AAE 490: TracSat Design Team

Final Report

Authors:

Braden Anderson, Sandeep Baskar, Josh Fitch,

Wellington Froelich, Alec Leven, Robert Miller,

Yash Pahade, Pat Pesa, Bethany Price,

Ben Ranft, Dante Sanaei

May 9, 2020

TracSat Member Roles

Prof. Alex Shashurin, *Instructor*
Glynn Smith, *Teaching Assistant*

Braden Anderson
Laser Tracking System

Sandeep Baskar
Laser Link System

Josh Fitch
Laser Link System

Wellington Froelich
Laser Tracking System

Alec Leven
Laser Link System

Robert Miller
Hardware and Propulsion

Yash Pahade
Laser Tracking System

Pat Pesa
Hardware and Propulsion

Bethany Price
Hardware and Propulsion

Ben Ranft
Electronics

Dante Sanaei
Laser Link System

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 PROJECT OVERVIEW

1.1 PROJECT

TracSat is a General Atomics (GA) supported Design-Build-Test (DBT) project for undergraduate students to gain hands-on experience with CubeSats. The TracSat project focuses on the areas of Laser Communications, Guidance, Navigation and Control (GNC), and Propulsion for CubeSats using a 2D-low friction space analogue.

1.2 MISSION

The proposed mission for the 2019-2020 academic year is to conduct a basic translational maneuver and simultaneously stream video through laser communications. The mission will utilize one CubeSat levitating on the 2-D low friction surface and one stationary Ground Station (GS) located off the low-friction surface. All communications between the GS and the CubeSat will be supported via laser communication technology. The mission sequence will be as follows:

(1) GS and CubeSat establish the laser communication link.
(2) GS sends command to the CubeSat to fire cold gas thrusters on the CubeSats for 30 seconds.
(3) CubeSat transmits real-time video from on-board cameras to the GS while moving.

1.3 OBJECTIVES

To successfully complete the mission described in the previous section, the TracSat team determined several technical objectives. These objectives were initially set in Fall 2019, and were then shifted to realign the focus of the project in Spring 2020. These objectives address each of the necessary technologies, including a modified and improved levitation base, a CubeSat capable of translational motion, a high-data-rate laser communications board, a tracking and control mechanism for maintaining laser communications pointing, and a fully functional ground station.

Table 1.1. Mission Objectives

MO 1	Develop a "levitation base" for CubeSat testing and demonstration.
MO 2	Develop a CubeSat capable of translational movement.
MO 3	Demonstrate transmission of video between a CubeSat and Ground Station via a 50 kHz tracking laser communication.
MO 4	Maintain communication between CubeSat and Ground Station during movement

1.4 SYSTEM REQUIREMENTS

To ensure progress was made over the year in accordance with the objectives, requirements were established at the beginning of the academic year and adjusted at the half-way point. The following are the updated requirements as of January 2020. These system requirements informed all design decisions, component selection, and testing metrics throughout the year for each of the subsystems.

Table 1.2. System Requirements

SR 1	LTS must track and maintain link with CubeSat at velocity up to 10 cm/s
SR 2	LTS must be able to re-establish communication link if lost.
SR 3	LLS must be able to transmit data at rate of at least 50 kHz
SR 4	PLS air bearings must be able to maintain levitation with less than 5° of rotational drift and 5 cm of translational drift, after 30 s of operation
SR 5	PLS thrusters must be able to produce purely translational movement with less than 10° of rotational drift and 10 cm of translational drift, after 8 ft of travel

1.5 SUCCESS CRITERIA

Table 1.3 details the minimum and full success criteria for the project. A system that meets the minimum success criteria can be considered to meet the bare minimum objectives of the project and is considered a success. Exceeding these minimum success criteria is considered additionally beneficial and above and beyond. To meet or exceed the full success criteria is an ideal scenario to be celebrated as a complete success. To quantitatively define these benchmarks numbers were established to mark these thresholds.

Table 1.3. Minimum and Full Success Criteria

Criteria	Minimum Success	Full Success
LTS Connection (FSC 1)	System must maintain laser link in all directions	N/A
LLS Connection (FSC 2)	System must be able to re-establish link if lost.	System must be able to re-establish link if lost within 5 seconds
Data Transfer (FSC 3)	System must be able to transmit data at rate of at least 50 kHz	System is capable of supporting 240p video at 10 fps
Stationary Operation (FSC 4)	5° of rotational drift and 5cm of translational drift, after 30 s of operation	5° of rotational drift and 5 cm of translational drift, after 45 s of operation
Translational Operation (FSC 5)	10° of rotational drift and 10 cm of translational drift, after 8 ft	5° of rotational drift and 5 cm of translational drift, after 8 ft

FSC 1 did not change from the minimum criteria due to the qualitative nature of this criteria. FSC 2 sets a specific recovery time for the laser tracking system, because theoretically the process of finding the signal could take anywhere from 5 seconds to more than an hour. FSC 3 challenges the team to deploy use of the PYNQ board and to develop better code so as to achieve better footage. Meeting FSC 4 and 5 indicates that the propulsion system has been thoroughly tested, reworked, and improved in its stability.

1.6 TIMELINE

Due to COVID-19, this schedule shown below in Fig. 1.1 was unable to be maintained after March 16th. Before this disruption, the propulsion and laser tracking systems were on schedule, and the laser link system was slightly behind schedule.

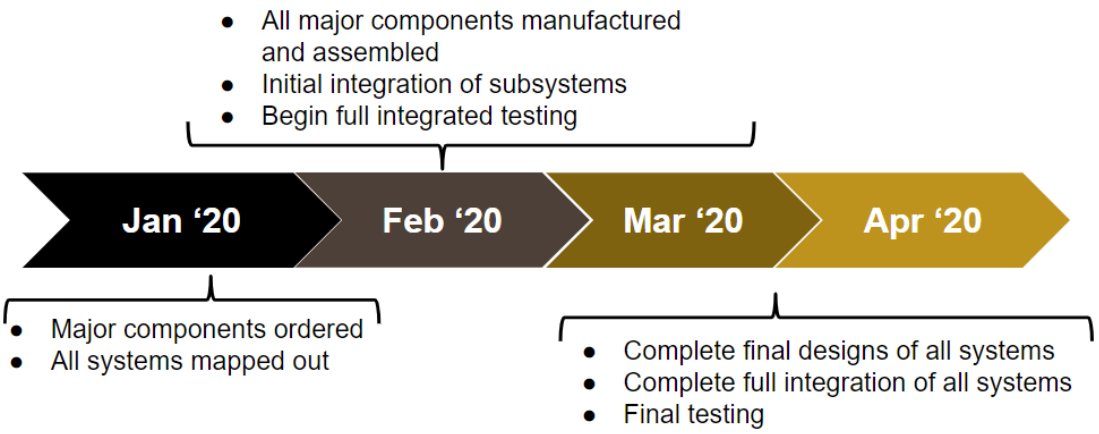

Figure 1.1. Timeline of Spring Semester

Post COVID-19, the final integration and test of the systems could not be done. This is because the CubeSat could only be operated on the low-friction table at Purdue. Since the Purdue campus was closed, the team could not perform the final test. In lieu of performing the final test, it was decided that the individual subsystem teams would complete testing of their subsystems by the end of the semester. Final system integration would be left for future semesters.

2 REVIEW OF SYSTEM DESCRIPTION

2.1 BRIEF SUMMARY

The TracSat system consists of a single CubeSat mounted on a low-friction levitation base. The CubeSat translates using pneumatic thrusters. The CubeSat communicates with a fixed ground station via laser diodes and detectors utilizing PYNQ controller boards. The laser communication system is actively controlled and rotates to maintain direct contact between the Ground Station and CubeSat at all times.

2.2 PUBLIC HEALTH AND ECONOMIC FACTORS

Due to the COVID-19 pandemic, all in-person classes were cancelled at Purdue University. This included the TracSat DBT. Meetings were moved online and tasks were suspended or put on halt. The major impact of the virus on our team was preventing a full system integration and test. Individual systems returned home with team members to work on, but the assembly of the project was discontinued for the semester. One other issue the team faced was ordering parts. A laser diode that was under consideration was eliminated because importing goods from China was at a standstill.

In light of these challenges, the team still met virtually every week to discuss progress and even present a Test Readiness Review. On a positive note, the team learned how to work in conditions that true engineering firms faced around the world.

2.3 ENGINEERING STANDARDS

The team followed general engineering practices for setting requirements, brainstorming, designing, building, and testing the Tracsat system. The Fall 2019 semester was used as an opportunity to brainstorm and test technologies for the final product, which was to be made in the Spring 2020 semester. An iterative approach was taken when problems arose that were open ended. In addition, test systems were developed in the Fall 2019 semester to allow for more rapid prototyping of new systems. Examples of this include the wheeled Testbeds, which were used in lieu of CubeSats to develop the laser tracking systems.

Two components of the project required additional safety measures. The lasers involved were all below 5 milliwatts in power. This meant that, according to Purdue guidelines, special precautions would not have to be performed around them. However, care was taken to make sure they weren't shown into people's eyes. The high pressure tank system also required special safety precautions. This affected the design of the system and its operation.

2.4 TEAM ORGANIZATION

2.4.1 Team Structure

The structure of the mission for the 2019-2020 TracSat team led to the decision to create three fundamental sub-teams. These sub-teams are Laser Tracking, Laser Link, and Hardware. The team assignments and members are listed in Table 2.1 below.

Table 2.1. TracSat Subteam Organization

Subteam	Responsibility	Members
Laser Tracking Team	*Laser Tracking Subsystem (LTS):* Motorized laser source/detector unit, tracking system, processing power choice	Braden Anderson Wellington Froelich Yash Pahade Ben Ranft
Laser Link Team	*Laser Link Subsystem (LLS):* Laser communications link, PCB for cold gas thruster valves	Sandeep Baskar Josh Fitch Alec Leven Dante Sanaei
Hardware Team	*Propulsion and Levitation Subsystem (PLS):* All hardware of CubeSat and GS, pneumatics, air bearings, gas tanks, manifolds, cold gas thrusters.	Robert Miller Pat Pesa Bethany Price

2.4.2 Meetings

The team met three times a week as a full group. Two hour long class sessions were run each week; Monday lectures were used for a presentation on a technology relevant to the project, while Wednesdays were used as a chance for the subteams to present their weekly progress reports. Lab time was reserved on Tuesdays for three hours for members to work on their subsystems. Additionally, each sub-team took time to meet separately and work on specific technical tasks.

2.4.3 Communication and Documentation

The team communicated through Slack primarily as well as email for important team-wide announcements. Important files were uploaded to Blackboard while most other files were maintained on Google Drive. The CAD files were stored and maintained on a Purdue network for ease of access and configuration management. Finally, code was stored on GitHub. GitHub allowed for several branches of the code to be made; this was useful for providing a central repository of code while allowing for multiple people to edit different versions of the code.

2.4.4 Presentations and Design Reviews

Large formal presentations such as the Preliminary Design Review (PDR), Critical Design Review (CDR), and Test Readiness Review (TRR) were held to give updates to the professor and our sponsors/clients at General Atomics. A video was made in conjunction with this paper as a final deliverable.

3 LASER LINK SYSTEM (LLS)

3.1 LLS INTRODUCTION AND OBJECTIVE

The creation of the Laser Link System (LLS) is an updated form of the Electronics Team that is focused solely on the task of facilitating laser communication between relevant interfaces of the TracSat operation. As mentioned, a key objective of the TracSat project is to achieve live video downlink from the CubeSat onboard the testbed while the CubeSat is in movement. This downlink (as well as uplink) is achieved with a laser communication system ("LaserCom") which allows interaction between the CubeSat and Ground Station systems. Thus, the purpose of the Laser Link System is to provide a controller, user interface, and system architecture to transmit various telemetry between the CubeSat and Ground Station. This overall purpose was determined to be achieve if the following requirements were met:

1. LLS shall provide uninterrupted uplink and downlink communication throughout the movement of the CubeSat
2. LLS shall be transmit the data at the rate no less than 50 kHz
3. LLS shall transmit live video during downlink

The first two of these requirements were set early on in the project when the existence of the laser communication system first began. Early on, a basic system (using Raspberry Pi's) was created that could send data through pulsed laser flashes from a simple laser to a receiver. In this condition, small packets of information could be encoded into binary and sent. This system formed the basis of the LLS system and the continuing goal for the team was to improve both transmission speed and transmission size capabilities. At the end of the Fall Semester, the team was able to reach speeds of 1 kHz through an Arduino based system (which is currently the highest performing system in operation). With this system, the first requirement was met after testing with the TestBed Car which had a receiver onboard and the laser was set aside (acting as the Ground Station). In this test, simple directional commands could be sent to the testbed to control its movement in translational and rotational capacities. Furthermore, the purchasing of the PYNQ-Z1 Microcontroller, more responsive photodiodes, and higher quality lasers were done to meet the second requirement. Essentially, this goal is limited by hardware capabilities as the software that underpins the Laser Communication system has been adequately streamlined. Thus, to achieve

higher transmission rates (eventually beyond 50 kHz), a hardware system needs to be operated that can physically run at those rates. With such a system designed, the last requirement can start to be approached. The idea of live video during downlink was suggested later in the project as a natural final progression of the Laser Communication system to its highest performing capability. In the following section, the relevant concepts, theories, and products that went into achieving these LLS requirements will be presented.

3.2 THEORY OF LASER COMMUNICATION

In the early stages of the Laser Communication project, the natural solution to the issue of transmitting data was quickly brought up — that laser pulses could transmit simple binary data by turning on and off. By programming a microcontroller, a laser could be controlled to turn on and off to represent a value of '1' or '0' respectively. For instance, a simple word could have its letters converted to integers and those integers would be converted into binary.

Eventually, this concept became adapted into using a specific type of laser to better achieve this goal. Transistor-Transistor-Logic (TTL) is a specific type of laser modulation for some laser sources. With TTL lasers, the source can only be operated in two states: on and off. Shown below is a visual representation of the laser modulation used:

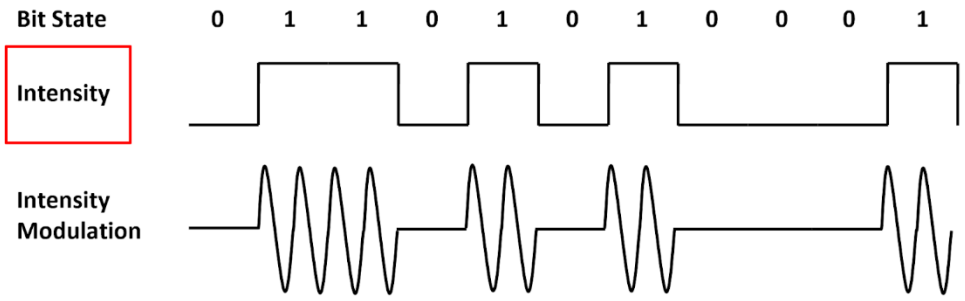

Figure 3.1. Diagram of TTL Modulation

Physically, a key difference between TTL lasers and other basic lasers is the existence of a third wire that allows for this modulation. The laser would be powered through a power and a ground connection, and the third connection would solely control the "on" or "off" state of the device. Without a TTL laser, replicating this intensity modulation process would mean sending the electrical pulses from the microcontroller into the power connection of the diode. Instead, the third connection of the TTL will allow the laser to reproduce these electrical pulses while keeping laser

circuitry powered during the process. This is a capability that allows for TTL lasers to reach transmission rates within the megahertz range. Additionally, TTL lasers are generally cheaper than lasers with more modulation functionality since the system is relatively simpler. Thus, for a standard price, TTL lasers can be bought that achieve 1 MHz transmission speeds. This standard is far beyond the mission requirement of 50 kHz; however, to achieve video downlink, such speeds are needed. The context of this laser within the entire transmission system is as expected. Shown below in Fig. 3.2, the laser allows pre-converted data to be sent to a respective receiver (that has been linked in a separate process).

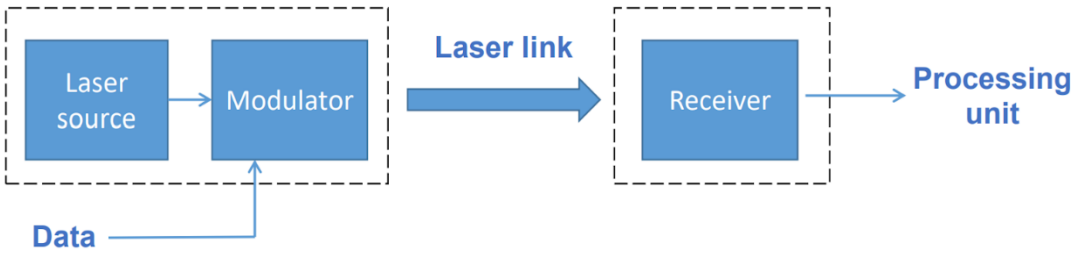

Figure 3.2. Diagram of Laser Communication System

After the individual pulses have been detected by the receiver, which for the LLS system is a photodiode, the flashes are then converted back into readable data using a processing unit. For the LLS system, respective algorithms have been designed that achieve this binary conversion, reconversion, and control over the laser pulses. A more detailed view at how these tasks are accomplished (namely the algorithm governing sending and receiving pulses) will be explained in Sec. 3.5.

3.3 DATA AND TELEMETRY

Table 3.1 illustrates the planned evolution of the LLS capabilities. The full success criteria for the LLS is to transmit 240p video at 10 fps. This is an ambitious end goal that required logical progressive steps of higher complexity levels. This is shown in the below table, as the first level of telemetry to achieve is basic test information. This level was achieved successfully with the Arduino LLS system and served as a proof-of-concept for achieving basic laser communication. The next level of telemetry planned was sending commands to be uplinked to the CubeSat. This level was partially successfully demonstrated by the Arduino system, as testing at the end of the Fall 2019 semester culminated in a simple start command sent from the very basic ground station

Table 3.1. Progression of Telemetry Capabilities for LLS

Level of Complexity	Telemetry Contents	Data Rate Required	Additional Challenges
1	Basic test information (e.g. words, simple bit string)	< 1 Kbps	N/A
2	Commands to CubeSat (e.g. translate, stop, send specific data)	~ 1 Kbps	Requires developed Ground Station user interface
3	CubeSat state information (e.g. translation speed, rotation rate, power levels)	~100 Kbps	CubeSat must be integrated with primary avionics sending information to LLS
4	CubeSat Downlink video at varying levels of quality	> 1 Mbps	Requires little-to-no packet loss and very high data rates

set-up to the CubeSat which instructed the CubeSat to begin a propulsive maneuver. To execute more complex commands and send more detailed information a more sophisticated ground station was required. This ground station was developed during the Spring 2020 semester but was not able to be fully tested. The next level of telemetry is downlinking CubeSat state information. This level of telemetry was not achieved due to both an inability to achieve the 100 kbps rate with the Arduino, the issues with the PYNQ, and an avionics system unable to collect and organize the required information. Finally, after these three levels of telemetry had been achieved, video transmission at various levels would have been attempted. This was not achieved due to the same reasons as the CubeSat state information. Details on the progressive video levels is described in Table 3.2 below.

Table 3.2. Telemetry Details for Different Video Levels

Type of Image	Size of Pixel Matrix (bits)	Transmission Time at 1 Mbit/s	Data Rate Required for 15 fps
Uncompressed High Quality	18,864,000	18.86 s	282.9 MHz
Compressed Low Quality	754,560	0.75 s	11.25 MHz
Ultra-Compressed Black and White	94,320	0.094 s	1.41 MHz

Table 3.2 expands upon Table 3.1 by explaining the underlying details of each video level. The simplest and easiest video type is compressed black and white. This both reduces the number of pixels due to compression as well as reducing the complexity by eliminating color. This level of

video would have required 1.41 MHz, still well beyond the capabilities achieved with the Arduino LLS. The next levels of video include color, increasing the pixel number, and go from compressed to uncompressed imagery. The three levels of video transfer each require more than an order-of-magnitude increase in the data rate required for 15 fps, culminating in nearly 300 MHz. This data rate is highly ambitious and is a good goal for future TracSat teams. Future TracSat teams should continue to develop the PYNQ as the primary platform for laser communications. The PYNQ architecture is built to handle data rates of up to 50-100 MHz, which should enable compressed colored imagery and could possibly be pushed to achieve full quality colored images at a lower fps. This would be considered a massive success and should be pursued in the future.

Fig. 3.3 displays the contrast between each of the video levels described in Table 3.2 The Purdue logo does not fully display the disparity between each quality level. Black and white images with images of the CubeSats surroundings would be highly difficult to discern due to the lack of distinct color or even grayscale. Additionally, with less crisp images the difference in quality between the compressed and full quality images would be highlighted by blurry and low quality images of the CubeSats surroundings.

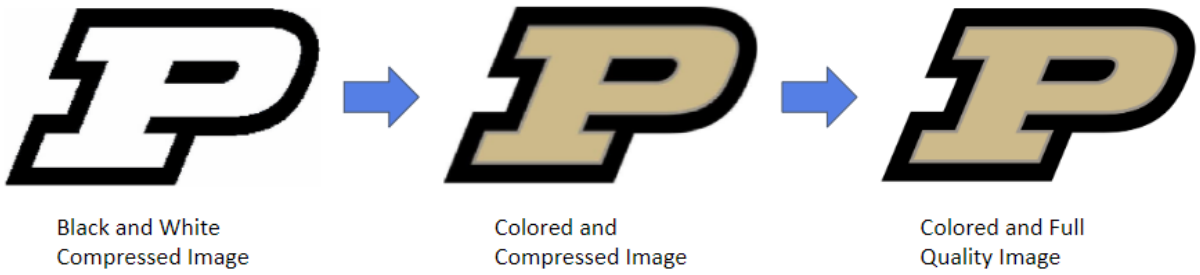

Figure 3.3. Progression of Image Quality

3.4 HARDWARE

3.4.1 Laser

The laser is one of the most critical aspects of the entire system, but fortunately it has also proven to be one of the more simple and straightforward hardware components. From the principles described in Sec. 3.2, a laser that can pulse at frequencies up to 1 MHz is required in order for the mission objectives to be completed, but there are other design aspects of the laser that are important to understand.

Due to the relatively short range from the ground station to the CubeSat testbed, laser intensity drop-off effects can be ignored. This means that most commercially available, low power lasers that are intended for circuit use can be used in the system (for low transfer rates). The upper limit for laser power allowed in a non-certified lab at Purdue University is 5 mW, and all of the lasers purchased were under this limit. While a TTL laser was eventually used in our efforts to increase the frequency of intensity modulation, most of the initial testing and design prototypes used normal, non TTL lasers. These lasers had a power and ground connection, and were able to directly link up to the microcontroller. Instead of sending TTL pulses through the third independent wire, the electric pulses were sent into the power connection, thus powering the entire laser system for an "1" pulse and depowering it for a "0" pulse. The laser utilized for the software development and Arduino testing was the KY-008 Laser Transmitter Module, seen in Fig. 3.4.

Figure 3.4. KY-008 Laser Transmitter Module

While this initial laser was very important in the development of the fundamental architecture of the system, the laser diode was not intended for high speed intensity modulation. To improve the system's capability, though, a new laser selection process took multiple factors into account as shown in Table 3.3. below.

Table 3.3. TTL Laser Options

Company	Max TTL Frequency	Supply Voltage	Power	Wavelength	Price	Notes
Adafruit *(USA)*	50 kHz	5±0.2 V DC	5 mW	650 nm	$18.95	Meets temporary requirement of 50 kHz
Civil Laser *(China)*	1 MHz	3-5 V DC	5-90 mW	650 nm	$49.00	Limited shipping due to COVID-19 restrictions
Roithner Laser *(Austria)*	1 MHz	3-5 V DC	5 mW	635 nm	$66.47	Minimum 50-piece order
FLC *(Germany)*	1 MHz	5±0.5 V DC	1-50 mW	630-960 nm	$65.00	Several specification options

The key factors in making this decision were based on each laser's respective maximum TTL frequency and price, while design factors of power and voltage could be easily adapted into the electronics budget of the system. For the system, the wavelength of the laser is not a mitigating factor because it will be modulating on and off. Since red lasers are generally cheaper than other colors, diodes in this wavelength range were looked at to meet our cost requirements. Additionally, the laser needed to have appropriate physical dimensions to actually fit in the planned LLS design, so only lasers near the size limits of the Adafruit 1054 were used. Fortunately, most lasers with similar power capacities are similarly sized, so this was not a severe issue. The last key factor in laser selection was the ability for the laser to be focusable. In previous tests with the 1054 and other simple laser diodes, a large distance would cause the laser beam to diverge and create a large diffuse point on the final surface. For these cases, it was found that the a diffused beam caused complications with the photoresistor/photodiode and the laser communication software to accurately detect flashes. In particular with the photoresistor, the laser needed to be directly pointing at the receiving surface or else communication would not be possible. Thus, the laser needed to have the capability of a focusable lens in order to sharply tune the laser beam to ensure the receiver could clearly receive signals.

Figure 3.5. Adafruit 1056 TTL Laser

To proceed with simple testing, the team decided to select the Adafruit 1056 TTL Laser Diode that had the capability of 50 kHz. At the time, many of the other laser options were not available due to outside factors (such as COVID-19 import restrictions), so the 1056 laser acted as a temporary laser to test preliminary data transfer with the PYNQ-Z1. Furthermore, the 1056 laser has similar dimensions to the original 1054 laser, so previous structural mechanisms were not

changed significantly. However, to achieve the long-term objectives of video transmission, lasers within the megahertz range are needed. To this end, the team believes the Civil Laser 64/64 TTL Laser Diode would be an optimal selection. At the current time, this choice is not available due to COVID-19 regulations as Civil Laser is based in China. When regulations become more relaxed, this laser will obtain an ideal transmission rate for a reasonable price (compared to other options). Likewise with the Adafruit 1056 laser, the 64/64 has a similar physical and structural design, so it will be simple to update the LTS tracking system.

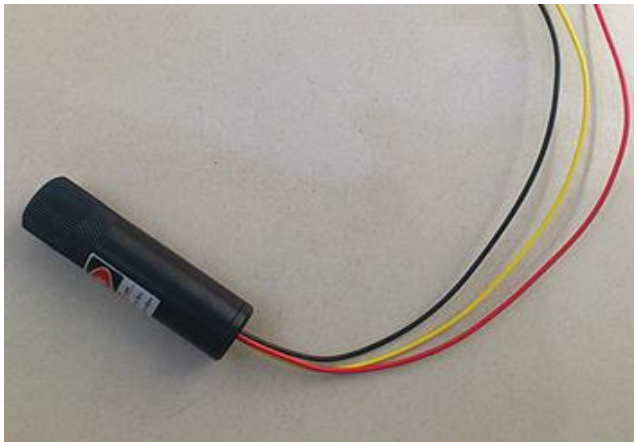

Figure 3.6. Civil Laser 64/64 TTL Laser Diode

3.4.2 Receiver

The next vital aspect of the LaserCom hardware is the laser receiver. In order for communication to occur, a device must convert the concentrated laser radiation into an electrical current that will allow the microcontroller to convert the optical modulation into interpretable data. Additionally, the perfect receiver must be able read spikes in light intensity at near instantaneous speeds, as delays in the process can cause irreparable damage to the data.

The cheapest and most plentiful device that satisfies these basic requirements is a photoresistor, or light dependent resistor (LDR). This device acts as a resistor with a variable resistance, and the light it senses has an inversely proportional relationship with the resistance it produces. In other words, as the intensity of light increases, the electrical resistance of the device reduces to near zero. Most market photoresistors have a “dark” resistance value of upwards of 1 MΩ and a “light” resistance of 5-100 kΩ. When the photoresistor is integrated in a simple powered circuit (seen in Fig. 3.7), modulations of the light intensity can be detected in the current, and when inputted into an analogue device (such as an Arduino), can be interpreted as binary.

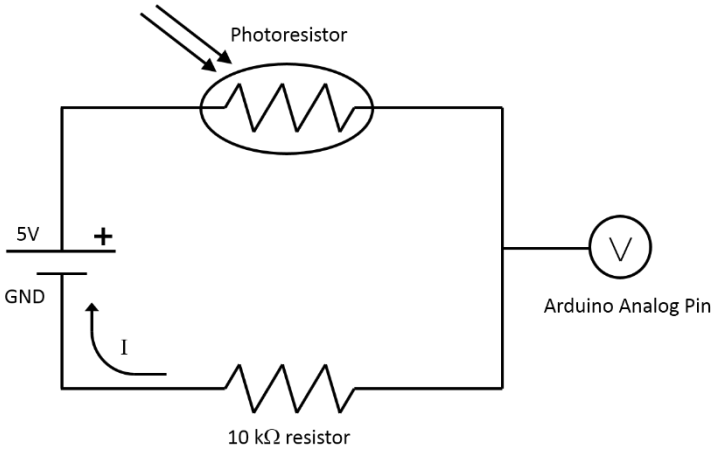

Figure 3.7. Simple Photoresistor Circuit

Although the photoresistor is a great device for early prototyping and testing, it has one major flaw that disqualifies it from the final designs: the response time is far too slow. On average, most photoresistors have response times in the millisecond range. While this is quite fast to the casual observer, the LaserCom system itself will be running at frequencies up to 1 MHz, thus requiring response times several magnitudes faster than the system itself. At low speeds of even 50 ms, the receiver will not be able to accurately interpret the modulations from the laser that are being transmitted at speeds up to 1 μs. Not only does the slow nature of the photoresistor foil our goals, the design of most are not applicable to be included in the final system. Figs. 3.8 and 3.9 show two photoresistors that have been used in initial tests, and their small light receiving surface areas greatly amplify the challenge of tracking and acquisition of the lasers. It is simply not effective to use such a small receiver when the CubeSat is in a dynamic and unpredictable environment. It is vital that the final design contains receivers that are large enough for a margin of error.

Figure 3.8. EBOOT Photocell

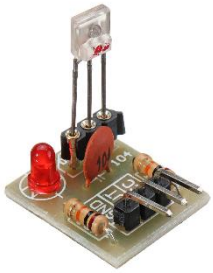

Figure 3.9. DZS Laser Receiver Sensor

While the photoresistor cannot be used in our system, it has proved to be very useful in the design and testing of the main and auxiliary software of the system. Because of the highly scalable nature of the integration and ground station GUI software, 1 Mbit/sec speeds are not necessary for initial builds of the code; thus, all of the software listed in Sec. 3.5 were tested with a photoresistor acting as the main receiver of the system.

While the photoresistor is the cheapest and most accessible option, there are other types of receivers on the market that better fit the requirements of our design. There are several other electrical components that provide the same light detecting service, these include the photodiode, phototransistor, photojunction, photocell and others. Before settling on our final device, we also attempted to use a photocell. This is a miniaturized version of a common solar panel, and it is the device with the most surface area which would majorly benefit the tracking portion of the mission. Unfortunately, the solar cell proved to be ineffective due to a highly noisy analogue output and slow response time. The photocell is fundamentally different from the photoresistor as it produces current instead of blocking it, and while the solar cell didn't work, it has a sister system that utilizes the same method: the photodiode.

The photodiode is a light sensing diode that, like the solar cell, produces electric current proportional to the intensity of light. Importantly, it does not use the photovoltaic effect like the solar cell, instead the semiconductor system produces current in a much faster, accurate and less noisy way. This device has response times above 1 ns and usually are built with enough receiving surface area so the LTS team can have a large enough margin for error. The circuit used in a photodiode system is nearly as simple as the previous photoresistor system and it can be seen later in Fig. 4.3.

For use in the LTS system, there were two photodiodes considered to help improve the accuracy of the Laser Communication system: DigiKey SD 445 and ThorLabs FDS1010. Shown below in Table 3.4, the performance specifications of each were compared against each other for relevant operating features.

Table 3.4. Photodiode Receiver Options

Company	Response Time	Responsivity	Wavelength Range	Active Area	Price
DigiKey	13 ns	0.55 A/W	350-1100 nm	100 mm^2	$75.09
ThorLabs	< 100 ns	0.725 A/W	300-110 nm	100 mm^2	$55.73

In comparing the operating performance of the two models, both have similar specifications – the main difference arising with the price. Since the Laser Tracking System requires ten photodiodes (five for each LTS array), a difference in price for a single unit would have a sizable effect for ten. Another factor in the selection process came with the physical design of each photodiode. For the DigiKey model, there is a large border around the actual active area, seen below in Fig. 3.10.

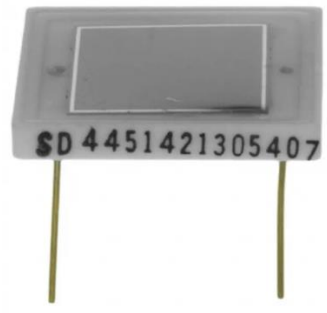

Figure 3.10. DigiKey Photodiode

Since the LTS array is built to have five photodiodes in linear succession (seen in Fig. 4.2 in Sec. 4.2.1), this large border would entirely defeat the purpose of the array. If the laser array with the DigiKey photodiodes were to rotate, then there would be some space where the detectors are not receiving any input and thus its angular orientation could not be ascertained for tracking. Thus, it was decided to use the ThorLabs FDS1010 photodiode shown in Fig. 3.11 as its design removes the outer border and creates a more-uniform tracking array.

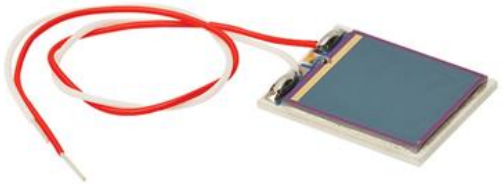

Figure 3.11. ThorLabs Photodiode

3.4.3 Microcontroller

The laser communication system has a unique requirement of high speed and ultra-accurate computing; thus, the selection of the microcontrollers is of ultimate importance. Most communication systems in the world utilize custom built electronic circuits to perform the rapid data transfer required, but due to the lack of circuit design experience within the team, an alternative programmable system was created. While this allowed the team to create LaserCom without spending several months self-studying highly complex data transfer circuits, it did pose a new issue of finding microcontrollers that were able to satisfy our goals.

First, it is important that the controller execute the commands within the laser/receiver codes at extremely high speeds in order to properly send and receive data. The lower the execution time of the programs, the faster the transmission rate can become. Therefore, high level, interpreted languages such as Python cannot be used for the laser or receiver. Additionally, controllers like a Raspberry Pi cannot be used due to their large amount of system overhead; the Pi contains an entire Linux operating system that slows down the entire system. Due to the lack of testing of a wide array of controllers, more specific requirements are unknown, but we hypothesize that the best controllers are those that process low level code with no operating systems or other large overhead architectures. Most likely, the laser and receiver would require their own independent controller to reduce the amount of processes running at one time, thus increasing execution times.

3.4.3.1 Arduino System

The Arduino system was the first microcontroller that allowed for data transfer at rates that were acceptable to the mission. We were able to obtain one to ten kilohertz intensity modulation frequency which netted us one to ten kilobits per second of data transfer. The system used two Arduinos at each transfer endpoint (CubeSat and Ground Station), one Arduino for the laser, and one Arduino for the receiver. This was done in order to reduce the execution time of the code, thus increasing transfer rates. Each Arduino was linked to its endpoints (either ground station or CubeSat) through serial connection. Commands, data or messages would be sent into the laser Arduino through its serial port to be transformed into laser pulses and the receiver Arduino would convert its received laser pulses back into readable data and export it through the serial port. This serial architecture is further described in Sec. 3.5.3.

While the Arduinos were able to successfully send and receive data and commands at transfer rates high enough to be acceptable for the mission, it soon became our goal to expand the system to higher rates. Once these new objectives were taken on, it became apparent that the Arduinos could not continue to be a part of the TracSat LaserCom system. There were several reasons for our departure from an Arduino based control scheme, but the most apparent reason is that these devices can not accurately send electric pulses at the speeds required to send telemetry that requires ultra-high-speed transfer rates (beyond 1 MHz). While the laser and receiver code were not heavily optimized and are not running at maximum efficiency (interrupts and other advanced features were not included), research has shown that speeds beyond 100 kHz is simply not possible no matter the code efficiency. Therefore, we must either look toward designing our own circuitry or employing more advanced controllers.

3.4.3.2 PYNQ System

In order to achieve these improved transmission rates, the team explored different microcontrollers that could operate at higher speeds than the Arduino-based system. The ultimate goal of this investigation was to find boards that would have multiple fast Digital Inputs (DIs) and Digital Outputs (DOs). These input and output channels would be used to drive laser sources and photodiodes and subsequently send and receive data at the required improved rate (> 1 MHz). Another point of consideration was the compatibility of the board with the existing code that had been written for the Arduino system. Since Arduino is coded in C, the optimal selection would be a board that could be coded similarly so that LaserCom software did not have to be rewritten or intensely rewritten. Similarly, for the code to stay constant over this transfer, it was desirable to have a board with as many (or more) I/O pins as the Arduino. Regarding physical requirements for the board, it was agreed that the board needed to fit the form factor of the CubeSat such that it would be sized correctly. As this section of hardware had already been developed, the size constraint could not be altered. Likewise, the board needed to satisfy the power requirements set by the Hardware team.

Initially, the team looked at the Intel D2000 board which had the key features of a 32 MHz clock speed and a similar architecture to the Arduino. However, upon the suggestion of a relevant industry professional, the microcontroller chosen was the Digilent PYNQ-Z1 board (shown below). Some important capabilities that were used to make this decision include the PYNQ's 8 high-frequency IO PMOD ports, high frequency Arduino Shield Variable, an acceptable power range (7-15 V), and a familiar Arduino GPIO pin structure. However, perhaps the most important factor in this choice was that the PYNQ board is capable of running in the range of 50-100 MHz. This range would easily meet the transmission rate requirements of exceeding 1 MHz and subsequently meet the standards necessary for Level 4 Telemetry (as outlined earlier in Sec. 3.3).

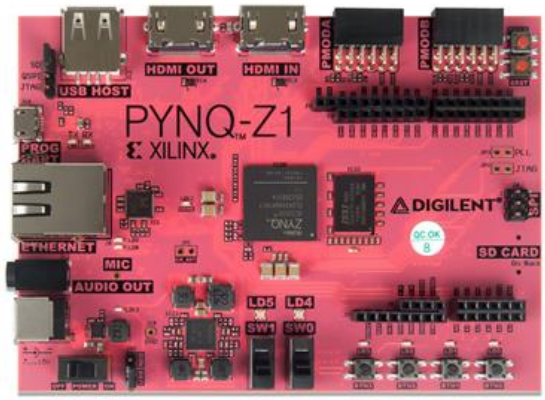

Figure 3.12. PNYQ-Z1 Microcontroller

Shown below in Figs. 3.13 and 3.14, the I/O pins were initially planned to have all relevant functions on board the CubeSat and Ground Station to be run by a respective PYNQ board. For instance, in Fig. X, the CubeSat will have its solenoids, detectors, and servo run by a single PYNQ board. Here, the PMOD's will be dedicated to laser I/O with power being supplied from board through VCC/Ground connectors on PMOD. For this system, the laser diode is considered a peripheral in this system. Additionally, due to the number of pins, there is capability for this to expand to control the transmission of live video from a camera attached to the system.

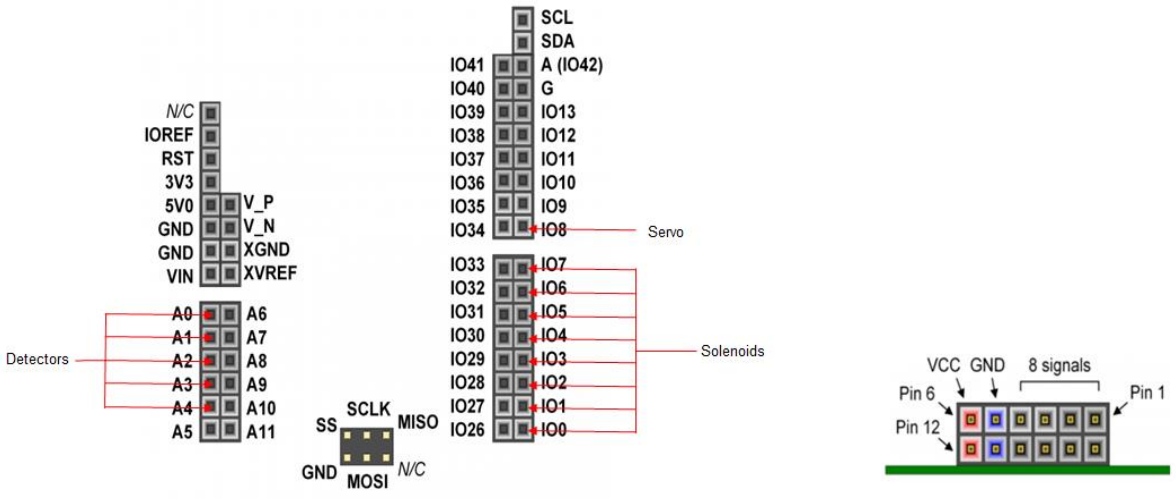

Figure 3.13. CubeSat I/O Diagram

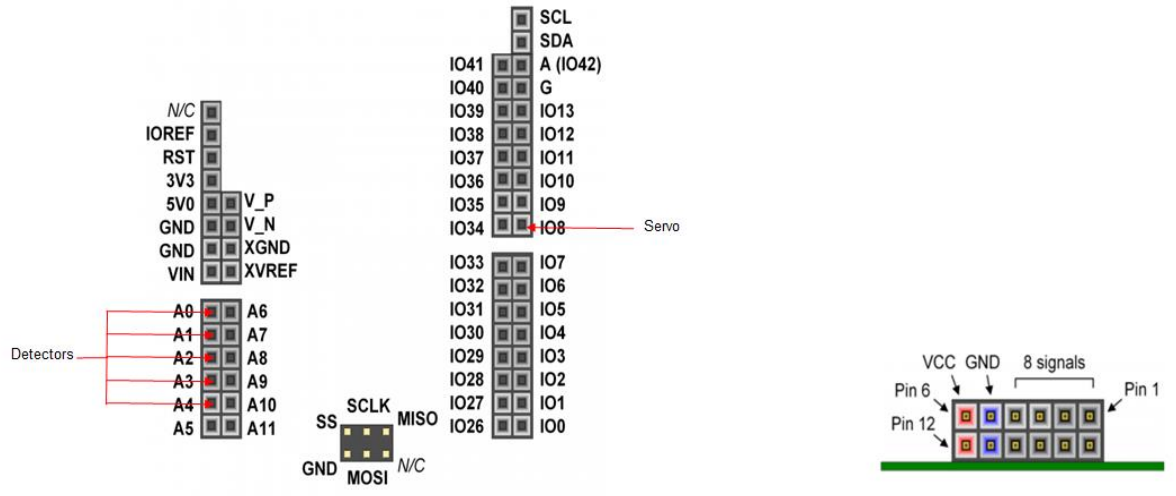

Figure 3.14. Ground Station I/O Diagram

However, some complications arose with the actual setup of the PYNQ board. To instruct the reader of how to operate the board, some instructions are discussed here, and more can be accessed on the Digilent website or on pynq.readthedocs.org.

Firstly, to activate the board, the user needs the PYNQ power cable, an Ethernet cable, a USB to USB-C cable, and a microSD card. Firstly, the microSD needs to be prepared by downloading the PNYQ-Z1 image from ww.pynq.io and then writing this to the SD card with Win32 Disk Imager. Next, ensure that the PYNQ jumpers are correctly oriented. For this, set the jumper (JP5) to REG (upper two pins) if using an external power source and USB (bottom two pins) if using a USB cable. For this project, since there are laser peripherals being added, the external power source is recommended. Next, set the boot mode setting by placing the jumper in the SD setting (upper two pins). This configures the board to boot from the microSD card. Next, connect the ethernet, USB, and USB-C cables to the PYNQ board.

With direct connection between the ethernet port of the PYNQ board and a computer, you need to manually configure the board to have a static IP (connecting to the IP of the board). This can be done by setting properties of Internet Protocol Version 4 (TCP/IPv4) such that the IP address is `192.168.2.1`, the subnet mask is `255.255.255.0`, and the preferred DNS server is `127.0.0.1`.

When the board is turned on, the power LED will be turned on signifying the board is on. After some time, the board will boot when the two blue LEDs and the four yellow-green user LEDs flash. This indicates that the board has booted successfully, and you can connect to Jupyter Notebook, the application where coding is done on the PYNQ board. As the board is connected directly to the computer, you can access the Jupyter Notebook software by typing in `http://192.168.2.99:9090` into a Chrome Browser and using the password “xilinx”. In Jupyter Notebook, the relevant programs can be accessed as they are already loaded onto the PYNQ boards.

After the Jupyter Notebook has been set up, the board can be programmed. However, the GPIO pins that are connected to the laser and receiver need to be programmed separately. To activate the GPIO pins, download and install VIVADO software which is the environment where individual pins can be assigned values and labeled. In this regard, there was a lot of complication in understanding this process as VIVADO is coded with a different computer language, VHDL, and necessitates a particular knowledge base of electrical and computer engineering skills.

Understanding the IP integrator block design and other aspects of VIVADO required a considerable amount of effort as the team was not well-experienced in this field. To solve this, it is recommended that users are well adapted with the software by either reading documentation on VIVADO or using a programming tutorial by XILINX. As a consequence of the team's diversion in first understanding the VIVADO software, the Laser Communication code programmed on the Arduinos was not able to be easily transferred over to the PNYQ. This transfer was also exacerbated by the fact that the PYNQ is programmed in Python whereas the Arduino code is in C. As explained in the software Sec 3.5, two boards are timed to run concurrently and send/detect laser signals for the same incremental timestep. Also, the Arduino system had several built-in functions that simplified the code and facilitated the communication process. The difficulty of the code's C-to-Python translation is unknown, but it is known that several sections of additional code would be required to assist this process.

For future expansion, as noted, it would be incredibly beneficial to have users who are already experienced with PYNQ or similar high-level microcontroller coding. For such experienced students, this translation process can be more efficiently achieved, and laser communication could be transferred to the PYNQ. To achieve higher levels of laser communication complexity, the PYNQ board theoretically serves as the optimal choice purely on the basis of its medium-ranged megahertz capabilities. However, there are numerous complexities involved in the actual programming process itself to achieve such a feat. Nevertheless, the PYNQ board, purely for its transmission capabilities, is the optimal choice to implement video transmission for future tests.

3.4.4 Full Design

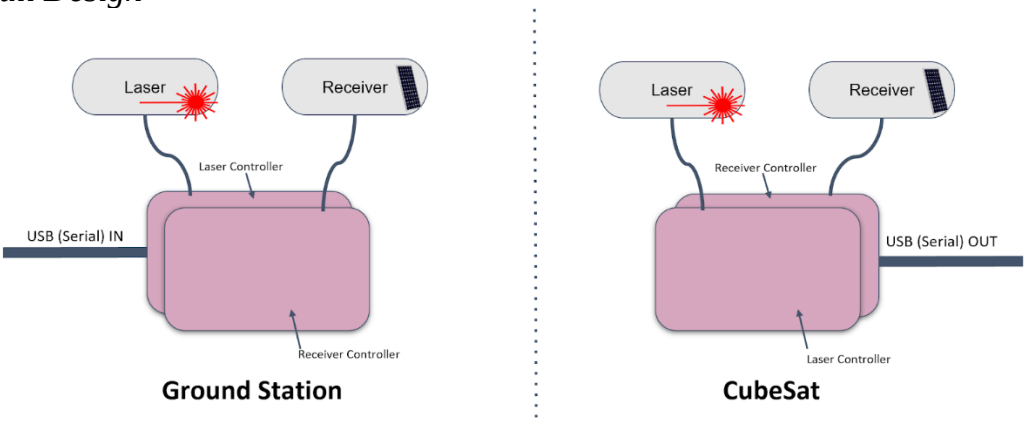

Figure 3.15. Full LLS System Diagram

In the current iteration of the Laser Communication code, the laser and receiver code are run simultaneously but needs to be done on separate microcontrollers. As explained in Sec. 3.5, the code is based on a laser and receiver pair running on the same clock intervals for each packet of data. Thus, even with the capability of parallel processing, a single controller would not be able to operate both the laser and receiver. The final design, consisting of two boards for laser and receiver is shown on the previous page in Fig. 3.15. In conjunction to these two boards, there will be a PCB board that contains all the electronics which will be stacked on top of the other microcontrollers. As this final "LaserCom Assembly" consists of three boards, it will need to be tested for form fitting to ensure it can fit on both stations (particularly on the CubeSat).

3.5 SOFTWARE

Throughout this section, all code was written using the Arduino as the main controller of the LaserCom system. The laser and receiver code were specifically written for the Arduino, but the integration code (LaserCom class and GUI) can be used on any Ground Station device, but will need some editing if the Arduino are no longer the main controllers. It is also important to note that *live* data streaming was not accomplished this semester, and the code below only works for short commands. All code can be found at the Purdue TracSat Github and in Appendix A.

3.5.1 Laser

The laser code was written in the custom Arduino framework (built upon C/C++), but the fundamental logic and algorithms can be implemented for any controller in any language to produce similar results. The main objective of the laser controlling Arduino is to ingest data from the ground station or CubeSat (this data can be commands, telemetry, video, etc.) and convert it into binary, then produce electrical pulses that control the laser. As stated in the introduction to the software section, the following code does not work for live data, but instead it was built upon the idea that a singular string would be sent through the LaserCom system, but through small adjustments, live telemetry can be accomplished.

Fig. 3.16 is a flowchart describing the basic logic of the laser code, and it displays the process of how each of these tasks are accomplished. First, the data streams into the controller from an auxiliary device, and the code will immediately convert it into an array of binary. Every character in the initial data input is converted into a corresponding digit (in ASCII), and that number is then

converted into a series of bits. The bits from the entire input string are stored in the binary array. The setup is now complete.

The program now loops through the entire array, sending 8 bits at a time. Before every byte (containing 8 bits), it sends an *Initialization Pulse*. This pulse is vital to mitigating error in the system, and it will be further explained in the receiver code section. After every loop, there is a time delay function in the code. This delay corresponds to the data transfer rate of the LaserCom system; the smaller the delay is – the faster the transmission rate becomes. It is vital that the receiver and laser code have identical delay values in order to sync up perfectly.

Lastly, unless the board is reset or a stop command is inputted, the code will continuously loop through the binary array – thus sending the same message indefinitely. This is an optimal feature for several reasons. First, if the connection is interrupted or lost, the message will still be received after reconnecting to the receiver. Also, it will allow the CubeSat code to incorporate an error checking system; in earlier demos, the LaserCom class included a function `receiveData()` that ensured there were no errors in the command by only receiving the string if it matched the last three inputs. For example, if the following strings were received in quick succession: "`test`", "`test`", "`tet`", then it would detect the error and not read the incorrect "`tet`". Due to the high data transmission speed, this added time the error detection system costs are negligible.

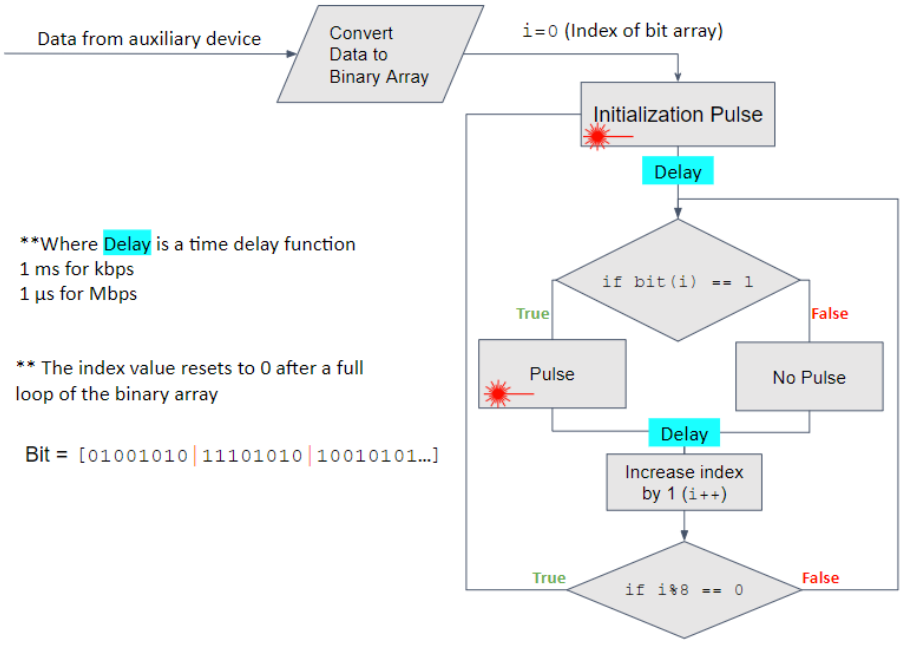

Figure 3.16. Logic Architecture of LLS Laser Transmitter

The code also has several features specifically built for the user experience during testing within the Arduino program. It is important to note, this is only required when debugging or testing the Arduino, and the message sending process is automated in the LaserCom class. Every time the board starts up or is reset, the serial monitor will prompt the user to input data (seen in Fig. 3.17). The user will then input the command or message inside of a start character and end character. For example, the message "tracsat" would be inputted as "<tracsat>". The serial monitor will then display the binary conversion for each letter (seen in Fig. 3.18). Lastly, inputting '0' will reset the laser and allow for a new command. Automatically, the '#' character will be assigned at the end of the inputted string, and this acts as a new-line flag for the receiver.

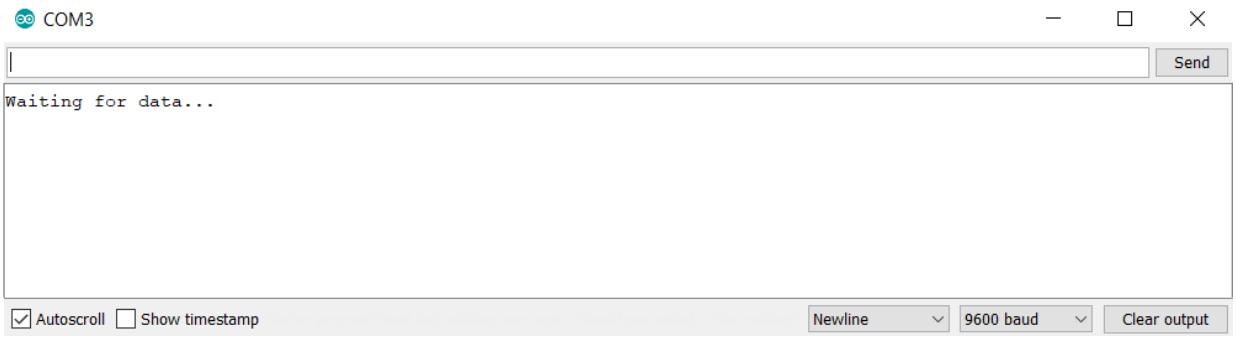

Figure 3.17. Serial Monitor Prompt Screen

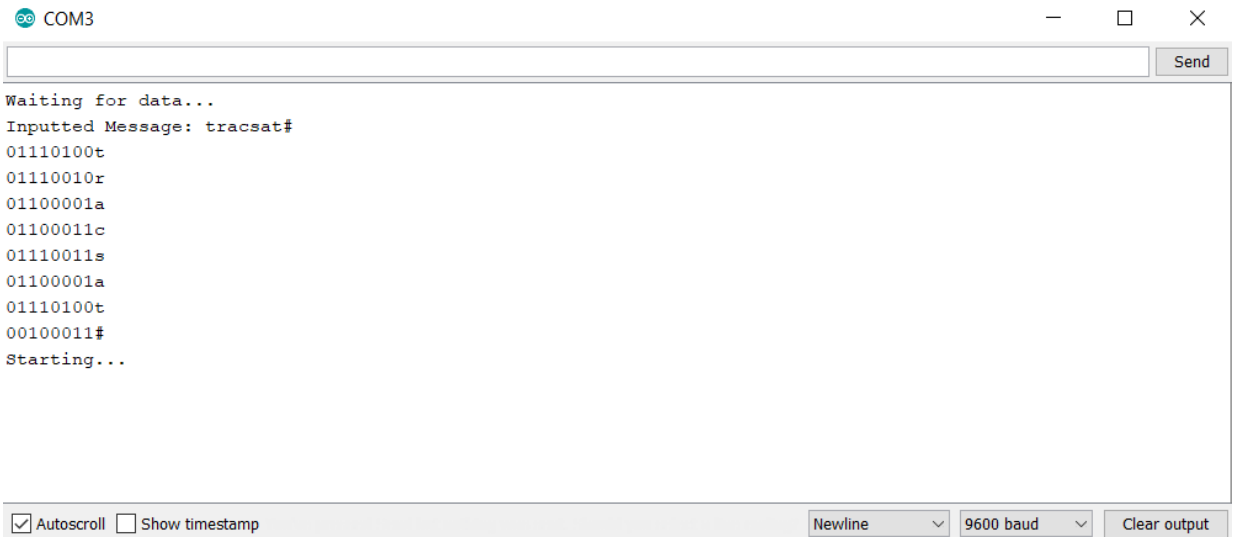

Figure 3.18. Serial Monitor Binary Conversion

3.5.2 Receiver

The receiver is built upon a similar principle as the laser, but instead of sending electrical pulses, it receives them and interprets them into readable data. Just like the laser code, this program is built specifically for the Arduino, but the base logic can be applied for any controller system. A flow chart outlining the logic can be seen in Fig. 3.19.

When the board is initially turned on, the code gets a preliminary reading from its detector in order to calibrate the system. The starting light level value, along with the small error threshold, becomes baseline ambient value that is used to compare all analogue inputs.

After the setup process, the board enters an *Idle State*. In this mode, it continuously loops looking for the *Initialization Pulse*, as long as this pulse does not occur the code will remain in the idle state. After this pulse is detected the code will loop eight more times and detect a full byte of data. If the input is greater than the ambient baseline it records a '1', and if it's under the ambient reading it records a '0'. The reason for the initialization pulse becomes evident when diving deep into the logistics of error mitigation. The two main sources of error will be covered in more depth in the next section, but the initialization pulse mitigates the "phase shift" error. This error occurs when both the laser and receiver boards are looping at the exact same frequency for extended periods of time. Due to the nature of the requirements of each code, the execution times for one receiver and laser loop are very slightly different. If the two boards run in parallel for long periods

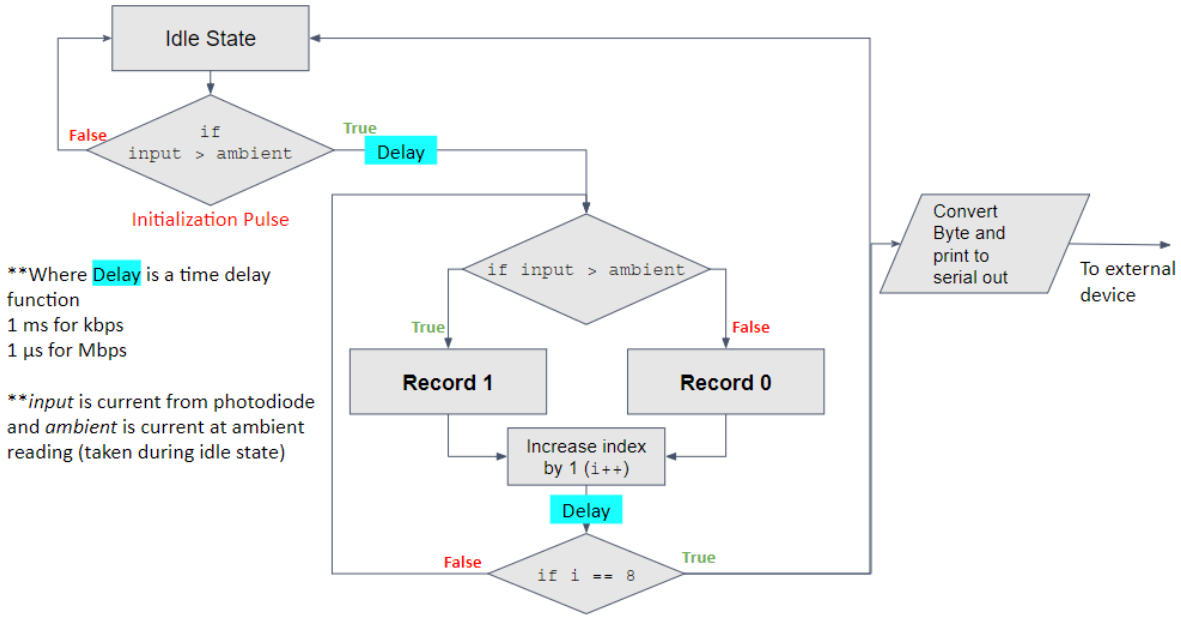

Figure 3.19. Logic Architecture of LLS Receiver

of time, this execution time lag will eventually cause the two boards to no longer be in sync. In other words, the minor execution time lag will cause a slow buildup of a phase shift in the modulation. To remove this source of error, you can reduce the amount of time the two boards are running in sync. This is where the initialization pulse comes into play; by only looping 8 times (recording 1 byte) before returning to the idle state, you reduce the amount of time spent in sync with the laser board. Instead of spending minutes reading all of the data in one long-iterating loop, you return to idle after every byte and wait for the next initialization pulse – thus re-syncing the boards after every single byte. This is a confusing system and we recommend more self-study into the algorithm.

After each byte is read and stored, the code will convert them into a single character and print it to serial out. This is one major benefit of the Arduino, as it can directly print characters into a serial out buffer.

3.5.2.1 Potential Sources of Error

As mentioned in the previous section there are two main sources of error that were encountered during the development and testing of the core LaserCom software.

The first is a "Phase Shift" error that occurs when the laser and receiver boards run synchronously for long enough that the minor execution time lag desynchronizes the system and corrupts the data. As stated before, the fix to this issue was to include an initialization pulse that tells the receiver code to exit its idle state and record a single byte of data. After every eight loops, the code returns to the idle state to resynchronize after a new initialization pulse is received. This successfully reduces the amount of time that the two boards must run in perfect synchronization, and it allows the receiver board to reset its phase after every byte. Unfortunately, this system adds a new delay function which slows down the data transfer, but we believe this minor sacrifice is worth it.

Additionally, error comes from the delay function itself. If the board does not have the capabilities to accurately produce a time delay in the small timespan we need, transmission will be impossible. From Arduino documentation and testing, the `delay()` and `delaymicroseconds()` commands were only able to generate accurate code stoppages in the millisecond range, thus 10 kHz was the upper limit of data transfer. Other methods such as assembly programming in the Arduino or using interrupts were not explored, but it is suspected that the

accuracy would still be an issue. For example, if we require a 1 microsecond delay and the accuracy is within 0.5 microseconds, this would significantly alter the synchronization of the laser and receiver loops – thus corrupting data. In order to reach speeds above megahertz, you must have time delays with an accuracy of ~1 – 10 % time error.

3.5.3 Satellite and Ground Station Integration

In order to properly utilize the LaserCom system, it must be seamlessly integrated into the larger TracSat assembly. This means that we need to have a "plug and play" system in which the laser/receiver/PCB assembly can be connected into either the ground station computer or the CubeSat Raspberry Pi. There should be minimal hassle when transmitting commands from the ground station through the LaserCom system or sending those commands into the CubeSat for processing.

This integration system was made possible through the serial data transfer framework that has existed for several decades. The Arduino is heavily integrated within this system, as they have a built-in serial monitor and several functions that easily print or receive input through the serial USB connection. Through the Python serial library, we are able to read all incoming serial data through the specified port, and we are able to transmit data into the serial pipeline to be analyzed by the LaserCom system.

Because of these requirements, we created a python class that, once imported into any Python code, will provide several key LaserCom functions to the user. This class is called `lasercom.py` and it is crucial in integrating the LaserCom system into both the satellite and the Ground Station. This code is attached in Appendix A, but its key functionalities will be explained on the following page.[1]

[1] Note that all functions in the class require a "Port" string. This is the name of the serial port the specific board being used is connected to.

These are the three key functions that comprise the LaserCom class:

`receiveCommand(port)`: This function uses the receiver port, and like the name suggests it will output a singular string. This will be best used on the CubeSat to receive initial commands such as coordinates or other setup data at the start of the mission. It also has a built in error checking method that ensures the received command is correct (as explained in Sec. 3.5.1)

`receiveData(port)`: This function provides live data from the serial input. This will most likely be used on the ground station to print and analyze live telemetry.

`sendData(port, data)`: This function takes in a "data" string as a parameter and sends it to the laser board which then transmits the string. This function is useful for both sending initial commands to the satellite or sending back live telemetry to the ground station.

3.5.3.1 Satellite

The primary functions of the satellite are controlled using Python code in a Raspberry Pi, thus it is important to integrate the LaserCom system into this already written software. Through the LaserCom class, we can simply import and use all the functions with no hassle. All that is required is the serial port names, and the functions can be implemented directly into the CubeSat code.

3.5.3.2 Ground Station and GUI

The ground station will act as the main hub for the entire LaserCom system. From this endpoint, the user will be able to send commands and messages to the CubeSat, track all telemetry and data, and see system performance metrics. It is the control center for the entire operation, therefore a graphical user interface (GUI) must be created to make the process of controlling and analyzing the mission as carefree as possible. The GUI can be seen in Fig. 3.20.

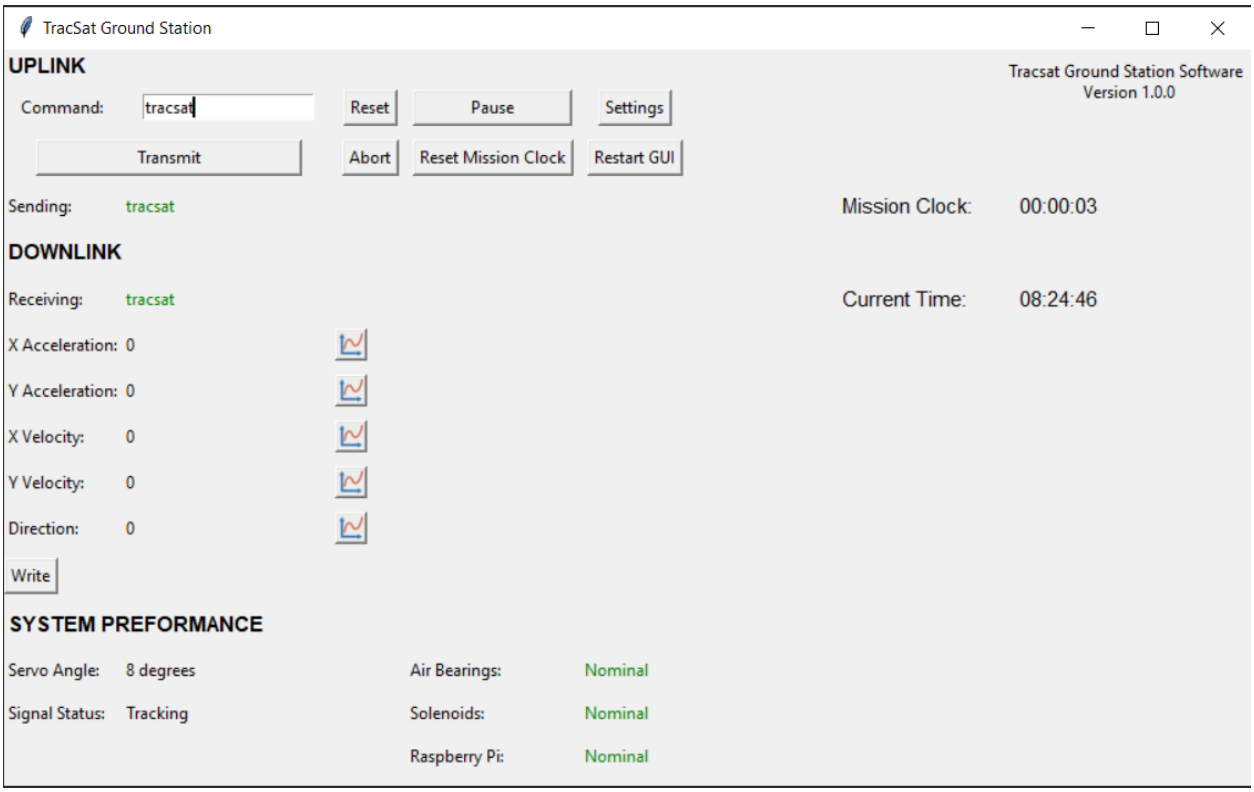

Figure 3.20. Ground Station GUI Interface

Similar to the satellite, the LaserCom class allows the ground station code to seamlessly interact with the LaserCom system through the serial connection. Due to the implementation of this class, the Ground Station GUI program does not have to scrape or import data from separate parallel programs, instead it can communicate with the LaserCom controllers on its own.

As stated previously, the ground station GUI is split into three main sections. The first component is “Uplink”. In this section the user controls all transmissions from the ground station to the satellite. They will be able to send commands, pause the laser, abort the mission, or reset the program from this section. Uplink will most likely be used at the beginning of the mission, as it is important to send initial conditions, coordinates or other starting data to the CubeSat before it can begin its autonomous tasks.

The downlink area is responsible for displaying all telemetry, data, and video sent from the CubeSat. There will be several telemetry generating devices onboard (such as an accelerometer, gyroscope, camera, etc.), and all of these will be streaming live data back to the GUI. Here the user will be able to plot the data it is receiving and see running averages of the values. Each data type has a live plotting option, and this can be seen in Fig. 3.21. Additionally, there is an option to

write the data to a .csv file in order to analyze the telemetry at a later date. The backend for this system will mainly be housed in the lasercom.py class, and code will sort the stream of data it receives (currently does not exist due to lack of full system tests).

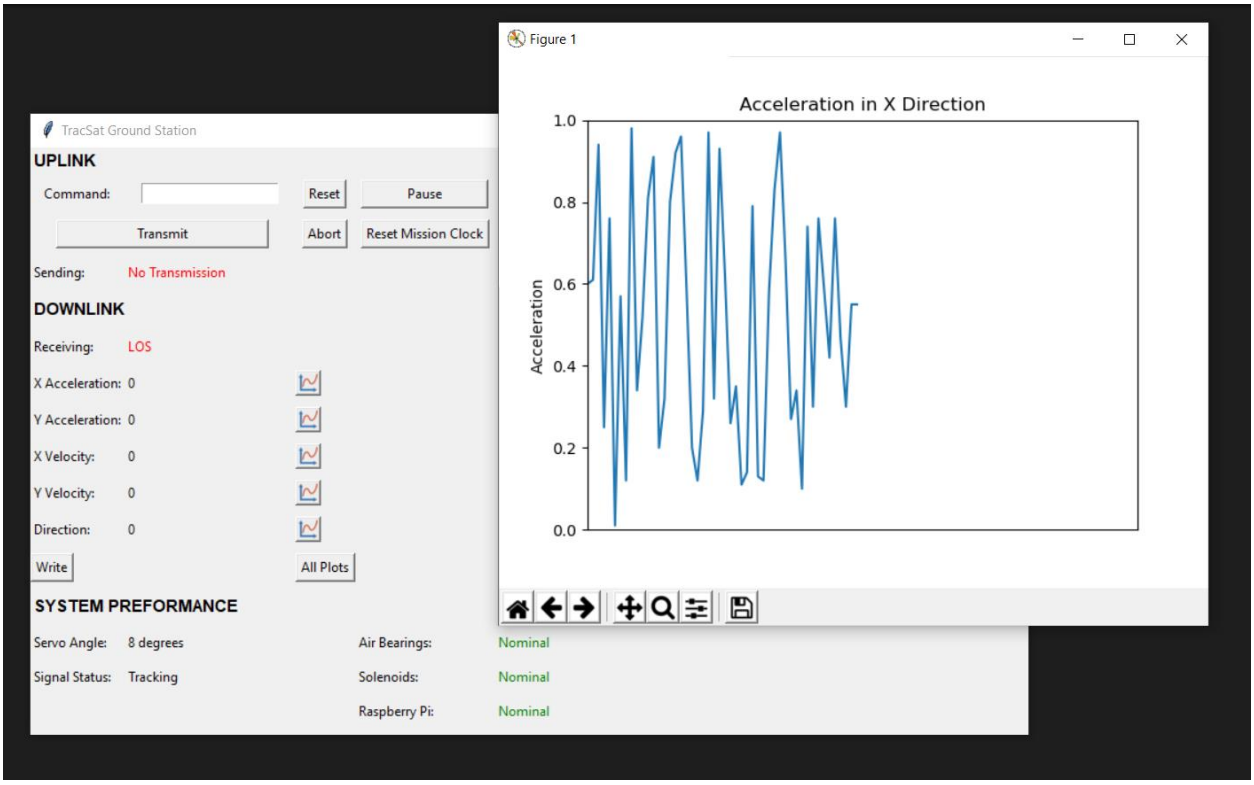

Figure 3.21. GUI Live Plotting Window

Lastly, the system performance section displays the status of key mission components from all subteams in one area. Here the user will be able to see the status of the tracking algorithm (tracking or acquisitions), the tracking servo angle, what solenoids are deployed, and other vital performance metrics. The status for the CubeSat systems will also be included in the telemetry and sorted in the LaserCom class, while other non-CubeSat values such as tracking servo angle can be directly fed into the ground station from the LTS controller.

It is also crucial to develop the GUI and Ground Station software to be completely independent of the Arduino system, as the future of TracSat is destined to leave these boards behind. The only code that would need to be significantly altered when swapping out the Arduinos would be the LaserCom class, and the ground station GUI would remain untouched.

The ground station will act as the relay of data and commands from the CubeSat to the GUI, which displays the data for the user. The design is a flat plate to which the laser turret (see Sec.

4.2.3) and the LaserCom assembly are fixed. The flat plate is mounted to an adjustable camera tripod which will be stationed some distance away from the table. The closer the tripod is to the table the more challenging it will be for the control system to maintain laser lock, so a reasonable distance will be found when system integration and testing comes along. Shown below in Fig. 3.22, the final Ground Station design is shown with both LTS and LLS systems combined.

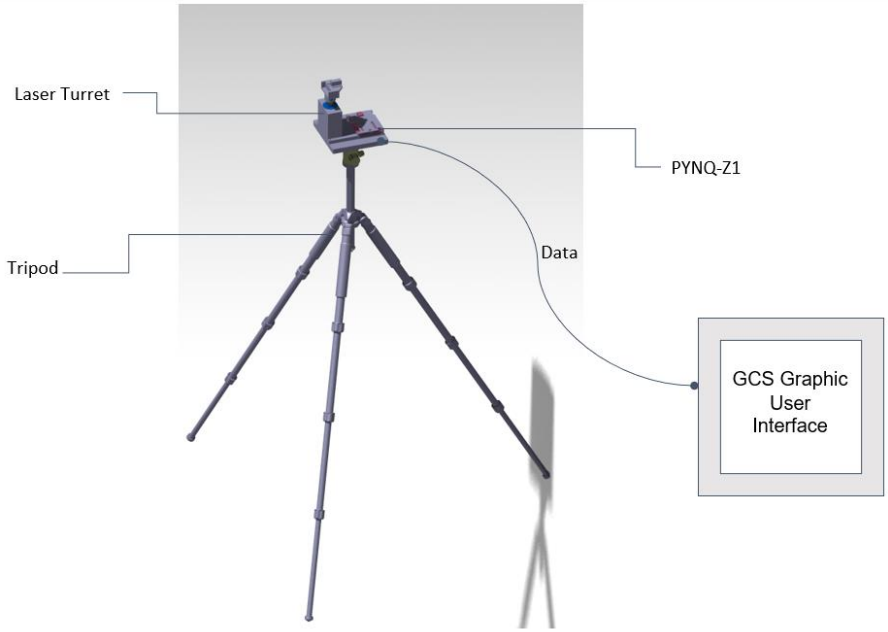

Figure 3.22. Ground Station Design

3.6 CONCLUSIONS

The Laser Link System has undergone significant progress during the 2019-2020 academic year. The physics and fundamental concepts behind laser communications were researched and a design down-selection of specific lasers and electronic systems was completed. After these components were acquired, a proof-of-concept Arduino system was implemented which successfully transmitted data and commands in a number of tests, culminating in December 2019 with a successful test where a command was sent from the ground station to the CubeSat via laser which initiated CubeSat translation. In the Spring 2020 semester, a fully-fledged GUI was developed to allow for direct user manipulation and monitoring of the Laser Link System. Additionally, the leap was made from the Arduino prototype to a PYNQ board with the theoretical capabilities to achieve the necessary data transfer rate to meet the full mission success criteria. This board was thoroughly researched, and while a final system was not completed utilizing the PYNQ, future teams can build off of the software and logical architecture and framework put in place during this academic year.

4 LASER TRACKING SYSTEM (LTS)

4.1 LTS INTRODUCTION AND OBJECTIVE

The purpose of the Laser Tracking System (LTS) is to maintain laser communication between the CubeSat and Ground Station while the CubeSat is moving. This feature is important for the full CubeSat system because consistent laser communications requires that the transmitting laser diode remain aligned with the laser receiver at all times. The LTS has the following components:

- **Laser Diode:** A laser diode serves as the transmitter for the communications systems. It uses TTL logic to serve as a communication device.
- **Laser Receivers:** An array of five laser receivers serve two purposes; first, they receive data from the opposing laser diode. Second, they help the LTS determine which way to move to maintain laser lock.
- **Laser Turret:** The laser turret holds both the receivers and the laser diode. The turret will be able to rotate to allow for the LTS to maintain laser lock.
- **Servo:** A continuous rotation servo will rotate the laser turret to allow it to maintain lock.
- **Turret Base:** The turret base holds the servo.
- **Control Software:** The software is used to interpret data from the laser receivers to activate the servo in a certain speed and direction to allow for the LTS to maintain laser lock.

To fulfill its purpose, the LTS most fulfill several requirements. The main requirement is that the LTS must be able to establish and maintain laser contact between the ground station and the CubeSat before, during, and after it moves. In addition, if laser contact is lost, the LTS must be able to recognize that it is lost and re-establish contact. To fulfill the main requirements, the LTS must have a motor capable of rotating the turret at an acceptable speed, and the LTS must use the microcontroller selected by the team.

There are two main criteria with which the LTS is judged: 1) At a minimum, the LTS should maintain laser link in all directions and 2) be able to re-acquire signal if it is lost. Ideally, the link should be re-established in five seconds or less.

4.2 COMPONENTS

Shown below in Fig. 4.1. is a full CAD rendering of the LTS Array system. Within Sec. 4.2 here, several components within the full design will be looked at in more specific detail.

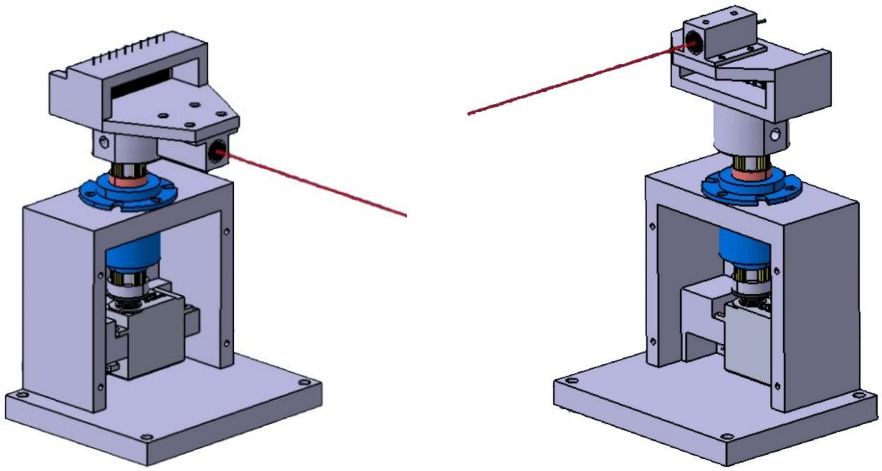

Figure 4.1. Full LTS Array CAD Model

4.2.1 Receiver Array and Circuit

The receiver array consists of five FDS1010 photodiodes from ThorLabs. Photodiodes are circuit elements that vary in resistance depending on how much light is shown at them. Each of these receivers have 100 square millimeters of area. They respond to light with a wavelength of 350-1100 nanometers; the laser diode used in this design emits light at 650 nanometers. The receivers are mounted as shown below in Fig. 4.2.

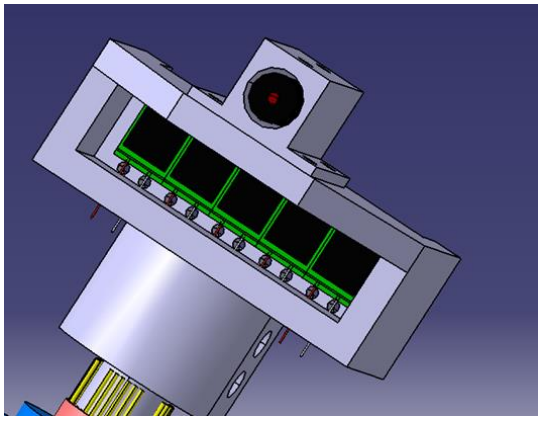

Figure 4.2. Receiver Array Close-Up

To read signals from one of the photodiodes, one must use the following circuit seen in Fig. 4.3,

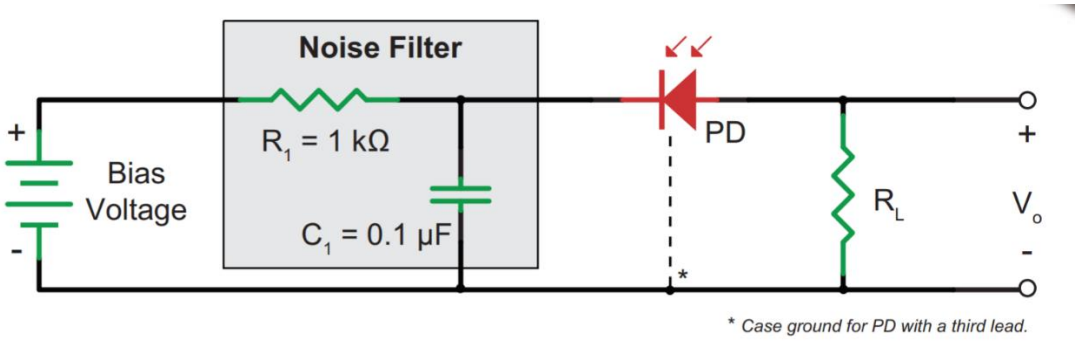

Figure 4.3. Single Photodiode Circuit Diagram

The bias voltage is 5 V. The photodiode is shown by the PD symbol. The load resistance needed for our circuit was 1500 Ω. The positive end of V_O is connected to a GPIO pin on a Raspberry Pi or other microcontroller. The circuit for all five photodiodes is shown below in Fig. 4.4,

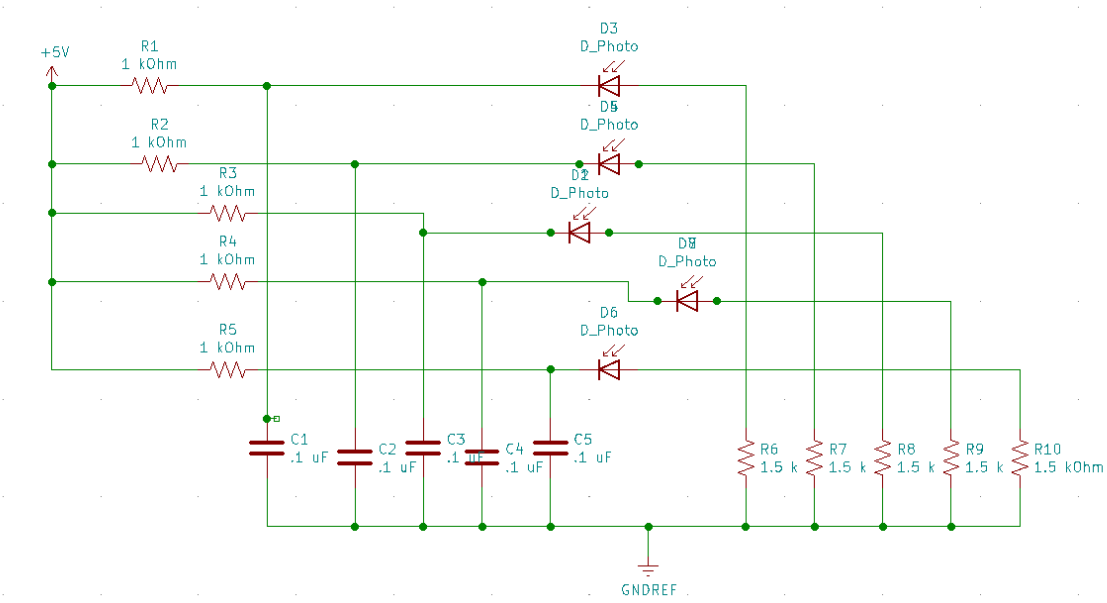

Figure 4.4. Full Photodiode Circuit Diagram

4.2.2 Laser Diode

The laser diode intended for use on the LTS is the TTL Laser Diode from Adafruit Industries. The laser diode is capable of Transistor-Transistor Logic, which allows for binary signals to be transmitted by laser to a receiver. It has a maximum frequency of 50 kHz. It also uses 5 V of electrical power. One of these diodes is mounted on a metal mount on the Turret Head. This diode is shown below in Fig. 4.5.

Figure 4.5. Adafruit TTL Laser

4.2.3 Laser Turret

All of the main structures of the LTS were made from 3-D printed materials. This allows for designs to be rapidly iterated and modified as the LTS shape changes. In addition, since there were few structural loads involved in the LTS, a strong structure was not needed.

The turret head is designed to house the laser diode and receiver array in a compact fashion. The head is mounted on a servo to allow it to rotate three-hundred and sixty degrees. It features holes for both mounting the laser diode mount and threading the receiver wires through. There are two versions of the turret head; one with the laser on top, and the other with the receivers on top. Both versions have the laser and receivers at an equal distance apart, so that they can communicate back and forth when they are level with one another. The turret head is mounted on a 3-D printed axle, which is threaded through a slip ring to allow for wires to pass from the fixed base to the rotating head without getting tangled.

The base of the turret is designed to mount the servo and the slip ring. It features holes for mounting those two components, as well as holes allowing it to be mounted to the CubeSat or the Testbed. It also has open on one side to make installing components and wiring easier.

In order for the head to be able to rotate freely without risk of twisting the wires, a slip ring was used to connect the laser and sensors to the main computer. The model selected has a wide through-hole for the axle, which allows for a much sturdier end product.

4.2.4 Servo

The servo we decided to use is the AR-3606HB Continuous Rotation Servo. Continuous Rotation (CR) servos differ from standard servos because, instead of being commanded to rotate to a certain angle, the CR servo rotates a certain speed and direction. This allows for more precise speed control as compared to a normal DC motor. The CR Servo is controlled using Pulse-Width Modulation (PWM). Using PWM parameters, the servo is controlled by inputting square wave signals with varying pulse widths to control the speed of the rotation. The servo has the following qualities shown in Table 4.1.,

Table 4.1. Servo Physical Specifications

Quality	Value
Operating Voltage	4.8 V
Stall Torque[1]	6 kg·cm
No Load Speed	62.5 rpm
Dimensions (cm)	4.05 × 2 × 3.8

[1]Stall Torque and Load Speed at 4.8 V

The pulse widths that can be inputted into the servo range from 800-2200 microseconds. The relationship between pulse-width and orientation is as follows:

Table 4.2. Pulse-Width and Rotational Orientation Relation

Pulse Width (μs)	Orientation
800 – 1500	Clockwise
1500	No Rotation
1500 – 2200	Counterclockwise

4.3 CONTROL SYSTEM

The control software component of the LTS system uses inputs from the laser receivers and translates it into PWM commands for the servo to rotate the laser turret. Due to the lack of sensors on the CubeSat, the only way to determine if an individual laser receiver is going to lose laser contact is when the laser diode doesn't point towards that receiver anymore. Thus, five receivers are used to determine if laser contact is going to be lost. By determining which receiver is receiving data, and which different receiver was last receiving data from the laser diode, the direction that the laser turret needs to rotate can be determined. The operation of the control system can be tracked using Fig. 4.6 below.

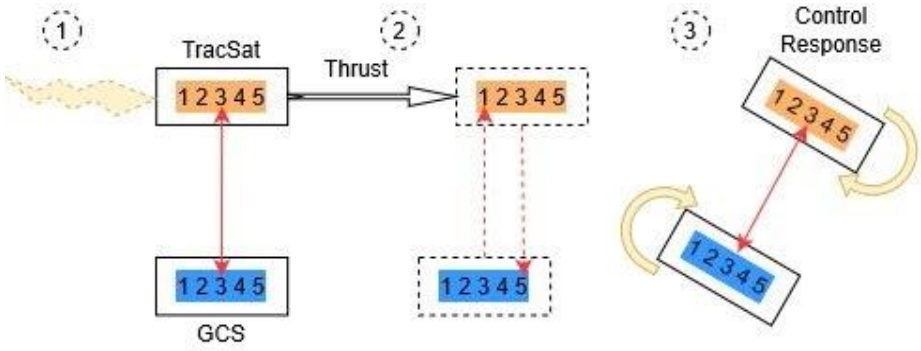

Figure 4.6. LTS Tracking Operation

In Phase One of operation, both the Ground Station (GCS) and TracSat turrets rotate to obtain lock. Once both turrets are receiving laser communication through Receiver 3, the TracSat begins to move. As it moves, the receivers that are in contact with the laser beam change. This is shown in Phase Two as the CubeSat is reading communication on Receiver 1 while the GCS is reading communication on Receiver 5. Since we know that we are getting data from these receivers, the control systems in the Ground Station and TracSat command both turrets to rotate clockwise to have laser signals go through the center receivers again (Receiver 3). The decision tree for the TracSat LTS control system is shown on the following page in Fig. 4.7,

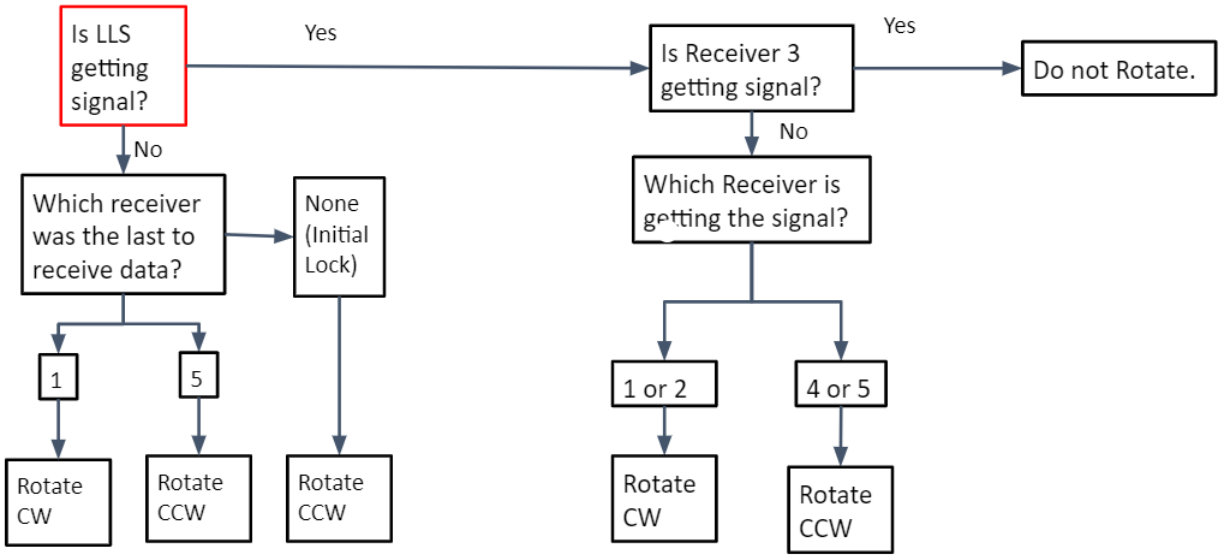

Figure 4.7. LTS Control System Diagram

The Ground Station LTS control system operates with the same logic, but with the directions reversed in every case. In addition, the logic can be scaled up or down with any odd number of detectors.

5 HARDWARE AND PROPULSION

5.1 INTRODUCTION AND OBJECTIVES

The primary objective of the hardware and propulsion systems is to demonstrate the usage of a levitation base to accurately simulate a CubeSat in a space environment. Secondarily, this would allow laser communications between the CubeSat and a ground station. The three main components of the hardware and propulsion system are the chassis, the propulsion system, and the levitation base. The chassis serves to contain all physical components of the CubeSat in place while adhering to the form factor and organizing the array of electronics and connections. The purpose of the propulsion system is to accurately and precisely maneuver the CubeSat in a stable manner. The levitation base is used to "levitate" the entire assembly above a very flat granite table, allowing near-frictionless two-dimensional motion as it glides across the table.

All components must work together in harmony to provide a stable and precise platform on which laser communications technology could be used. This played a key role in design considerations, as the hardware required for such equipment had massive implications for the motion of the CubeSat, and vice versa. It was quickly realized that in order to provide a consistently-functioning product, many iterations of the design would be needed. Numerous lessons were learned and they were all carefully considered to produce a refined system that is a significant improvement over the preliminary designs.

5.2 CUBESAT CHASSIS

The CubeSat chassis primarily needed to be easy to adjust and configure as the components in the CubeSat were changed due new designs and tests that needed to be completed. There have been many iterations, but the team finally settled on a 4U configuration, organized in a square, as can be seen on the following page in Fig 5.1.

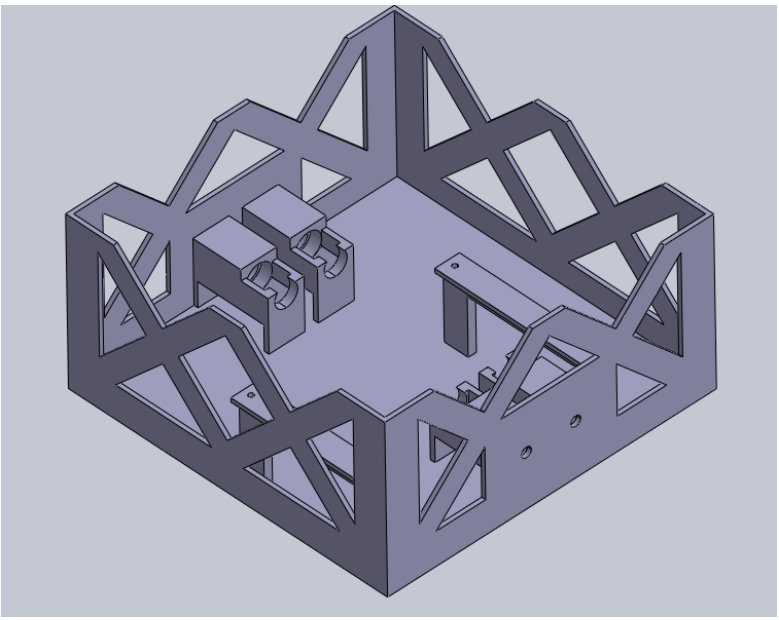

Figure 5.1. CubeSat Model with Nozzle Mounts

The two shelves located inside the CubeSat are there to provide a higher location where electronics can be mounted, out of the way of the propulsion system. The two mounts on either side are designed to hold the four adjustable air nozzles. The many iterations of the model have been 3D printed because of the low cost and speed of manufacturing, which allowed the team to create and test many different versions quickly. This particular size was selected in order to have enough space to house the many systems which will be installed inside it. This includes the propulsion system with nozzles, manifolds, solenoids, and tubing, as well as all of the electronics and the laser system. The final model pictured above was never tested with all systems installed, but enough previous testing was completed with most of the parts needed to ensure this orientation would be large enough.

5.3 PROPULSION SYSTEM

The propulsion system is used to move the CubeSat and base translationally. There were considerations earlier in the design process to include rotational motion through a reaction wheel and adjustability of the nozzles to allow for fine tuning of the system, but due to constraints for this team, the objective was reduced to purely translational motion. These capabilities will be included in future iterations, but for now, since translational motion was the only requirement, the

propulsion system was able to be substantially simplified from initial plans. The pneumatic system in place is illustrated in the Fig 5.2 below.

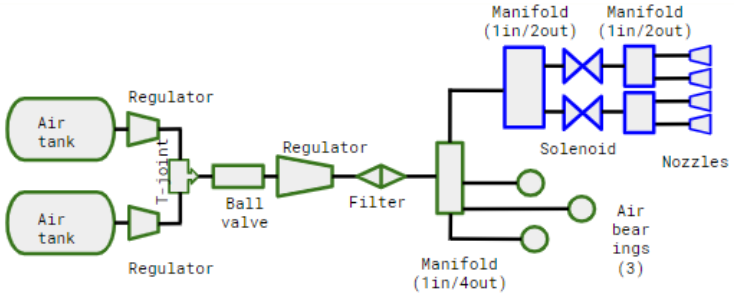

Figure 5.2. Propulsion System Diagram

The propulsion system in question consists of the blue parts shown above, and is located within the CubeSat. The green parts are located in the levitation base and include the air tanks which provide the pressurized air for the system, a ball valve to turn the system on and off, a regulator to reduce the pressure, and a filter. Then, a 4-way manifold sends the air to three air bearings, which levitate the base, and to the propulsion system. The air is split to two streams by a manifold and each is sent through a solenoid that can be activated to open the flow to either set of nozzles. Another manifold is located on both sides to split the flow to two nozzles each. Fig. 5.3 below demonstrates the location of this system within the CubeSat chassis, with important systems identified by black labels and the flow direction indicated with yellow arrows.

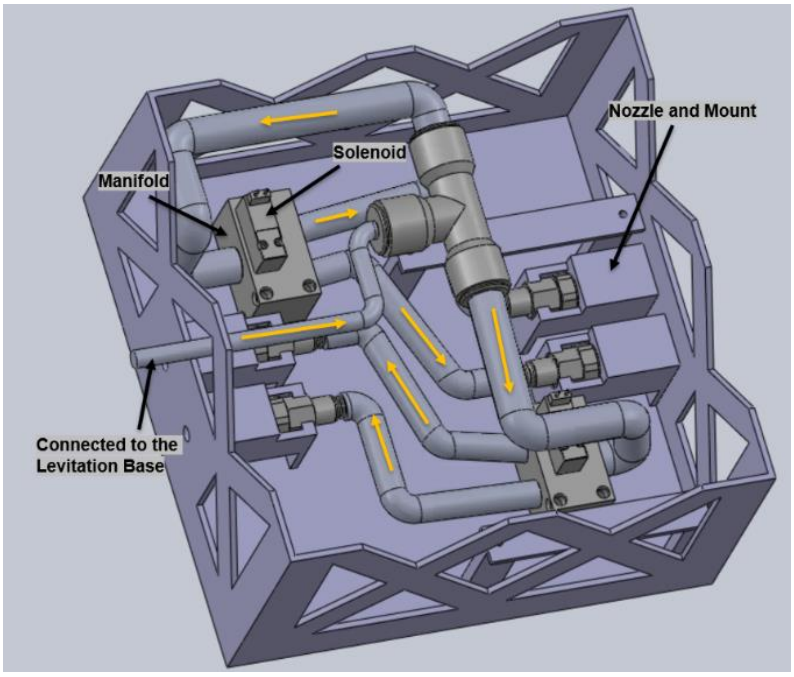

Figure 5.3. CubeSat with Propulsion System

As can be seen, there are two nozzles on one side and two on the other. This provides 2D translational motion, as required. This nozzle configuration was tested in an earlier iteration of the CubeSat model, with promising results. Due to the COVID-19 and the team's inability to work on the system on campus after spring break, this updated design iteration has not been constructed or tested. However, due to the fact that all of the parts selected were tested and operated as expected, all information the team was able to acquire seems to indicate this design will function well.

5.4 LEVITATION BASE

The main purpose of the levitation base is to hold the CubeSat in a steady hover above the granite table's surface. In addition, the base provides infrastructure for the propulsion system, as well as stability and drift mitigation technology that is capable of producing steady translation for the CubeSat. Certain considerations such as height (for laser safety), surface cover (to avoid bumping into the walls of the table), and aesthetics were taken into account. Previous iterations were based on the same basic model: a base plate that has a structure on the bottom to hold the air bearings in place, and on top, a space for the propulsion hardware which is mounted horizontally (the two tanks laying on their sides next to each other). Above the propulsion hardware, a top plate is mounted to hold the CubeSat, and this top plate is attached to the base plate. Small variations included cut outs in the base plate to lower the tanks and allow for a shorter base, and making a thicker and larger "disk" shape for the base plate for strength (as opposed to the original spoked version which proved to be prone to breaking).

The current iteration keeps many of the same features. Beginning at the top of the base, a flat top plate provides stability for the CubeSat chassis. The top plate is then supported by two supports on either side, that each span the length of the plate. These supports also double as holsters for the two air tanks and the propulsion assembly. All of the propulsion components are therefore held rigidly in place, preventing them from shifting during maneuvers. Below the supports is the base plate, which is responsible for holding the entire assembly together, as well as two other main purposes. The first is drift mitigation. By adding inch-deep pockets, the base can hold up to approximately 10 lb of steel. This added weight significantly increases the inertia of the CubeSat, which effectively prevents the CubeSat from drifting unnecessarily. In addition to a central pocket, there are four "edge pockets" that allow for smaller additions of weight to account for unsymmetrical inertia distributions. The weight in each of these pockets can be chosen

independently to account for any specific case. On the very bottom of the base plate are three "cups" with hemispheres in the center designed to hold in place the air bearings. There is a slit in each cup to accommodate the tubing. This helps mitigate drift because it was found that depending on how the air bearings are rotated, the CubeSat drifts in different directions. The cups ensure that a consistent angle is kept, and therefore the drift due to the air bearings can be addressed consistently and easily.

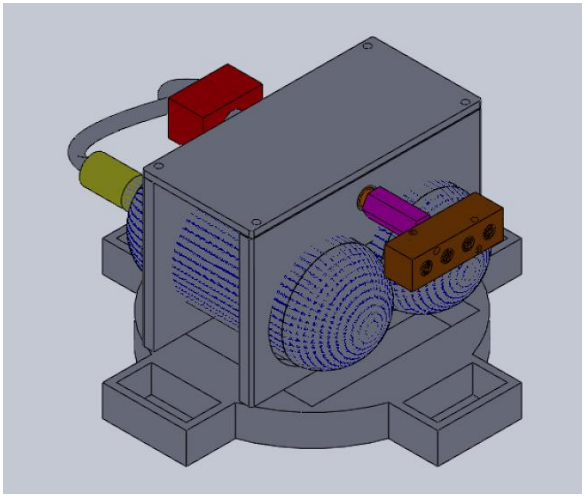

Figure 5.4. Levitation Base CAD (Front View)

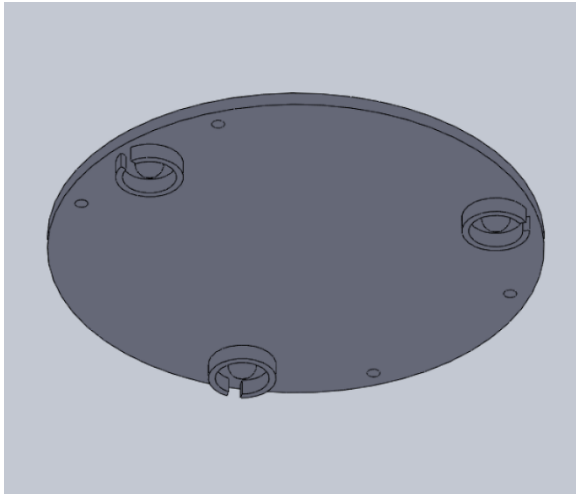

Figure 5.5. Levitation Base CAD (Bottom View)

The levitation base is 3D printed in order to reduce production costs and maximize manufacturing speed, which was especially important if a component broke. The measurements and dimensions were also nearly exact due to the fact that a computer printed the parts. In addition, this choice allowed for simple modification of the design; all that needed to be done to change the design was edit the model.

6 ELECTRONICS

6.1 INTRODUCTION AND OBJECTIVES

The purpose of the electronics system is to provide adequate power to the on board components and allow for solenoid actuation. The main driver of the electronic system is a custom designed printed circuit board (PCB). The PCB is designed to act as the interface between the battery, microcontroller, and control surfaces. The PCB needs to have flexibility in its ability to supply power to the microcontroller if a different selection must be made. The battery must have enough capacity to allow the system to operate for a sufficient amount of time to complete maneuvers.

6.2 SOLENOID SELECTION

The solenoid selection process took multiple factors into account while making this decision. They needed to easily fit the formfactor of the satellite previously described by the hardware team. The solenoids also needed to have minimal impact on the overall power budget of the system. Only loose guidelines were applied to this selection process. The design criteria for the solenoids were that they must be able to open and close in under 0.5 seconds and they must fit within the CubeSat form factor. The solenoid valve selected was the Pneumadyne S15MML-20-12-2D. This solenoid is a magnetically latching variety which provides benefits in form factor and power consumption. This valve is able to actuate between open and close in 10 to 12 milliseconds, which is well under the design constraints that have been laid out. The power consumption in order to trigger an open or close maneuver is 4 W.

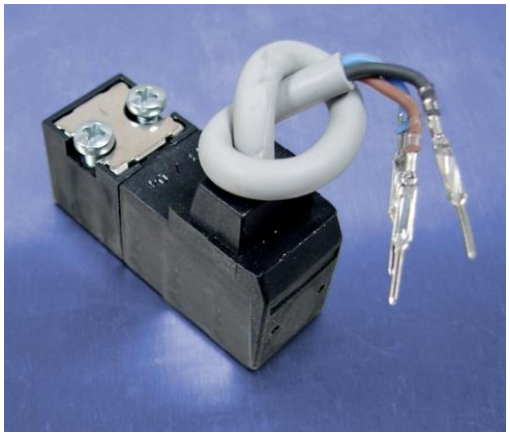

Figure 6.1. Pneumadyne Solenoid

6.3 POWER BUDGET

With all main components of the system identified, a power budget needed to be established. The four major components taken into account when identifying the power needs of our system is the microcontroller, the solenoids, and the servo motor. Two different use cases were identified when evaluating our power budget. The first use case was intended to simulate a normal use scenario under 40 minutes of operation. The second use case considered is a worst case scenario. The worst case scenario assumes that all onboard systems are drawing as much power as possible during 40 minutes of operation. Without in-person testing due to COVID-19, these values are purely estimates. Since the PYNQ has been difficult to work with, the microcontroller specs below are based off of estimates from Raspberry Pi and Arduino boards.

Table 6.1. Nominal Power Use Case

Item	Amps (A)	Voltage (V)	Power (W)	Duty Cycle	Avg. Power (W)	Energy for 40 mins. (J)	Power (mAh)
Microcontroller	0.80	12	9.6	1.0	9.60	23040	533.33
Solenoid (x4)	0.33	12	4.0	0.22	0.088	35.2	0.815
Servo Motor	0.25	5	1.3	1.0	1.25	500	27.78
						Total	561.93

Table 6.2. Worst Power Use Case

Item	Amps (A)	Voltage (V)	Power (W)	Duty Cycle	Avg. Power (W)	Energy for 40 mins. (J)	Power (mAh)
Microcontroller	2.50	5	12.5	1.0	12.5	30000	1666.67
Solenoid (x4)	0.33	12	4.0	0.22	0.88	1056	24.44
Servo Motor	0.8	5	1.3	1.0	4	1600	88.89
						Total	1691.11

6.4 BATTERY SELECTION

After the preliminary power budget was created, a battery that fit our needs was selected. The battery selected for our project is an 11.1 V LiPo drone battery. The capacity of the battery is 3200 mAh. This battery capacity is almost double what the estimated worst case power use scenario would require, therefore it provides flexibility in the event the estimates are inaccurate. Due to the nature of LiPo batteries, the battery is unable to provide a constant voltage. The battery outputs anywhere from 12.6 V when it is fully charged, to 10.5 V when it is completely drained. Given that our capacity is much larger than even the worst use case calls for, the output voltage should remain high enough to actuate the solenoids as needed.

6.5 PCB DESIGN

The PCB is designed to act as the interface between the microcontroller and the solenoids. The PCB receives power at 12 V from an on board battery source and distributes it to the solenoids as needed. It is also intended to supply power to the microcontroller. The PCB is also designed to take input signals of 3.3 V from the microcontroller that signal an open or close of each solenoid. The servo motor is to be powered using one of the 5 V pinouts on the microcontroller.

Since the solenoids that were selected have a common ground with two positive current wires, a custom circuit needed to be designed to allow the microcontroller to actuate the solenoids. To allow a microcontroller to interact with the solenoids, a two transistor setup was designed. An N-Channel MOSFET and a P-Channel MOSFET were used. When a GPIO pin on the microcontroller is turned on, it sends 3.3 V to the N-Channel MOSFET gate pin. This allows the gate to be opened and current flows from the drain pin to the source pin. Now that the first MOSFET has been opened, this creates a voltage difference between the gate and source of the P-Channel MOSFET. These work differently from N-Channel MOSFETS in that the source is connected to a positive voltage source and is opened by measuring the difference in voltage between the source and gate. The gate must have a negative voltage applied to it to open. Once that negative voltage is applied the gate it opened and current is allowed to flow to the solenoid.

The P-Channel MOSFET selected was the FQP27P06. This MOSFET is opened when a source to gate voltage difference of at least -4 V is detected. In this circuit, the difference is -12 V. The N-Channel MOSFET selected was the 2N7000. This MOSFET is opened when a positive voltage

of at least 3 V is applied to the gate. In this circuit, 3.3 V is supplied to the gate from the microcontroller.

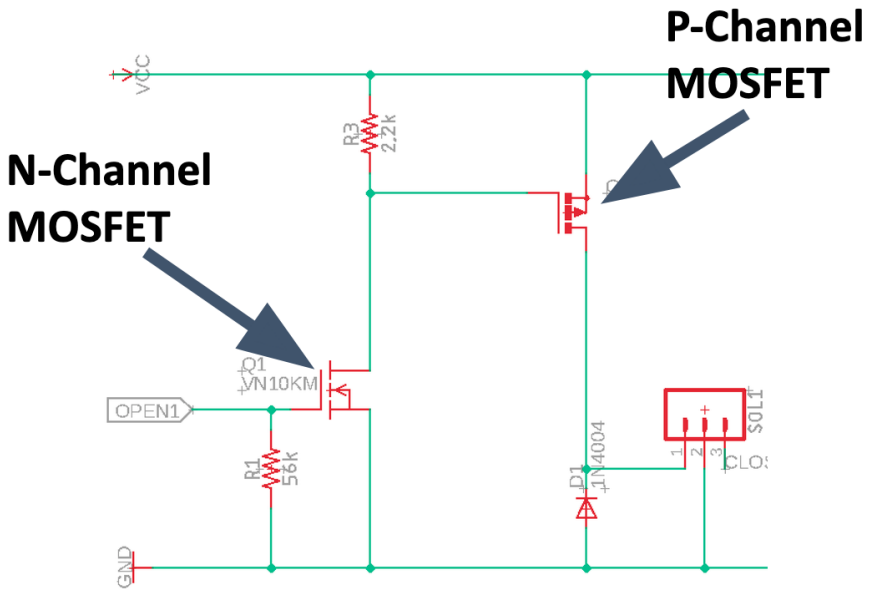

Figure 6.2. Sample Circuit for Open or Close Solenoid Maneuver

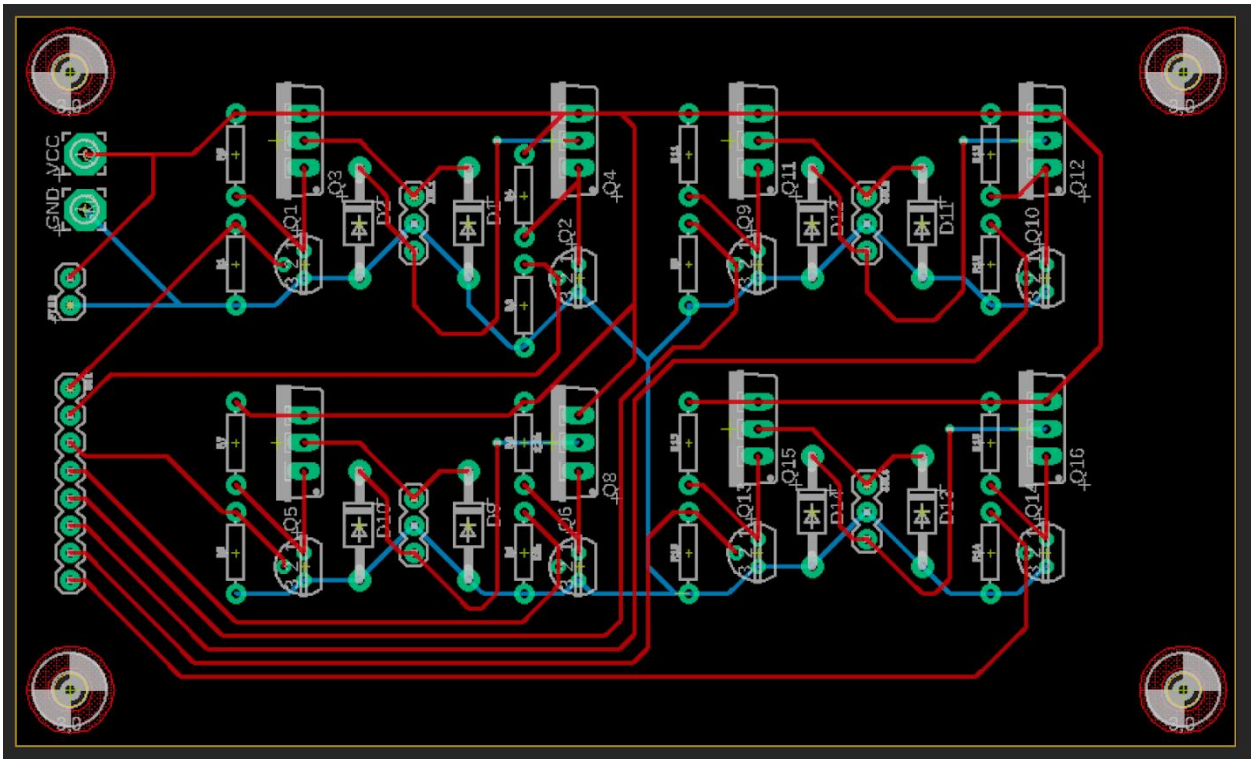

Figure 6.3. Board View Layout of PCB

7 TESTING

7.1 LASER LINK SYSTEM

7.1.1 LLS Testing Architecture

The initial test of the Laser Link System involved using a controller which would send data packets using laser communication to a receiver. The first proof of concept was tested using an Arduino based system. Speeds of 1 kHz were achieved with this system. The initial test also ended up being the baseline for future iterations and improvements to the system. The diagram for the laser communication system is shown below (as previously shown in Sec. 3.2) and this would be carried on for future iterations.

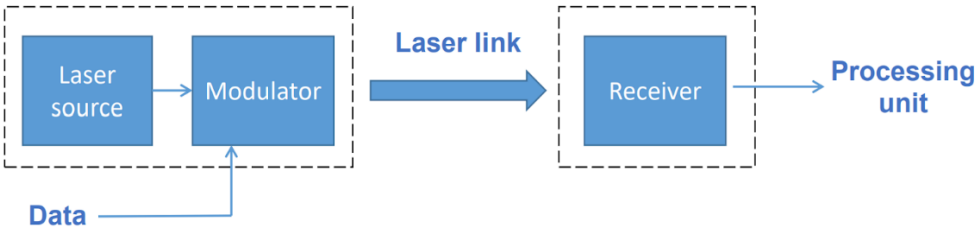

Figure 3.2. Diagram of Laser Communication System

After the individual pulses have been detected by the receiver, which for the LLS system is a photodiode, the flashes are then converted back into readable data using a processing unit. For the LLS system, respective algorithms have been designed that achieve this binary conversion, reconversion, and control over the laser pulses.

7.1.2 LLS Testing Milestone/Goals

The future iterations as mentioned in Sec. 7.1.1 were to fulfill the system requirement of downlinking video, more specifically images at a certain refresh rate. In order to do that, further technological developments needed to occur such as a maturation of data processing algorithms and the acquisition of more robust electronics to handle the large bandwidth required of video downlink. Recall from Sec. 3.3 Table 3.1, which explains how each iteration of testing led to more complexities.

Table 3.1. Progression of Telemetry Capabilities for LLS

Level of Complexity	Telemetry Contents	Data Rate Required	Additional Challenges
1	Basic test information (e.g. words, simple bit string)	< 1 Kbps	N/A
2	Commands to CubeSat (e.g. translate, stop, send specific data)	~ 1 Kbps	Requires developed Ground Station user interface
3	CubeSat state information (e.g. translation speed, rotation rate, power levels)	~100 Kbps	CubeSat must be integrated with primary avionics sending information to LLS
4	CubeSat Downlink video at varying levels of quality	> 1 Mbps	Requires little-to-no packet loss and very high data rates

While the additional challenges may not be associated directly with the LLS system's overall infrastructure, the technical details with respect to evolving the system became more and more complex. To execute more complex commands and send more detailed information a more sophisticated ground station was required. This ground station was developed during the Spring 2020 semester but was not able to be fully tested. The next level of telemetry is downlinking CubeSat state information. This level of telemetry was not achieved due to both an inability to achieve the 100 kbps rate with the Arduino, the issues with the PYNQ, and an avionics system unable to collect and organize the required information. Finally, after these three levels of telemetry had been achieved, video transmission at various levels would have been attempted.

7.2 LASER TRACKING SYSTEM

7.2.1 LTS Initial Tests

Initially, the Laser Tracking System team performed a series of tests to determine the ability of the individual components of the LTS system to work properly. The two components that needed testing were the laser receivers and the servos. The servo test was to determine if the servo worked as advertised, and was able to be controlled using the testbed. The laser receivers would be tested to see if they worked properly.

The servo was tested by connecting it to the testbed shield, and testing if it could operate from commands on the Raspberry Pi. The testbed was able to control the servo properly. However, it was learned that the pulse-width required to make the servo stop was 10 microseconds less than listed on the servo datasheet.

The laser receivers were tested by connecting to them the circuits describing in the LTS system section. They were then tested by monitoring their responses to having a laser aimed at them. It was shown during the tests that all of the receivers were able to be read using the GPIO pins on the Raspberry Pi. One limitation shown by the test was that the circuit for the laser receivers was very delicate. In the future, it would be ideal to either solder together a permanent circuit using a perfboard or create a PCB circuit.

7.2.2 LTS Final Tests

The objective of the LTS final test is to ensure the full LTS system is capable of maintaining laser contact as the Testbed moves. The completion of this test will allow for future versions of the laser tracking system to be used on the CubeSat and Ground Station.

Originally, the final test was to consist of two LTS systems; one for the testbed and one for the fixed Ground Station. The test procedure is shown below. In the first part of the test, the Testbed (black) and Ground Station (blue) LTS systems rotate to achieve laser lock. After the two systems have laser communications, the second part of the test begins. The testbed moves forward, and the LTS systems rotate to track each other and maintain laser communication. After the testbed moves forward 1 meter, it stops. The test steps are shown below, with the leftmost picture being step 1 and rightmost being the final state of the test.

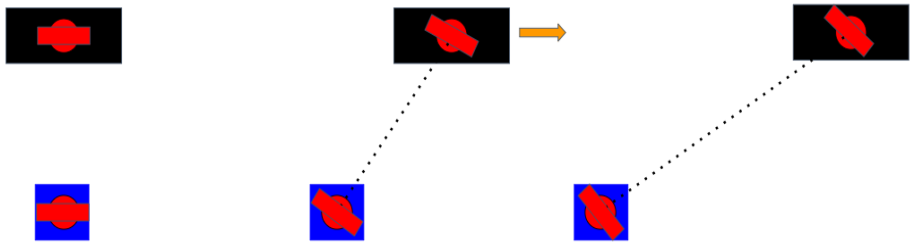

Figure 7.1. LTS System Final Test Diagram

In the updated version of the test, the Ground Station LTS is replaced with a laser diode held by a team-member. The testbed LTS will still rotate to achieve laser lock, and will move forward while the laser diode rotates to follow the testbed's progress. During the test, the LTS was able to achieve and maintain laser lock as the testbed moved. However, the laser signal was often received at very oblique angles; in the future, the hood on the laser turret should be longer to prevent this.

7.3 HARDWARE AND PROPULSION

7.3.1 Translation

The system performed well during translational testing. After being fed a command to fire the cold gas thrusters for 5 seconds, the system moved straight, without visibly deviating from its original path, at approximately 3 cm/s without significant rotation. The “stop” command (firing the thrusters on the opposite side) was not executed, so the system was not able to stop itself.

7.3.2 Drift Mitigation

Drift had proved to be a significant concern that had only recently been considered. As testing began, it was observed that even before any planned translation, the CubeSat and base would drift in random directions at velocities up to 1 cm/s. It would also rotate at speeds up to 5 deg/s (approximately). The major concern with this is that the LLS would not be able to have any initial contact because of the randomness of the movement, and the test could not begin. Thus, the solutions included adding weight to the levitation base to increase inertia, evening out the weight distribution in the base and on the CubeSat, and fixing the air bearings in place. Because of complications due to COVID-19, actual tests were not able to be performed. However, it was noted that in the preliminary diagnosis of the drift problem that adding weight, no matter where, positively impacted stability. Fine-tuning this weight distribution would have been critical in effectively stopping the drift. In just the preliminary diagnosis testing, adding weight in some cases allowed the CubeSat to remain completely stationary for up to 10 seconds, when without weight, the CubeSat would start drifting within 3 seconds.

7.3.3 Release Mechanism

Upon initialization of the testing one issue was having a consistent release mechanism for the CubeSat. When the levitation base is started and the low-friction environment is initialized, any forces applied to the CubeSat will impart unintended torques and translations. Highly of note, and mentioned previously, the effect is present just from the tester aligning the CubeSat before starting the test. A release mechanism was constructed to hold the CubeSat in place for the duration between the levitation base being started up and when the gas thrusters would begin. The mechanism was a swinging arm that comes down over the CubeSat and holds it firmly in place on a single point to eliminate any uneven forces. This mechanism only saw minimal testing due to

COVID-19, however the team is confident that this type of mechanism would greatly improve initial stability and should be implemented in future iterations.

7.4 ELECTRONICS

Due to the limitations created by COVID-19, only a partial test of the electronics system was done. In order to test the effectiveness of the circuit design, the PCB was assembled and the solenoids were attached to their respective locations. The battery was connected via an XT-60 connector. Due to a limitation on the number of available microcontrollers, a power supply was used to simulate a 3.3 V signal. After completion of the test, solenoids 1, 2, and 4 actuated as expected. Solenoid 3 did not actuate as expected. Using an altimeter, it was found that current was constantly flowing through the “open” side of the circuit. It is believed this is due to either a soldering error or faulty components.

8 CONCLUSION

8.1 SUMMARY OF PROJECT STATUS

As of May 2020, TracSat has made significant progress in a number of key areas. The levitation base and propulsion system has undergone significant improvements that allow for longer operation times and improved stability to minimize drift. The laser communications system, while not currently operational, has had much progress in terms of research and understanding of the capabilities necessary to achieve the relevant objectives, such as transmitting data at a rate of at least 50kHz, and being able to support 240p video at 10 fps. The discovery and usage of the PYNQ controller will eventually allow for 50-100 MHz data rate transmission which could enable the transmission of full quality colored video at 15 fps. Additionally, a fully functional GUI (Graphical User Interface) has been developed for the ground station to be actively controlled and monitored by a user. The Laser Tracking System is capable of maintaining laser contact while the testbed is moving, and can regain contact in a few seconds if contact is lost. The fundamental architecture of the TracSat is well defined, and future teams should build upon this by focusing on the development of specific technical subsystems.

8.2 TAKE-AWAYS

Throughout the Fall 2019 - Spring 2020 academic year, a number of lessons were learned ranging from understanding of the difficulties in embedded systems to the value of good documentation and systems engineering. One of the largest take-aways was that solid, well-defined, and realistic objectives need to be established and discussed as the first step. Due to ambitious initial goals, the systems and requirements needed to be re-evaluated between the Fall 2019 and Spring 2020 semesters. This could have been prevented by selecting more feasible objectives at the beginning, and led to many of the systems not being as advanced as they could have been. An example of this is the Laser Link System. While the prototyping of the Arduino laser link system was a successful and necessary proof-of-concept, the PYNQ board was too complicated to budget less than 10 weeks to learn how to use it for a laser link system. Additionally, extensive time and effort was spent developing systems that were not useful for the redesigned mission objectives, such as the reaction wheel, computer vision, and docking architectures. Another take-away from this year has been the difficulty of system integration and testing. Sub-team progress was held up in many cases

by the time and effort required to conduct integrated tests due to unforeseen complexities and complications. These issues could have been prevented by better systems engineering processes including facilitating constant communication between sub-teams whenever making significant design decisions. Learning how to conduct trade studies and their value was another valuable take-away. The mission objectives and requirements allowed for a variety of potential systems and designs to be utilized, and so down-selecting to the best system or design required consideration of all the possible metrics and impacts both on the relevant subsystem as well as on other subsystems.

8.3 IMPROVEMENT AND SUGGESTIONS

8.3.1 Ground Station Improvements

The system was tested on a prior laser turret design. The tripod was in fact very difficult to align with a turret. The adjustments take time to finely tune especially as the distance increases between the two turrets. Furthermore, the laser beam is small so it is difficult to target it that way as well. A suggestion for the design might be to develop a rack and pinion system controlled by a servo which may be commanded to turn a certain degree laterally and vertically. One potential inspiration might be the system that moves a laser cutter into position.

8.3.2 Laser Tracking System Improvements

Several improvements should be made to the LTS to improve functionality. Firstly, the laser receivers should be put deeper into the “hood” of the LTS turret. This would only allow lasers to hit the receivers from directly in front of the hood, and would prevent laser signals from coming in at oblique angles. Another improvement to be made to the LTS system would be to decrease the speed of the servo while the laser is hitting an off-nominal receiver. This would prevent the rotation of the servo causing an overshoot, where the laser goes from hitting an off-nominal receiver to a nominal receiver and then again going off-nominal.

8.4 FUTURE DEVELOPMENT

There are several future development paths for the TracSat System. From the perspective of the Propulsion and Levitation systems, the TracSat system can be modified to operate on the low-friction table better by further mitigating translational and rotational drift. For example, the TracSat base could be redesigned to change the center of mass, or systems could be added to counteract the drift effects. From a Guidance, Navigation, and Control perspective, the CubeSat could be equipped with sensors to allow it to figure out where it is on the table and where it needs to go. The CubeSat could also be equipped with a rotation wheel to stabilize and control its rotation to allow for complex maneuvers to be made. For the Laser Link System, future efforts should be focused on development of the PYNQ architecture. The PYNQ board enables high data rates which can achieve the full success criteria, but requires extensive software engineering efforts. From a Laser Tracking System perspective, the LTS could be improved by using sensors to improve the LTS algorithm. Using sensors, the LTS algorithm could be modified to anticipate how the LTS turret would need to rotate to maintain nominal laser lock, instead of rotating after the laser lock is lost. Overall, there is a well-defined path forward for future TracSat teams.

APPENDIX A – LLS SOFTWARE

Laser Code:
Written in C++ with Arduino framework

```
#define LASERPIN 12

  const byte numChars = 132;
  char receivedChars[numChars];
  boolean newData = false;
  boolean inputRetreived = false;
  int bits[500];
  int byteIndex;
  int size;
  void(* resetFunc) (void) = 0;

void setup() {
  // Setup code
  pinMode(LASERPIN, OUTPUT);
  Serial.begin(9600);

  //Wait for serial input.
  Serial.println("Waiting for data...");
  while (inputRetreived == false){
    recvWithStartEndMarkers();
    showNewData();
  }
  String serialInput = receivedChars;

  convertToBinary(serialInput);
  Serial.println("Starting...");
}


void loop() {

  //If '0' is inputted in the serial monitor, then the program
     will reset for a different set of data to be inputted.
  if (Serial.available() > 0 && Serial.read() == 48) {
    resetFunc();
  }

  //Start flash
  digitalWrite(LASERPIN, HIGH);
  delay(1);
  digitalWrite(LASERPIN,LOW);
```

```
  //Send one byte
  for(int i = byteIndex; i < 8 + byteIndex; i++){
    digitalWrite(LASERPIN, bits[i]);
    //Serial.print(bits[i]);
    delayMicroseconds(1000);
  }
  byteIndex+=8;
  //Serial.print(" ");

  //Reset the byteIndex back to 0 when the serialInput has been
     fully sent
  if (byteIndex >= size){
    //Serial.println("");
    byteIndex = 0;
  }

  //Delay between bytes.
  digitalWrite(LASERPIN, LOW);
  delay(5);
}
//Function to receive serial input
void recvWithStartEndMarkers() {
    static boolean recvInProgress = false;
    static byte ndx = 0;
    char startMarker = '<';
    char endMarker = '>';
    char rc;

 // if (Serial.available() > 0) {
    while (Serial.available() > 0 && newData == false) {
        rc = Serial.read();

        if (recvInProgress == true) {
            if (rc != endMarker) {
                receivedChars[ndx] = rc;
                ndx++;
                if (ndx >= numChars) {
                    ndx = numChars - 1;
                }
            }
            else {
                receivedChars[ndx] = '#';
                receivedChars[ndx+1] = '\0'; // terminate the
                                                 string
                recvInProgress = false;
                ndx = 0;
                newData = true;
```

```
            }
        }

        else if (rc == startMarker) {
            recvInProgress = true;
        }
    }
}

//Show serial input
void showNewData() {
    if (newData == true) {
        Serial.print("Inputted Message: ");
        Serial.println(receivedChars);
        inputRetreived = true;
        newData = false;
    }
}

//Convert serial input to binary
void convertToBinary(String serialInput){
  byteIndex = 0;
  int position =  0;
  size = serialInput.length() * 8;
  int printIndex = 0;

  //Serial.println(serialInput.length());
  for(int i=0; i<serialInput.length(); i++){
    char myChar = serialInput.charAt(i);
    String binaryString = String((int) myChar, BIN);
    //Serial.println(binaryString);
    if (binaryString.length() < 8){
      for (int u = 0; u < 8 - binaryString.length(); u++){
        bits[position] = 0;
        position = position + 1;
      }
    }
    //Serial.println(bits[position]);
    for (int k = 0; k < binaryString.length(); k++){
      bits[position] = String(binaryString.charAt(k)).toInt();
      //Serial.println(bits[position]);
      position = position + 1;
    }
  }

  //Print all bytes for visual inspection
  for (int i=0; i < size; i++){
```

```
    Serial.print(bits[i]);
    if (!((i+1)%8) && i != 0){
      Serial.println(serialInput.charAt(printIndex++));
    }
  }
}
```

Receiver Code:
Written in C++ with Arduino framework

```
#define SOLARPIN A0
#define THRESHOLD 30

int ascii = 0;
int ambientReading;
int k = 1;

void setup() {

  // Setup Code.
  pinMode(SOLARPIN, INPUT);
  Serial.begin(9600);
  ambientReading = analogRead(SOLARPIN);
  Serial.println(ambientReading);//Set the ambient reading of
                                 the receiver.

}
void loop() {

  int reading = analogRead(SOLARPIN);
  int bits[8];
  //When a start flash is read, an 8 bit loop begins.
  if (reading > ambientReading + THRESHOLD) {
    delay(1);
    for (int i = 0; i < 8; i++){
      if (analogRead(SOLARPIN) > ambientReading + THRESHOLD){

        bits[i] = 1;
      } else{
        bits[i] = 0;
      }
     // Serial.println(analogRead(SOLARPIN));
    delayMicroseconds(1000);
    }
    convertToText(bits);
  }
}
```

```
//Convert array of bits to text. Outputted to serial monitor.
void convertToText(int bits[]){
    int i = 0;
    int j = 0;
    int len  = 8;
    double sum = 0;

    for(i=(len-1);i>=0;i--)
    {
        sum = sum +  (pow(2,i) * (bits[j]));
        j++;
    }

    char aChar = (int) lround(sum);
    if (aChar == '#'){
      Serial.println("");
    } else {
      Serial.print(aChar);
    }
    /*
    for (int l = 0; l < 8; l++){
      Serial.print(bits[l]);
    }
    Serial.println("");*/
 }
```

GUI Code:
Written in Python

```
import tkinter as tk
from lasercom import lasercom
import time
import os
import sys
from serial import Serial
from datetime import datetime
import pytz
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from itertools import count
import pandas as pd
import random


if len(sys.argv) < 3:
```

```
    print("Error: Groundstation software requires both uplink
and downlink ports as CLI arguments \nEx. python groundstation
COM3 COM4")
    sys.exit()


coms = lasercom()
uplinkPort = sys.argv[1]
downlinkPort = sys.argv[2]
coms.resetArduino(uplinkPort)
ser = Serial(downlinkPort, 9600)
root = tk.Tk()
root.title("TracSat Ground Station")
root.geometry("900x450")
#root.resizable(width=False, height=False)
root.minsize(width=900, height=530)
pauseCommand = "empty"
seconds_MC = 0
minutes_MC = 0
hours_MC = 0
stopMissionClock = 1
plot = tk.PhotoImage(file = r"icon_small.png")

#################
######UPLINK#####
#################



uplinkLabel = tk.Label(root, text='UPLINK', font='Helvetica 11
bold')
command = tk.Label(root, text="Command:")
currentlyTransmitting = tk.Label(root, text="Sending:")
commandEntry = tk.Entry(root)
currentTransmission = tk.Label(root, text="No Transmission",
fg="red")
missionClock = tk.Label(root,text="Mission Clock:\t00:00:00",
font='Helvetica 11')
missionClock.grid(row=3,column=5,pady=5, padx=10)

def refresh_missionClock():
    global seconds_MC
    global minutes_MC
    global hours_MC
    global stopMissionClock
    if not stopMissionClock:
        if seconds_MC < 59:
```

```
            seconds_MC = seconds_MC + 1
        else:
            seconds_MC = 0
            if minutes_MC < 59:
                minutes_MC = minutes_MC + 1
            else:
                hours_MC = hours_MC + 1

    missionClock.config(text='Mission Clock:\t%02d:%02d:%02d' %
(hours_MC, minutes_MC,seconds_MC))
    missionClock.after(1000, refresh_missionClock)


def transmitFunction():
    command = commandEntry.get()
    global stopMissionClock
    if command is not "":
        global stopMissionClock
        stopMissionClock = 0
        print("Inputted Command:" + command)
        coms.sendData(uplinkPort, command)
        currentTransmission.config(fg="green")
        currentTransmission.config(text= command)
        stopMissionClock = 0

def resetMissionClock():
    global seconds_MC
    global minutes_MC
    global hours_MC
    global stopMissionClock
    seconds_MC = 0
    minutes_MC = 0
    hours_MC = 0
    stopMissionClock = 0

def resetFunction():
    global seconds_MC
    global minutes_MC
    global hours_MC
    global stopMissionClock
    coms.resetArduino(uplinkPort)
    currentTransmission.config(fg="red")
    currentTransmission.config(text= "No Transmission")
    commandEntry.delete(0, 'end')
    stopMissionClock = 1
    seconds_MC = 0
    minutes_MC = 0
```

```
    hours_MC = 0

def pauseFunction():
    global stopMissionClock
    if pause["text"] == "Pause":
        global pauseCommand
        pauseCommand = commandEntry.get()
        coms.resetArduino(uplinkPort)
        pause.config(text='Unpause')
        print("Paused")
        currentTransmission.config(fg="red")
        currentTransmission.config(text= "No Transmission")
        stopMissionClock = 1

    else:
        print('Unpaused')
        coms.sendData(uplinkPort, pauseCommand)
        currentTransmission.config(fg="green")
        currentTransmission.config(text= pauseCommand)
        pause.config(text='Pause')
        stopMissionClock = 0

def restart_program():
    python = sys.executable
    os.execl(python, python, * sys.argv)

def abort():
    global stopMissionClock
    coms.sendData(uplinkPort, "abort")
    currentTransmission.config(fg="red")
    currentTransmission.config(text= "Abort Code")
    stopMissionClock = 1

def settings():
    window = tk.Toplevel(root)

    ## ADD SETTINGS HERE


refresh_missionClock()


transmit = tk.Button(root, text ="Transmit", width = 26,
command=transmitFunction)
stop = tk.Button(root, text ="Stop")
reset = tk.Button(root, text ="Reset", command=resetFunction)
abort = tk.Button(root, text ="Abort", command=abort)
```

```
pause = tk.Button(root, text ="Pause", width = 15,
command=pauseFunction)
resetMissionClock = tk.Button(root, text ="Reset Mission Clock",
command=resetMissionClock)
extrabutton1 = tk.Button(root, text ="  Settings  ",
command=settings)
extrambutton2 = tk.Button(root, text ="Restart GUI",
command=restart_program)
groundstationSoftware = tk.Label(root, text="Tracsat Ground
Station Software \n Version 1.0.0") #\n Copyright 2020. All
Rights Reserved. \n Tracsat is a subsidiary of GA
Electromagnetic Systems.")

uplinkLabel.grid(row=0,column=0, sticky="W")
command.grid(row=1,column =0, pady=5)
currentlyTransmitting.grid(row=3,column =0, pady=5, sticky="W")
currentTransmission.grid(row=3,column =1, pady=5, sticky="W")
commandEntry.grid(row=1,column =1, pady=5, padx=15)
transmit.grid(row=2,columnspan = 2, pady=5)
#stop.grid(row=1,column = 1,  pady=5)
reset.grid(row=1,column = 2, pady=5, padx=5)
abort.grid(row=2,column = 2, pady=5, padx=5)
pause.grid(row=1,column = 3, pady=5, padx=5)
resetMissionClock.grid(row=2,column = 3, pady=5, padx=5)
extrabutton1.grid(row=1,column = 4, pady=5, padx=5)
extrambutton2.grid(row=2,column = 4, pady=5, padx=5)
groundstationSoftware.grid(row = 0, rowspan=2,column = 5,
sticky="NE", pady=5,)


root.grid_columnconfigure(5, weight=1)




#################
#####DOWNLINK####
#################

def accel_x_plot_func():
    x_vals = []
    y_vals = []
    index = count()
    # Initialize
    x_axis_start = 0
    x_axis_end = 10
    #Animate plot
```

```
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
        y1 = data['total_1']
        plt.cla()
        plt.axis([len(x)-50, len(x)+50, 0, 1]) #update axis
        plt.plot(x, y1)
        plt.xticks([])
        plt.title("Acceleration in X Direction")
        plt.ylabel("Acceleration")
    ani = FuncAnimation(plt.gcf(), animate, interval=100)
    plt.show()

def accel_y_plot_func():
    plt.close(1)
    x_vals = []
    y_vals = []
    index = count()
    # Initialize
    x_axis_start = 0
    x_axis_end = 10
    #Animate plot
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
        y1 = data['total_1']
        plt.cla()
        plt.axis([len(x)-50, len(x)+50, 0, 1]) #update axis
        plt.plot(x, y1)
        plt.xticks([])
        plt.title("Acceleration in Y Direction")
        plt.ylabel("Acceleration")
    ani = FuncAnimation(plt.gcf(), animate, interval=100)
    plt.show()

def vel_x_plot_func():
    plt.close(1)
    x_vals = []
    y_vals = []
    index = count()
    # Initialize
    x_axis_start = 0
    x_axis_end = 10
    #Animate plot
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
```

```
        y1 = data['total_1']
        plt.cla()
        plt.axis([len(x)-50, len(x)+50, 0, 1]) #update axis
        plt.plot(x, y1)
        plt.xticks([])
        plt.title("Velocity in X Direction")
        plt.ylabel("Velocity")
    ani = FuncAnimation(plt.gcf(), animate, interval=100)
    plt.show()

def vel_y_plot_func():
    plt.close(1)
    x_vals = []
    y_vals = []
    index = count()
    # Initialize
    x_axis_start = 0
    x_axis_end = 10
    #Animate plot
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
        y1 = data['total_1']
        plt.cla()
        plt.axis([len(x)-50, len(x)+50, 0, 1]) #update axis
        plt.plot(x, y1)
        plt.xticks([])
        plt.title("Velocity in Y Direction")
        plt.ylabel("Velocity")
    ani = FuncAnimation(plt.gcf(), animate, interval=100)
    plt.show()

def direction_plot_func():
    plt.close(1)
    x_vals = []
    y_vals = []
    index = count()
    # Initialize
    x_axis_start = 0
    x_axis_end = 10
    #Animate plot
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
        y1 = data['total_1']
        plt.cla()
        plt.axis([len(x)-50, len(x)+50, 0, 1]) #update axis
```

```
        plt.plot(x, y1)
        plt.xticks([])
        plt.title("Direction")
        plt.ylabel("Angle")
    ani = FuncAnimation(plt.gcf(), animate, interval=100)
    plt.show()

def all_plot_func():
    plt.close(1)
    # Create a figure with two subplots
    fig = plt.figure()
    ax1 = fig.add_subplot(5,1,1)
    ax2 = fig.add_subplot(5,1,2)
    ax3 = fig.add_subplot(5,1,3)
    ax4 = fig.add_subplot(5,1,4)
    ax5 = fig.add_subplot(5,1,5)
    # Adjust spacing between plots
    plt.subplots_adjust(top = 0.93, bottom = 0.07, hspace = 0.6)
    #define the function for use in
matplotlib.animation.funcAnimation
    def animate(i):
        data = pd.read_csv('data.csv') #read data
        x = data['x_value']
        y1 = data['total_1']
        # Set subplot data
        xlim = len(y1)
        ax1.clear()
        ax1.plot(y1)
        ax1.set_xlim(xlim - 30, xlim)
        ax1.set_xticks([])
        ax2.clear()
        ax2.plot(y1)
        ax2.set_xlim(xlim - 30, xlim)
        ax2.set_xticks([])
        ax3.clear()
        ax3.plot(y1)
        ax3.set_xlim(xlim - 30, xlim)
        ax3.set_xticks([])
        ax4.clear()
        ax4.plot(y1)
        ax4.set_xlim(xlim - 30, xlim)
        ax4.set_xticks([])
        ax5.clear()
        ax5.plot(y1)
        ax5.set_xlim(xlim - 30, xlim)
        ax5.set_xticks([])
        # Set subplot titles
```

```
        ax1.set_title("X Acceleration")
        ax2.set_title("Y Acceleration")
        ax3.set_title("X Velocity")
        ax4.set_title("Y Velocty")
        ax5.set_title("Direction")
    ani = FuncAnimation(fig, animate, interval=100)
    plt.show()

downlinkLabel = tk.Label(root, text='DOWNLINK', font='Helvetica
11 bold')
currentReceivingLabel = tk.Label(root,text="Receiving:")
currentReceiving = tk.Label(root,text="LOS", fg="red")
currentTime = tk.Label(root,font='Helvetica 11')
GATime = tk.Label(root,font='Helvetica 11')
moscowTime = tk.Label(root,font='Helvetica 11')
accel_x_label = tk.Label(root,text="X Acceleration:")
accel_y_label  = tk.Label(root,text="Y Acceleration:")
direction_label  = tk.Label(root,text="Direction:")
vel_x_label  = tk.Label(root,text="X Velocity:")
vel_y_label  = tk.Label(root,text="Y Velocity:")
accel_x = tk.Label(root,text="0")
accel_y = tk.Label(root,text="0")
direction = tk.Label(root,text="0")
vel_x = tk.Label(root,text="0")
vel_y = tk.Label(root,text="0")
accel_x_plot = tk.Button(root, image = plot, width = 17, height
= 17, command=accel_x_plot_func)
accel_y_plot = tk.Button(root, image = plot,
command=accel_y_plot_func)
direction_plot = tk.Button(root, image = plot,
command=direction_plot_func)
vel_x_plot = tk.Button(root, image = plot,
command=vel_x_plot_func)
vel_y_plot = tk.Button(root, image = plot,
command=vel_y_plot_func)
write = tk.Button(root, text="Write")
allPlots = tk.Button(root, text="All Plots",
command=all_plot_func)


downlinkLabel.grid(row=4,column=0, columnspan=2, pady=5,
sticky="W")
currentReceivingLabel.grid(row=5,column=0,pady=5,sticky="w")
accel_x_label.grid(row=6,column=0,pady=5,sticky="w")
accel_y_label.grid(row=7,column=0,pady=5,sticky="w")
vel_x_label.grid(row=8,column=0,pady=5,sticky="w")
vel_y_label.grid(row=9,column=0,pady=5,sticky="w")
```

```
direction_label.grid(row=10,column=0,pady=5,sticky="w")
accel_x_plot.grid(row=6,column=2,pady=5,sticky="w")
accel_y_plot.grid(row=7,column=2,pady=5,sticky="w")
vel_x_plot.grid(row=8,column=2,pady=5,sticky="w")
vel_y_plot.grid(row=9,column=2,pady=5,sticky="w")
direction_plot.grid(row=10,column=2,pady=5,sticky="w")
write.grid(row=11,column=0,pady=5,sticky="w")
allPlots.grid(row=11,column=2,pady=5)

def refresh_serialInput():
    global ser
    bytesToRead = ser.inWaiting()
    data = ser.readline(bytesToRead).strip()
    print(data)
    if not data:
        data = "LOS"
        color = "red"
    else:
        color = "green"
    currentReceiving.configure(text=data, fg=color)
    currentReceiving.after(90, refresh_serialInput)

def refresh_currentTime():
    now = datetime.now()
    timeString = now.strftime("%H:%M:%S")
    currentTime.config(text="Current Time:\t" + timeString)
    currentTime.after(1000,refresh_currentTime)

def refresh_GATime():
    timezone = pytz.timezone("America/New_York")
    GATimezone = datetime.now(timezone)
    timeString = GATimezone.strftime("%H:%M:%S")
    GATime.config(text="General Atomics HQ Time:\t" +
timeString)
    GATime.after(1000,refresh_GATime)

def refresh_moscowTime():
    timezone = pytz.timezone("Europe/Moscow")
    moscowTimezone = datetime.now(timezone)
    timeString = moscowTimezone.strftime("%H:%M:%S")
    moscowTime.config(text="Moscow Time:\t" + timeString)
    moscowTime.after(1000,refresh_moscowTime)




currentReceiving.grid(row=5,column=1,pady=5,sticky="w")
```

```
currentTime.grid(row=5,column=5,columnspan=2,pady=5)
#GATime.grid(row=6,column=5,columnspan=2,pady=5)
#moscowTime.grid(row=7,column=5,columnspan=2,pady=5)

accel_x.grid(row=6,column=1,pady=5,sticky="w")
accel_y.grid(row=7,column=1,pady=5,sticky="w")
vel_x.grid(row=8,column=1,pady=5,sticky="w")
vel_y.grid(row=9,column=1,pady=5,sticky="w")
direction.grid(row=10,column=1,pady=5,sticky="w")


refresh_currentTime()
refresh_serialInput()
refresh_GATime()
refresh_moscowTime()

#currentReceiving.after(90, refresh_serialInput)




#################
######SYSTEM#####
#################

systemPref = tk.Label(root, text='SYSTEM PREFORMANCE',
font='Helvetica 11 bold')
servoAngleLabel = tk.Label(root,text="Servo Angle:")
trackingStatusLabel = tk.Label(root,text="Signal Status:")
airBearingsStatusLabel = tk.Label(root,text="Air Bearings:")
solenoidsStatusLabel = tk.Label(root,text="Solenoids:")
raspberryPiStatusLabel = tk.Label(root,text="Raspberry Pi:")
servoAngle = tk.Label(root,text="8 degrees")
trackingStatus = tk.Label(root,text="Tracking")
airBearingsStatus = tk.Label(root,text="Nominal", fg='green')
solenoidsStatus = tk.Label(root,text="Nominal", fg='green')
raspberryPiStatus = tk.Label(root,text="Nominal", fg='green')

systemPref.grid(row=12,column=0, columnspan=2, pady=5,
sticky="W")
servoAngleLabel.grid(row=13,column=0, columnspan=2, pady=5,
sticky="W")
trackingStatusLabel.grid(row=14,column=0, columnspan=2, pady=5,
sticky="W")
airBearingsStatusLabel.grid(row=13,column=3, columnspan=2,
pady=5, sticky="W")
```

```
solenoidsStatusLabel.grid(row=14,column=3, columnspan=2, pady=5,
sticky="W")
raspberryPiStatusLabel.grid(row=15,column=3, columnspan=2,
pady=5, sticky="W")
servoAngle.grid(row=13,column=1, columnspan=2, pady=5,
sticky="W")
trackingStatus.grid(row=14,column=1, columnspan=2, pady=5,
sticky="W")
airBearingsStatus.grid(row=13,column=4, columnspan=2, pady=5,
sticky="W")
solenoidsStatus.grid(row=14,column=4, columnspan=2, pady=5,
sticky="W")
raspberryPiStatus.grid(row=15,column=4, columnspan=2, pady=5,
sticky="W")

tk.mainloop()
```

LaserCom Class
Written in Python

```
from serial import Serial
import time

class lasercom():

    #The uplink function is used to read an array of data
received by the lasercom system
    def receiveCommand(self, port):
        ser = Serial(port, 9600)
        dataArray = []
        i = -1
        correct = 0
        while (correct <= 3):
            if (ser.inWaiting()>0):
                dataArray.append(ser.readline())
                i = i + 1
            if i >= 2 and dataArray[i] == dataArray[i-1]:
                inputVectorString = dataArray[i]
                correct = correct + 1

        inputVectorString = inputVectorString[:-3]
        inputVectorString = inputVectorString.decode('utf-8')
        inputVector = inputVectorString.split(",")
        print("Received data:")
        print(inputVector)
        #inputVector = list(map(int,inputVector))
        return inputVector
```

```
    def receiveData(self,port):
        ser = Serial(port, 9600)
        #ser.flushInput()
        time.sleep(1)
        while True:
            bytesToRead = ser.inWaiting()
            print(ser.readline(bytesToRead).strip())
            time.sleep(.1)

    # the Downlink function sends an array of data into the
lasercom system
    def sendData(self, port, data):
        ser = Serial(port, 9600)
        #ser.flushInput()
        data = "<" + data + ">"
        print(ser.readline())
        ser.write(data.encode())
        print(ser.readline())
        ser.close()

    # Resets the lasercom system and turns off the laser.
    def resetArduino(self, port):
        ser = Serial(port, 9600)
        #ser.flushInput()
        ser.write(b'0')

class ReadLine:
    def __init__(self, s):
        self.buf = bytearray()
        self.s = s

    def readline(self):
        i = self.buf.find(b"\n")
        if i >= 0:
            r = self.buf[:i+1]
            self.buf = self.buf[i+1:]
            return r
        while True:
            i = max(1, min(2048, self.s.in_waiting))
            data = self.s.read(i)
            i = data.find(b"\n")
            if i >= 0:
                r = self.buf + data[:i+1]
                self.buf[0:] = data[i+1:]
                return r
            else:
                self.buf.extend(data)
```

APPENDIX B – LTS SOFTWARE

LTS Control Software and Test Code
Integrates Testbed movement in the script with the control code. Written in Python.

```
#Location 0 is off of the laser array
import time
from easygopigo3 import EasyGoPiGo3
import RPi.GPIO as GPIO
#Defining objects

gpg =EasyGoPiGo3()

servo = gpg.init_servo("SERVO2")
servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
keepWorking = True
GPIO.setmode(GPIO.BOARD)
#setup gpio pins
#Sensor Numbers are Connected to the Numbered Pins as follows
sensor1 = 37 #need to be replaced
sensor2= 36
sensor3 = 35
sensor4 = 33
sensor5 = 31
sensorlist = [sensor1,sensor2,sensor3,sensor4,sensor5]
GPIO.setup(sensorlist,GPIO.IN)

time.sleep(20)
CLoc = 0
PLoc = 0
starting = True
while starting:
    c = GPIO.input(sensor3)
    servo.gpg.set_servo(servo.gpg.SERVO_2,int(1400))
    if (c==True):
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
        starting = False
        CLoc = 3
        PLoc = 3

time.sleep(3)
startTime = time.time()

while keepWorking:
    a = GPIO.input(sensor1)
    b = GPIO.input(sensor2)
    c = GPIO.input(sensor3)
```

```
    d = GPIO.input(sensor4)
    e = GPIO.input(sensor5)
    tCurr = time.time()
    gpg.forward()
    tElapsed = tCurr - startTime

    if (tElapsed >=10):
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
        gpg.stop()
        time.sleep(.5)
        break

    if (CLoc != 0):
       PLoc = CLoc
    if (a==True):
        CLoc= 1
    elif (b==True):
        CLoc = 2
    elif (c==True):
        CLoc = 3
    elif (d==True):
        CLoc = 4
    elif (e==True):
        CLoc = 5
    else:
        CLoc = 0

    if CLoc == 3: #if Laser is hitting center detector, don't
move
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
        time.sleep(2)
        print("Laser on 3, no adjustment needed!")
    elif CLoc == 1 or CLoc == 2: #if Laser is hitting left
detectors
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1450))
        print("Laser on detector ",CLoc)

    elif CLoc == 4 or CLoc == 5: #if Laser is hitting right
detectors
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1550))
        print("Laser on detector ",CLoc)

    elif CLoc == 0: #If laser is not hitting any detectors
        if PLoc == 1 or PLoc == 2 or PLoc == 3: #If last
detector with laser communications was 1,2,3
            servo.gpg.set_servo(servo.gpg.SERVO_2,int(1450))
            print("Laser off. It last was on detector ",PLoc)
```

```
        elif PLoc == 5 or PLoc ==4: #If laser exited the right
side of the laser array
            servo.gpg.set_servo(servo.gpg.SERVO_2,int(1550))
            print("Laser off. It last was on detector",PLoc)

        elif PLoc == 0: #Initial Lock Code
            servo.gpg.set_servo(servo.gpg.SERVO_2,int(1700))
            print("Initial Lock")
            print("Moving CounterClockwise")
        else:
            print("PLoc Error")
            keepWorking= False
            servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
    elif CLoc == 7:
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))
        keepWorking = False
    else:
        print("CLoc Error")
        keepWorking = False
        servo.gpg.set_servo(servo.gpg.SERVO_2,int(1490))

print("Code Done")
```