - Temperature Sensor Addition.
 * 06/27/2019  NMG      1.4      Heating Pad - Temperature Range Adjusted.
 * 07/29/2019  JCG      2.0      Valve System - Servo Motor Calibration
 * 07/31/2019  NMG      2.1      Valve System - Altitude/Ascent Rate Logic
 * 08/26/2019  NMG      2.2      Valve Open/Close System - Grease Problems
 * 09/18/2019  NMG      2.3      Valve System - Timer for GNSS Errors
 * 09/20/2019  NMG      3.0      Valve + Cutting System
 * 10/08/2019  NMG      4.0      Valve + Bluetooth System
 ******************************************************************/

#include <EEPROM.h>
#include <Wire.h>
#include <Servo.h>
#include <SoftwareSerial.h>
#include <SparkFunTMP102.h>


TMP102 sensor0(0x48);
Servo myservo;
float temperature;
int16_t intTemp;
boolean alertPinState, alertRegisterState;


/***************VALVE SYSTEM*******************/
#define POS_OPEN 0              //angle where valve is open [deg]
#define POS_CLOSED 180              //angle where valve is closed [deg]
#define PAD_ENABLE 4

unsigned long posOpen;
unsigned long posClosed;
unsigned long timeData;


SoftwareSerial BTserial(10, 11); // RX | TX
byte OPCLcode = 0xAA;
byte OPENcode = 0xBB;
byte CLOSEcode = 0xCC;
byte CUTcode = 0xDD;
byte CHECKcode = 0xEE;
byte c2;
byte msg[4];
int code;

byte output[4];
int valveStatus;

void setup()
{
  Serial.begin(9600);
  Serial.println("Arduino with HC-05 is ready");

  // start communication with the HC-05 using 38400
  BTserial.begin(38400);
  Serial.println("BTserial started at 38400");


  /*********VALVE SYSTEM*********/
  myservo.attach(9);       //Set servo PWM pin to pin 9

  pinMode(PAD_ENABLE, OUTPUT);
  digitalWrite(PAD_ENABLE, LOW);

  //pinMode(ALERT_PIN,INPUT);
  sensor0.begin();

  // Set the number of consecutive faults before activate pin.
  // 0-3: 0:1 fault, 1:2 faults, 2:4 faults, 3:6 faults.
  sensor0.setFault(2);

  // Set the polarity of the Alarm. (0:Active LOW, 1:Active HIGH).
  sensor0.setAlertPolarity(0); // Active Low

  // Set the sensor in Comparator Mode (0) or Interrupt Mode (1).
  sensor0.setAlertMode(0); // Comparator Mode.

  // Set the Conversion Rate (how quickly the sensor gets a new reading)
  //0-3: 0:0.25Hz, 1:1Hz, 2:4Hz, 3:8Hz
  sensor0.setConversionRate(1);

  //Set Mode.
  //0:12-bit Temperature(-55C to +128C) 1:13-bit Temperature(-55C to +150C)
```

```
  sensor0.setExtendedMode(0);

  //Set the upper limit to turn off the alert
  sensor0.setHighTempC(10); // set T_HIGH in C

  //Set the lower limit to turn on the alert
  sensor0.setLowTempC(0); // set T_LOW in C

  posOpen = POS_OPEN;
  posClosed = POS_CLOSED;

  myservo.write(posClosed);
  delay(2000);

  timeData = millis();
  valveStatus=0;
}


void loop()
{
 if (BTserial.available())
  {
    while(BTserial.available()>0){
      byte rec = BTserial.read();
      if((rec==CHECKcode)||(rec==OPCLcode)||(rec==OPENcode)||(rec==CLOSEcode))
      {
          c2=rec;
          Serial.print("Received Code: "); Serial.print(c2,HEX);
          Serial.print(", Temperature: "); Serial.println(temperature);
         if(c2==CHECKcode)
         {
            msg[0] = CHECKcode;
            //BTserial.write(CHECKcode);
            BTserial.write(msg,3);
         }
         else if(c2==OPCLcode)
         {
            msg[0] = OPCLcode;
            //BTserial.write(OPCLcode);
            BTserial.write(msg,3);
            for (int i=0; i<3; i++)
            {
                Serial.println("Opening valve...");
                myservo.write(posOpen);
                delay(5000);
                Serial.println("Closing valve...");
                myservo.write(posClosed);
                delay(2000);
            }
            valveStatus=1;
         }
         else if(c2==OPENcode)
         {
            Serial.println("Opening valve...");
            myservo.write(posOpen);
            msg[0] = OPENcode;
            //BTserial.write(OPENcode);
            BTserial.write(msg,3);
            delay(5000);
            valveStatus=2;
         }
         else if(c2==CLOSEcode)
         {
           Serial.println("Closing valve...");
           myservo.write(posClosed);
           msg[0] = CLOSEcode;
           //BTserial.write(CLOSEcode);
           BTserial.write(msg,3);
           delay(2000);
           valveStatus=3;
         }
       }
     }
  }
  sensor0.wakeup();
  temperature = sensor0.readTempC();
  intTemp = (int16_t)(temperature*100);
  msg[1] = intTemp;
  msg[2] = intTemp>>8;
  msg[3] = valveStatus;
  delay(100);

  alertRegisterState = sensor0.alert();   // read the Alert from register
  digitalWrite(PAD_ENABLE, alertRegisterState);
}
```