

Robinson Observatory Restoration

Phase A: Scale Model



University of
**Central
Florida**

University of Central Florida

Department of Electrical Engineering and Computer Science

EEL 4915L

Dr. Samuel Richie, Dr. Lei Wei
Senior Design II

Sponsor(s): *Florida Space Grant Consortium*

Under the direction of Mike Conroy, FSI Project Manager

Group A

- Anthony Eubanks
- Brian Glass
- Melinda Ramos
- Thomas Vilan

Electrical Engineering
Electrical Engineering
Electrical Engineering
Electrical Engineering

Table of Contents

1.	Executive Summary	1
2.	Project Description	3
	2.1 Project Motivation and Goals	3
	2.2 Objectives	3
	2.3 Challenges	4
	2.4 Requirements Specifications.....	5
	2.5 House of Quality Analysis	6
	2.6 Block Diagrams	7
3.	Research Related to Project Definition.....	10
	3.1 Existing Similar Projects and Products.....	10
	3.2 Relevant Technologies.....	11
	3.3 Strategic Components and Part Selections.....	16
	3.3.1 Declination and Right Ascension Motors	17
	3.3.2 Power Supply Unit	25
	3.3.3 Sensors	27
	3.3.4 Joystick.....	33
	3.3.5 Microcontroller	36
	3.4 Parts Selection Summary.....	40
4.	Related Standards and Realistic Design Constraints	43
	4.1 Standards.....	43
	4.1.1 ANSI Related Standards and their Design Impact.....	43
	4.1.2 USB Standards.....	44
	4.1.3 C Standard	45
	4.1.4 ATmega Standards.....	46
	4.1.5 NEMA ICS 16	47
	4.1.6 Communication Interface Standards	48
	4.1.7 Industry Standard 26C31 Differential Line Driver and 26C32 Receiver	49
	4.2 Realistic Design Constraints	51
	4.2.1 Economic and Time Constraints.....	51

4.2.2 Environmental, Social and Political Constraints	52
4.2.3 Ethical, Health, and Safety Constraints	52
4.2.4 Manufacturability and Sustainability Constraints	53
5. Project Hardware and Software Design Details	55
5.1 Initial Design Architectures and Related Diagrams	55
5.2 First Subsystem, Breadboard Test, and Schematics	57
5.2.1 DC to DC Converter Design and Schematic	57
5.2.2 DC to DC Converter Breadboard Test.....	59
5.3 Second Subsystem, Breadboard Test, and Schematics	59
5.3.1 Status LEDs Design and Schematic.....	60
5.3.2 Status LEDs Breadboard Testing	62
5.3.3 Sensors	64
5.4 Third Subsystem, Breadboard Test, and Schematics	68
5.4.1 Motors	69
5.4.2 Encoders	73
5.4.3 Power Supply	75
5.4.4 Motor and Encoder Breadboard Test	76
5.5 Fourth Subsystem, Breadboard Test, and Schematics.....	78
5.5.1 Integration of ATmega	78
5.5.2 Testing of the ATmega	78
5.6 Fifth Subsystem, Breadboard Test, and Schematics	81
5.6.1 Joystick Design and Schematic.....	81
5.6.2 Joystick Breadboard Test.....	82
5.7 Sixth Subsystem, Breadboard Test, and Schematics	83
5.8 Software Design	85
5.8.1 Basic Arduino programming	85
5.8.2 Integrated Development Environment.....	86
5.8.3 Interrupt Service Routines.....	86
5.8.4 Meridian Flip.....	89
5.8.5 Varying Frequency	90
5.8.6 Bit Banging Square Wave VS Direct PWM Outputs	91

5.8.7 Analog Inputs from an Analog Joystick.....	93
5.8.8 Serial Parser.....	94
5.9 Summary of Design.....	96
6. Project Prototype Construction and Coding.....	99
6.1 Integrated Schematics	99
6.2 PCB Vendor and Assembly.....	102
6.2.1 PCB Vendor.....	102
6.2.2 Assembly	104
6.3 Final Coding Plan.....	105
6.3.1 Information Flow	105
6.3.2 Controlling the Motors.....	106
6.3.3 Joystick control	108
6.3.4 Parsing the Directions from the Computer Science Team	110
6.3.5 Feedback Control using Sensors and Interrupts.....	111
6.3.6 Meridian Flip and Motor Control	111
6.3.7 Work Load Distribution	113
7. Project Prototype Testing Plan.....	117
7.1 Hardware Test Environment.....	117
7.2 Hardware Specific Testing	119
7.3 Software Test Environment.....	127
7.4 Software Specific Testing.....	129
7.4.1 USB Input/Output	129
7.4.2 Pulse Frequency Ouput.....	129
7.4.3 Tracking.....	129
7.4.4 Encoder	130
8. Project Operation	131
9. Administrative Content	133
8.1 Milestone Discussion	133
8.2 Budget and Finance Discussion.....	136
8.3 Work Distributions	136
8.4 Personnel.....	137

10. Project Summary and Conclusions	139
Appendices.....	140
Appendix A – References	140
Appendix B – Copyright Permissions.....	142

List of Figures

Figure 1: House of Quality.....	7
Figure 2: Control Box Block Diagram for EE Design Project Scope	8
Figure 3: Overall System Block Diagram of Existing System	8
Figure 4: Software Block Diagram.....	9
Figure 5: Altazimuth Mount.....	11
Figure 6: Dobsonian Mount	12
Figure 7: German Equatorial Mount	13
Figure 8: Fork Mount	13
Figure 9: Refractor Telescope	14
Figure 10: Reflector Telescope	15
Figure 11: Catadioptric Telescope.....	15
Figure 12: Alternating vs. Direct Current	17
Figure 13: Three-Phase vs. Single-Phase.....	18
Figure 14: Hall Effect and Hall Voltage.....	19
Figure 15: STM17R-3NE NEMA 17, courtesy Applied Motion.....	24
Figure 16: Torque Curves, courtesy Applied Motion	24
Figure 17: Power Supply Block Diagram	25
Figure 18: PS150A24 24VDC Switching Power Supply, courtesy Applied Motion	26
Figure 19: Existing Configuration of Optical Switch and Interrupter	28
Figure 20: Linear Potentiometer	29
Figure 21: Capacitive Position Sensor	29
Figure 22: Magnetostrictive Position Sensors	30
Figure 23: Optical Position Sensor (Switch)	31
Figure 24: Buffer Logic/Totem-Pole Output Drive Architecture, Courtesy Texas Instruments	33
Figure 25: Mini Analog Joystick.....	34
Figure 26: 2-Axis Joystick.....	34
Figure 27: Thumb Joystick with Select Button.....	35
Figure 28: Signal Sampling and Digital reconstruction	38
Figure 29: Major Components	41
Figure 30: USB Description Graphic	45
Figure 31: Stacked Arduino shield.....	56

Figure 32: DC to DC converter Schematic.....	57
Figure 33: Webench Power Designer, courtesy Texas Instruments	58
Figure 34: Status LEDs from ATmega2560	60
Figure 35: Status LEDs for ATmega328	61
Figure 36: Multisim testing of LED	63
Figure 37: IR Diode Schematic.....	65
Figure 38: Optical Sensor Internal Components, Courtesy Texas Instruments ..	66
Figure 39: Optical Sensor Schematics.....	67
Figure 40: Status LED and sensor testing	68
Figure 41: Complete Motor/Encoder Schematic from SD1	69
Figure 42: STM17R Motor Connections, courtesy Applied Motion	69
Figure 43: Connections from Microcontroller to Motor, courtesy Applied Motion	70
Figure 44: Motion Profile with Step Smoothing Filter, courtesy Applied Motion..	71
Figure 45: Power Supply Front Panel, courtesy Applied Motion.....	76
Figure 46: Motor Breadboard Test.....	77
Figure 47: ATmega2560 with Connections.....	80
Figure 48: Joystick Schematic	81
Figure 49: Joystick Schematic with LEDs for Testing	82
Figure 50: Breadboard Test for Joystick	83
Figure 51: ATmega 328 Schematic	84
Figure 52: ATmega 328 Programming.....	85
Figure 53: ISR Priority Diagram	87
Figure 54: Actual Digital Pin Interrupt Mapping	88
Figure 55: Digital Deconstruction and Characteristics of a Wave	88
Figure 56: Duty Cycle of a Square wave	92
Figure 57: Analog signal transformed into PWM signal	92
Figure 58: Analog Signal Deconstruction into a Set Digital Resolution.....	94
Figure 59: ASCII character code used for common American computers	95
Figure 60: Integrated Schematic, First Revision	99
Figure 61: Integrated Schematic, Final Revision	101
Figure 62: Board Layout	101
Figure 63: Direct information flow of components in the telescope system	106
Figure 64: Final Code Logic Diagram	116
Figure 65: RA Gear Drive and Sensor Interrupter	122
Figure 66: Testing of Pointing Accuracy	126
Figure 67: Integrated Testing.....	127
Figure 68: Final Software Logic Flow Diagram	132
Figure 69: Project Gantt Chart, Senior Design I	134
Figure 70: Integrated Team Gantt Chart.....	135
Figure 71: Project Gantt Chart, Senior Design II	135

List of Tables

Table 1: Stepper vs Servo Motor	21
Table 2: Motor dipswitches	22
Table 3: Comparison of Motors	23
Table 4: Comparison of Power Supply	27
Table 5: Comparison of Sensors	32
Table 6: Comparison of Joystick Models	36
Table 7: Comparison Table for Microcontrollers	39
Table 8: Parts Selection Overview	42
Table 9: Comparison of Telecommunication Standards	50
Table 10: IR Diode Characteristics	64
Table 11: OPB980 Electrical Characteristics	66
Table 12: Motor Configuration Selections	72
Table 13: Comparison of PCB Vendors	104
Table 14: OPB980T51Z Electrical Characteristics	119
Table 15: OPB980T51Z High and Low Output Voltages	120
Table 16: Proposed Testing Configuration for Motors	121
Table 17: PWM.h Library Frequency Ranges	123
Table 18: AnalogRead() Test Results	124
Table 19: Projected Voltage Outputs for Joystick	124
Table 20: LED Electrical Characteristics	125
Table 21: Senior Design I Project Milestones Table	133
Table 22: Senior Design II Project Milestones Table	134
Table 23: Project Budget	136
Table 24: Subsystem Design Work Distribution	137
Table 25: General Tasks Work Distribution	137

1. Executive Summary

Although the groundbreaking for the Robinson Observatory occurred in January of 1994, the story of our current telescope begins in 2007. It was at this point that the existing 26" Tinsley telescope was removed and the existing 20" telescope, manufactured by RC Optical Systems, was installed. Although the installation of the device was led by Nate Lust, students played a significant role in the effort. From the beginning, the telescope was a partnership between the University of Central Florida and its students.

The 20" telescope is far from the largest or most powerful in use, so it has been designated a unique role. The telescope was designed to fill the niche of a rapid response device; that is, it can be deployed at a moment's notice, and is therefore uniquely situated to make time sensitive observations of astronomical phenomena. In addition to its academic and scientific applications, the telescope has also served to arouse an interest in astronomy for countless Scout troops, student groups and people of all ages. In 2015, the Orlando Sentinel recommended the Robinson Observatory as one of the "things you have to do before graduating UCF."

The telescope served faithfully from its installation until approximately three years ago, when its functionality began to degrade. In brief, the various subsystems of the telescope are controlled through a combination of software and dedicated peripheral devices. The focuser is controlled by a dedicated piece of hardware. Control of the dome is accomplished through the PC. Two Pittman 4431E064-R3, 24V DC, 500 CPR motors control the rotation and elevation of the telescope, which are in turn driven by a software package designated as TheSkyX. TheSkyX references a database of coordinates and pushes control signals to the motors. A controller, branded as Bisque TCS, sits between the PC running TheSkyX and the motors driving the telescope. This controller translates the commands of the software into the inputs that allow the motors to track various astronomical objects.

As it is currently understood, the heart of the problem lies in the ability of the telescope to accurately track the coordinates provided by TheSkyX software. Images produced by the telescope are blurry, unfocused and include "streaking" of illuminated objects, suggesting that one of the motors is not correctly compensating for the earth's rotation. At this time, the exact cause of this breakdown is unknown; it could be due to the software itself (as the problems became more pronounced after an update of the software), an issue with the motors or the translation of the commands by the Bisque TCS controller. In addition to the tracking issues, the telescope is no longer correctly reporting its position back to TheSkyX software package.

Until initial investigation and reverse engineering are underway, it is impossible to say whether the Bisque TCS controller is functioning as intended. However, this

is a proprietary piece of hardware. There is a single individual who is able to service this equipment, and he must be flown in at great cost to the University any time that service is needed. Therefore, even if the controller is operating, it is the desire of the Robinson Observatory to replace it with an open source design. The core goal of our team was to make adequate progress towards replacing the controller. The first step for us was to create a scale model for use as a testing platform without posing harm to the expensive existing hardware such as the custom Pittman motors. It bears mentioning that the intent of our team was for this scale model design to be robust enough to serve as a replacement for the Bisque TCS controller, if the Observatory staff decided to implement it.

2. Project Description

The scope of this project was modified as the interdisciplinary teams began to have a more complete understanding of the state of the observatory's telescope, mount, proprietary software and embedded telescope controller. In brief, our overall project consisted of a scale-model of the current version of the 20" telescope. The electrical engineering team was focused on providing the embedded portion of this integrated device. That is, the piece of hardware that sits between the PC and the mount and translates positioning commands from the PC into the appropriate motor movements.

2.1 Project Motivation and Goals

The 20" telescope serves as the showpiece of the Robinson Observatory. In addition to performing the bulk of the scientific observations for the astronomy team, it is a significant draw to youth groups and has served as a destination for Scout troops, school fieldtrips and the general public. When the telescope is operating as designed, it fills scientific, academic and social needs of the Central Florida community.

It was the desire of our team to engage with a Senior Design project that has a lasting impact. We do not wish to diminish the creativity, technical challenge and opportunities for learning that are inherent in many other Senior Design projects, but the simple fact is that many of them are relegated to a storage closet after the team graduates. It is our belief that the successful execution of this project will restore a resource that will benefit the community for years to come.

In addition, our team was inspired by the close cooperation that has existed between the Robinson Observatory and the UCF student body since the initial installation of this telescope. We all feel that we have benefited greatly by our time at UCF and were excited to have an opportunity to continue this partnership and give back to the University that has been our home for the last few years.

2.2 Objectives

The minimum viable product, as defined by our contact at FSI, was sufficient progress towards the replacement of the proprietary controller that translates TheSkyX commands to the motors that drive the telescope. This was first and foremostly achieved through the design of a PCB that decodes a string of motor command signals as inputs and controls a scale model of the telescope. In a broader sense, the intention was to restore full functionality to the telescope. However, since parts of this goal were dependent upon the efforts of other teams working the project (e.g. if the motors were not restored to full functionality, no amount of effort from the E.E. team would have been able to overcome this

deficiency), the overall objective for the E.E. team was to make progress towards creating an open source replacement of the Bisque TCS controller, as requested by the customer.

There were a number of additional goals that have been defined by the Robinson Observatory team (and are enumerated in more detail in the Requirement Specifications section of this document). Key examples are wireless functionality and the ability to tie various telescope subsystems (e.g. focuser, dome control, etc.) into TheSkyX software to afford an all-in-one solution for observatory control.

Since much of this project was defined by the completion of the scale model, these objectives did not fit within the scope of our project. If we were to have extra time, we would have like to incorporate some of these “want tos” into our design. Since that was not the case, we have left room in the project (e.g. additional input/output pins on our controller and sufficient documentation) for a team to follow behind us and continue the observatory update.

2.3 Challenges

Before the scope of our project was officially decided, our team envisioned a number of challenges associated with this project. First, this project started as somewhat outside the scope of a traditional Senior Design effort. Instead of designing and implementing a project from the ground up, we were originally tasked with designing around a large amount of expensive, existing equipment. The cost of the equipment made it impractical to replace, therefore, our first order of business was to develop a scale model to demonstrate our understanding of the protocols that drive the telescope.

This first challenge was compounded by the fact that existing documentation for the hardware was sparse or nonexistent. Our team reached out to Pittman in an effort to better understand how to drive the motors and were told that the motor is proprietary and that they would not be able to offer any support. We were still able to find minimal documentation on the equipment, which helped us make some of our design choices later. During investigation, we also realized that the single individual who services this controller would not have been incentivized to work with a team who was attempting to make his equipment obsolete. These factors made us realize how large the reverse engineering portion of the project would be, which ultimately led us to narrowing the scope of our project to solely the scale model.

A second challenge was in finding the root cause of the problem with the telescope. At a very high level, there are three components in play: TheSkyX software, the Bisque TCS controller and the Pittman motors. At this time, it is unknown which of these elements is causing the breakdown, and therefore further analysis will be required before functionality can be restored.

A third challenge was found in the budget of the project. Our contact at the Florida Space Institute (FSI) oversaw engagement with the Florida Space Grant Consortium (FSG) to secure funding for the project, but the timeline for received funding ended up being delayed until the middle of the second semester of senior design. In the beginning of the project, each team was expected to receive \$750, however it was later clarified that a total of \$1,000 was to be split between the three teams. It came as a relief that the Computer Science team did not need their portion of the grant money, and therefore the remaining teams were able to have more of a leeway in selecting quality components without worrying about going too much overbudget.

A final challenge was rooted in working with an interdisciplinary team, where each sub-team had different deliverable requirements for UCF leading to possibly conflicting scopes of the overall project. There was a team of three mechanical engineering students and four computer science students committed to the project in addition to our team. The electrical team was able to 3D print their own temporary mount for testing purposes which came in handy when the mechanical design implementation was severely delayed due to manufacturing challenges. However, for the most accurate scale model, as well as the best chance for accurate tracking, all teams relied on the mechanical team to provide the final model with correct gear ratios and counterbalances to most accurately reflect the Observatory telescope during final demonstrations and showcase.

2.4 Requirements Specifications

The following specifications were created by our team in order to influence our design decisions in a way that would provide the most benefit to the future users of our scale model at The Robinson Observatory. There were a few quantitative design constraints placed upon this project. Rather, the challenge, and the requirement, came from the fact that the below specifications were to be implemented using existing open-source software (Stellarium). Design choices and budget informed quantitative design choices rather than explicit requirements from the customer (Robinson Observatory).

- Shall accept an input voltage of 120VAC +/- 15%.
- Shall have a sensor response time of less than 2 seconds.
- Shall have a cost of less than \$800.
- Shall have a power usage of less than 100W.
- Shall have dimensions less than 20" x 20" x 10".
- Shall have a weight of less than 5lb.
- Shall have an execution time of less than or equal to 60 seconds.
- Shall have a pointing accuracy of less than or equal to 3.5°.
- Shall interpret control signals from Stellarium software.
- Shall relay motor control signals to stepper motors.

- This includes both right ascension and declination as well as slew rates from Stellarium.
- Shall accept secondary input from user operated joystick to move motors manually at variable slew rates.
- Shall support home and park capabilities for the telescope.
- Shall support pointing limits (no declinations below the horizon; no horizontal azimuths that will damage the telescope)
- Shall support the ability to work in multiple modes:
 - Sidereal tracking: in which the declination motor does not move and right ascension motor tracks at sidereal rate
 - Nonsidereal tracking in which both motors move at non-standard tracking rates
 - These targets are delivered from Stellarium

2.5 House of Quality Analysis

In our attempts to understand the full system that is the UCF Robinson Observatory, the scale model described above served as the testing platform for our senior design team to use to learn how to control the core components of a telescope. However, the scale model that we made needed to serve a use for our customers at FSI so that future teams may expand upon it to eventually replace their entire existing control box. An analysis of whether our engineering requirements for this scale model met customer requirements is depicted in the house of quality chart in Figure 1 below.

Legend:

- + = Positive Polarity (Increasing Requirement)
- = Negative Polarity (Decreasing Requirement)
- ↑↑ = Strong Positive Correlation
- ↑ = Positive Correlation
- ↓↓ = Strong Negative Correlation
- ↓ = Negative Correlation

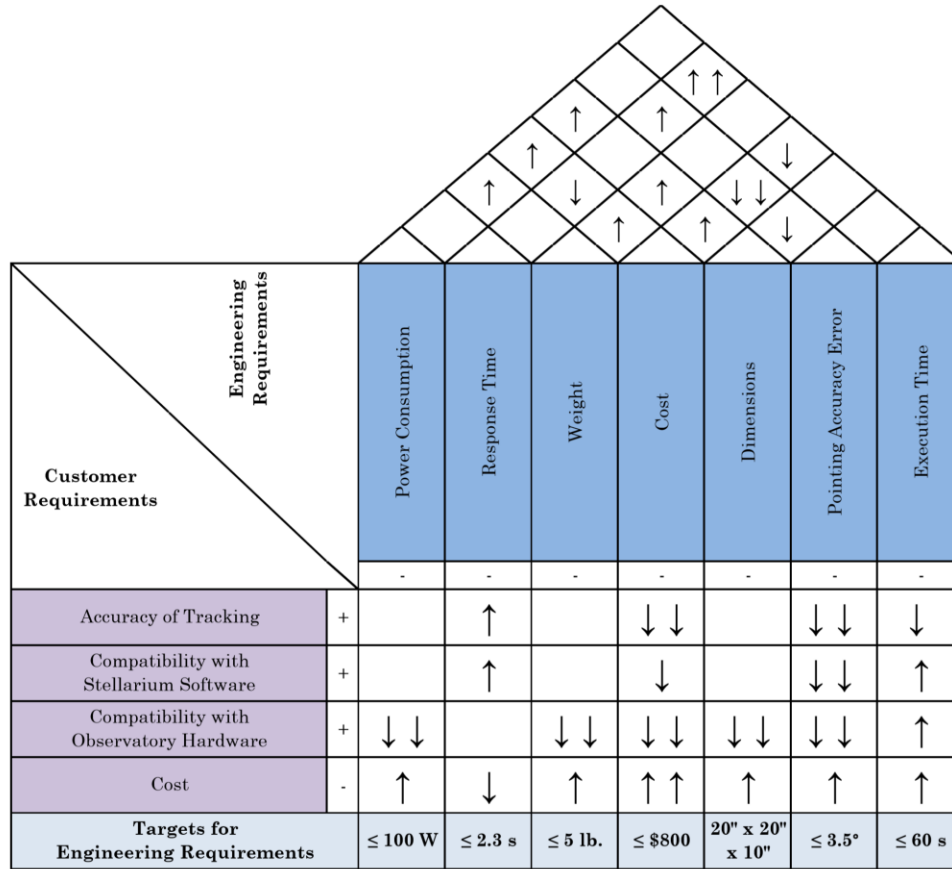


Figure 1: House of Quality

2.6 Block Diagrams

The essential components for a minimum viable product are shown in Figure 2 and Figure 3 below, with the exception of the camera module. The camera was not part of our electrical team’s design and was not a requirement of the project. The computer science team used a camera in their design and during integration their camera module would replace our laser pointer as the “telescope” on the final mount. The purpose of the laser pointer for us was to verify tracking, while the purpose of the camera for the computer science team to track near space objects easily visible in the sky to better mimic what the observatory serves to do.

The distribution of work among our team is shown on Figure 2 as far as design choices are considered. Investigations of the existing system that occurred during the initial weeks of the project design gave us a clear picture of the components in the existing system, shown in Figure 3.

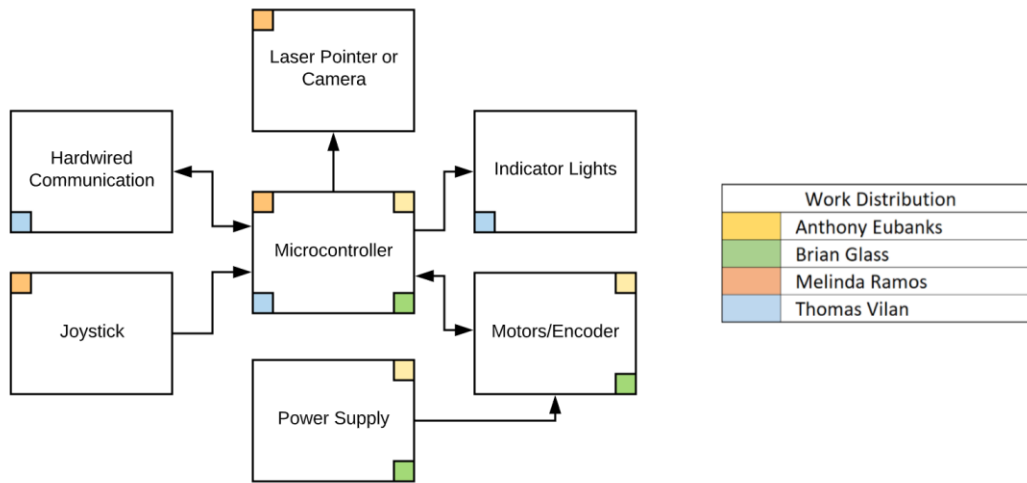


Figure 2: Control Box Block Diagram for EE Design Project Scope

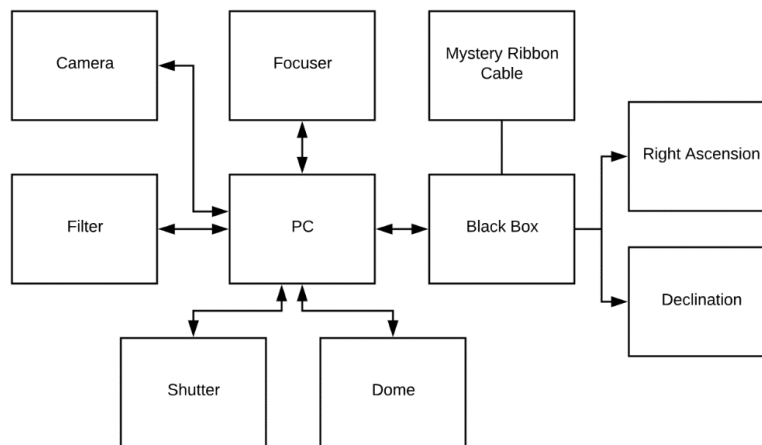


Figure 3: Overall System Block Diagram of Existing System

In Figure 4, the initial software block diagram is shown illustrating how the software on the PC communicates control signals to the existing Bisque TCS box which in turn controls the operation of the telescope with a secondary input of a joystick. In our design, in combination efforts with the Computer Science team, we attempted to most closely resemble this software logic.

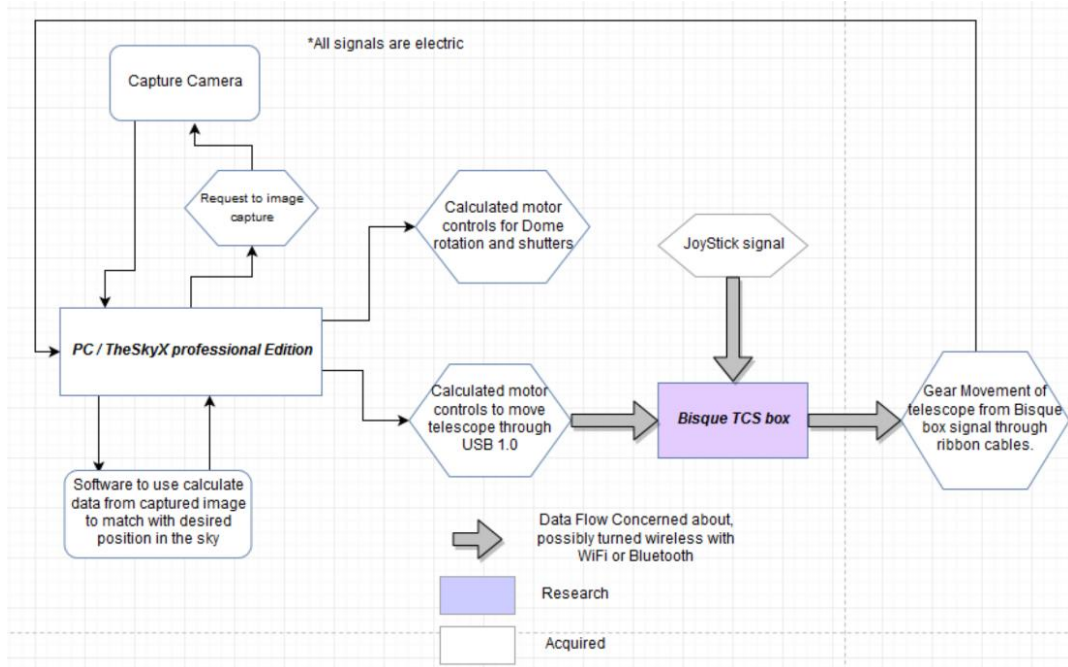


Figure 4: Software Block Diagram

3. Research Related to Project Definition

This section outlines the research related to our project before we had performed any design work. Research before designing was necessary because it helped us as a team understand the current products in the market and the current technology used. This would help us as a team provide a product that meets the customer's needs but at an advanced level. Once an understanding of the technology and similar products was determined, the parts selection was the next step in designing our system.

3.1 Existing Similar Projects and Products

This project for the University of Central Florida (UCF) and the Florida Space Institute (FSI) was unique. Typically, senior design projects are sponsored interdisciplinary projects, or they are performed by single disciplines. This project was different in that it had been developed around an existing system that needed to be redesigned. Instead of starting with a fresh design, this undertaking began with some amount of reverse engineering to understand the system; only then could the replacement system be designed. This not only added an immense amount of work to a project that was difficult to complete on time by the need for first understanding the current system before designing the replacement system, but it also provided the senior design team with experience that other students will not obtain. In the past, there has only been one other senior design project at UCF that concentrated on an existing system that needed to be redesigned.

The end goal of this project was to build a model that accurately represents the functionality of the current system at the Robinson Observatory. Building a model was essential because a model could assist in troubleshooting the current issues with the full-scale system without the risk of damaging the current system. Unfortunately, it was not feasible to replace the existing system through the efforts of just one senior design team; this system is too complex. A second, or possibly even a third, team will be needed to completely replace the existing system.

Despite extensive research, the team did not find information on any projects from a student level that either redesigned an existing observatory system or developed one from the ground up using telescopes with lenses. There are other projects performed by students that involve the design and construction of radio telescopes. These telescopes have similar functionality to the one used at the Robinson Observatory in the sense that they detect naturally occurring radio frequencies from celestial bodies. A series of senior design teams from Iowa State University developed a radio telescope system to be used at their university.

3.2 Relevant Technologies

Software Bisque, the developer of the current technology used at the observatory, was one of the first to come to market with a telescope control software that could be programmed to track, eliminating the need to have someone constantly operate the telescope. This breakthrough was revolutionary because the user could operate the telescope from across the United States or even program it to track the night sky while everyone sleeps. Software Bisque started off as a software company that provided astronomy software for specific mounts, but they began to develop their own mounts once their software capabilities exceeded the hardware capabilities of the current mounts on the market. Descriptions of the types of mounts in the market today, including the mount currently used at the Robinson Observatory (German equatorial mount), are discussed in further detail below.

There are many different types of mounts used for telescopes, including Altazimuth, Dobsonian, equatorial, German equatorial, and fork mounts. Altazimuth mounts, also known as Alt-Az, as seen in Figure 5, are the least complex mounts; they provide two motions: altitude and azimuth. These mounts feature slow-motion knobs that allow them to make accurate adjustments and aid in smooth tracking. These mounts are used for terrestrial observing and offer scans of the sky at lower power; however, they are not useful for sky photography. The computer-driven versions of these mounts provide more precise tracking of the sky [1].

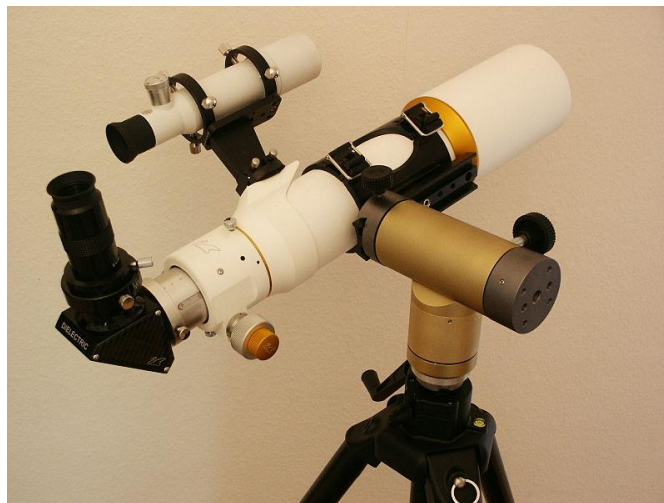


Figure 5: Altazimuth Mount

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

Invented by John Dobsonian in the 1970s, the Dobsonian mount, like the one shown in Figure 6, is essentially a modified version of the Altazimuth mount. These mounts are secured on the ground by a heavy platform, and they are designed to

support large Newtonian reflectors without sacrificing their ability to maintain a steady image. A Newtonian reflector, also called a Newtonian telescope, is a type of reflecting telescope that uses a concave primary mirror and a flat diagonal secondary mirror. Dobsonian telescopes usually have large apertures, ranging from 6" to 20" or more [2].



Figure 6: Dobsonian Mount

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 4.0 International license from commons.wikimedia.org

Equatorial mounts are excellent tools for astronomical observing throughout extended periods of time. Unlike non-computerized Altazimuth mounts, they are necessary for astrophotography. When using an Altazimuth mount, stars that are stationary will move out of view, but these stars, which only appear to move due to the earth's rotation, can be captured by an equatorial mount. Properly aligned, a telescope on an equatorial mount can be directed toward an object and then guided via an electric motor or manual controls. There are two basic types of equatorial mounts: German and fork.

Newtonian reflectors and refractor telescopes typically use a German equatorial mount. The German equatorial mount, seen in Figure 7, is distinguished by a large counterweight that extends on the opposite side from the telescope. Without this counterweight, the telescope would be unbalanced. A German equatorial mount is the type of mount that is currently being used at the Robinson Observatory.

An issue with the German equatorial mount is that most German equatorial mounts require the telescope to be flipped at the meridian line. This requirement comes from the way the mount is designed is that at the meridian, the telescope can come into contact with the mount and cause damage to the telescope or mess up the tracking of the telescope.



Figure 7: German Equatorial Mount

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported license from commons.wikimedia.org

The fork mount, shown in Figure 8, is typically used by catadioptric and other shorter optical tubes because it is more convenient than the German mount. Fork mounts are especially useful for astrophotography. The most common mount for modern research telescopes, a fork mount is operated by a computer, which controls the telescope. The computer calculates the altazimuth setting by utilizing an internal, digital equatorial drive. Since it is completely automatic, the fork mount simplifies observation, making it easier for the observer to find celestial objects. For example, an observer could point the telescope in one direction and enter the latitude and longitude, and then the computer would finish the alignment, directing the telescope to the location of the desired object [2].



Figure 8: Fork Mount

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported license from commons.wikimedia.org

There are a variety of telescopes, each with a unique design. The telescopes that are available today cover all the bands of electromagnetic radiation, from gamma rays to light to radio. This paper, however, will focus on the three types of optical telescopes: refractor, reflector, and catadioptric [2].

Refractor telescopes are the earliest type of telescope. Refractor telescopes are essentially a long tube with lenses on both ends. They work by concentrating the light, passing it through a common focal point on the two lenses. Compared with other telescopes, refractor telescopes are inexpensive.

Without an obstruction to block light, refractor telescopes can provide magnified images that are detailed and clear. The downside to using refractor telescopes is that they are normally heavier and longer. Additionally, because they are inexpensive, their size and aperture are limited, and they are prone to chromatic aberration. A layout of the refractor telescope is shown in Figure 9 [2].

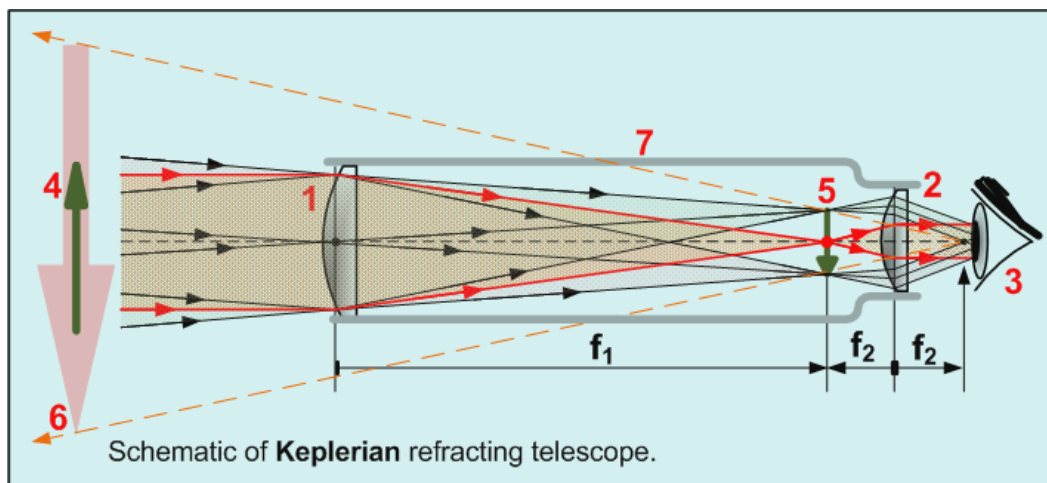


Figure 9: Refractor Telescope

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

Unlike refractor telescopes, reflector telescopes use mirrors. A large concave mirror focuses the light, which is then redirected by a smaller mirror into the eyepiece, producing a clear image. These telescopes can be very large because all the main optical equipment is on one end. A Dobsonian telescope is an example of a reflector telescope, which is shown in Figure 10.

Also, unlike refractor telescopes, reflector telescopes do not have chromatic aberration, and they are even more inexpensive to make than refractor telescopes. Reflector telescopes are excellent for deep sky viewing, and they even work if the mirrors are dusty. The disadvantages to reflector telescopes are that they are high maintenance, not very durable, and prone to coma aberration [2].

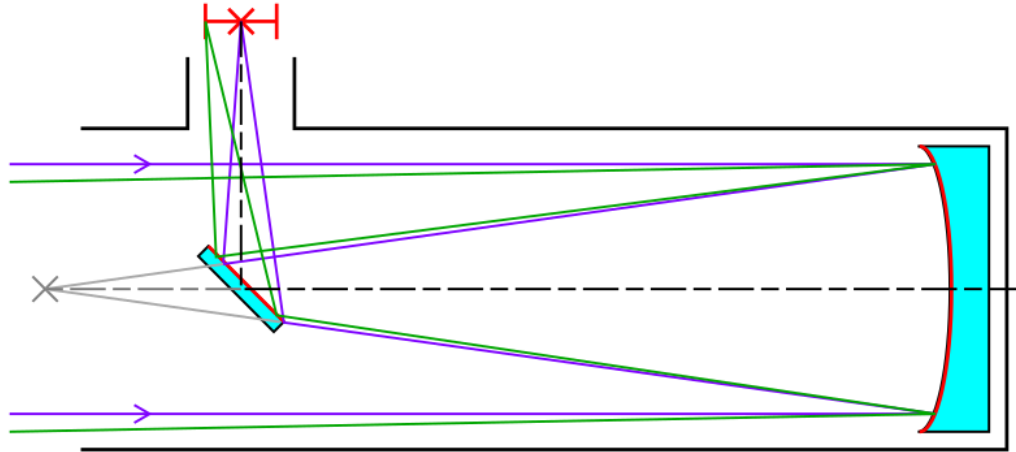


Figure 10: Reflector Telescope

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 4.0 International license from commons.wikimedia.org

Catadioptric telescopes, shown in Figure 11, blend the benefits of both refractors and reflectors because they combine a lens with two mirrors. They are the most expensive because they have a more elaborate design, and they are more compact than the two types of telescopes mentioned above; however, these are the most popular telescopes on the market today. The advantages of catadioptric telescopes are that they are easy to use, portable, durable, and versatile—they can be used for viewing deep sky objects, planets, stars, and even the moon [2].

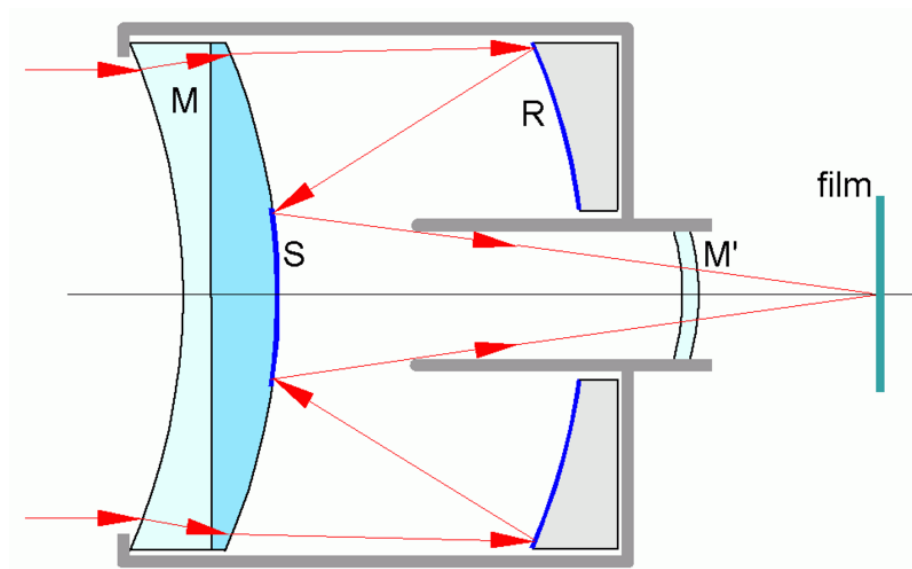


Figure 11: Catadioptric Telescope

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

The list of the different versions of software useful in astronomy is quite extensive. For simplicity, the programs discussed below are categorized according to their features.

Planetarium software can map the night sky from any location on the Earth. Typically, this type of software gives users the option to print out star charts for a night of viewing, and it offers a large database that contains, at minimum, the most popular night sky objects. The most popular planetarium software available is undoubtedly SkyX by Software Bisque, which is the software used at the Robinson Observatory. The SkyX software is available in a few different versions: a student edition, a serious astronomer edition, a professional edition, and even a pocket edition that provides access to a virtual sky from your personal device. Other types of planetarium software include Google Earth, SkyMap, World Wide Telescope, and Redshift.

Unlike planetarium software, which provides users with features for the full sky, specialty observing programs focus on aiding users who want to concentrate on specific objects. As mentioned above, TheSky from Software Bisque is the smoothest and most cohesive software, but DeepSky can provide a list of available targets for the evening so users can plan their viewing. Seeker, also by Software Bisque, is a 3D immersive tool for traveling through the solar system. Other specialty observing programs include VRMars, Night Sky Observer, and Heavenscape.

Robotic and remote-control software specializes in controlling telescopes from a remote location. TheSky used with CCDSoft (for imaging) allows users to control a telescope from a computer that is nearby. Other versions of this type of software include Orchestrate, which automates imaging sessions; PoleAlignMax, which assists the computer in pointing the telescope north; and SN Finder, which automates supernova searches.

3.3 Strategic Components and Part Selections

This section outlines the many different options considered for the selection of all major components incorporated into our project. The major components that constitute the project include the declination and right ascension motors, the power supply unit, the sensors, the joystick and the main microcontroller. Before down selecting to a specific part number and manufacturer, all available and relevant options on the market were explored. Following the corresponding section for each major component is a brief explanation of why the selected component was chosen as well as a table comparing selected characteristics across multiple manufactures. The highlighted column indicates which part was procured for this project.

3.3.1 Declination and Right Ascension Motors

Before determining the correct motor for this specific application, an in-depth understanding of the different types of motors—their functions, strengths, and weaknesses—needed to be established.

At the most basic level, there are two types of motors, alternating current (AC) motors and direct current (DC) motors. Both types of motors convert electrical energy into mechanical energy; however, there are significant differences between the two main types of motors in terms of how they are constructed and controlled. The most important difference between AC and DC motors is the type of current they use—AC motors are powered by alternating current, while DC motors use direct current [3].

The main difference between alternating current and direct current is that with alternating current, the flow of charge changes direction periodically. Direct current is easier to understand since the current does not alternate but flows in a steady manner [4]. Figure 12 below illustrates the flow of alternating and direct current.

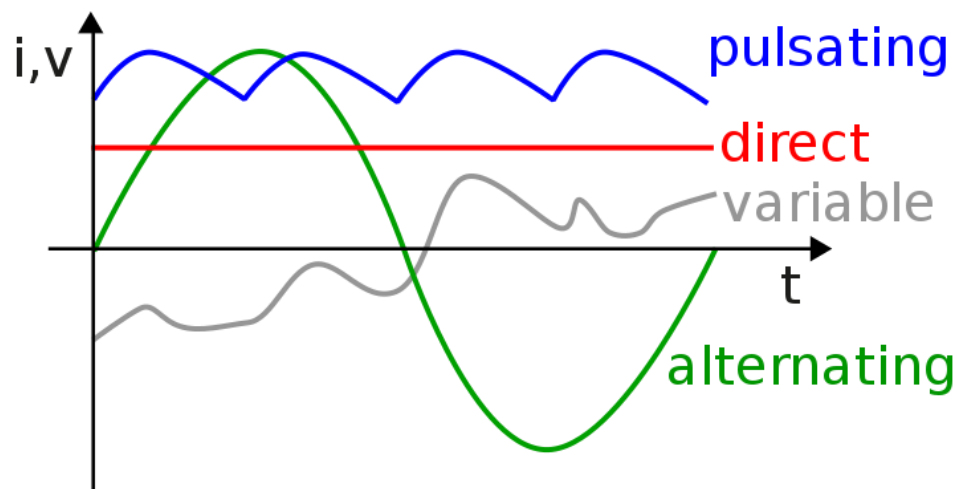


Figure 12: Alternating vs. Direct Current
Image available to public domain by Wikipedia.org

AC motors offer flexibility and certain advantages that DC motors do not. AC motors require minimal maintenance and a lower power demand on start. They also have adjustable torque limit and controlled acceleration. AC motors generally fall into two different categories: synchronous and induction (asynchronous) [5].

The defining characteristic of a synchronous motor is that it synchronizes the rotor's rotation with the frequency of the supply current. (In a motor, the rotor is the part of the motor that rotates, whereas the stator is the part of the motor that does

not rotate.) Synchronous motors are ideal for driving equipment at a constant speed because the motors ensure that the speed does not change regardless of the load [5].

While synchronous motors work because of how they synchronize the rotor, induction motors work because of how they use the other main part of a motor: the stator. The winding of the stator produces a magnetic field, and the induction motor uses electromagnetic induction from this field to produce an electric current. Induction motors are the most common type of AC motor. Their importance in the industry stems from their load capacity, with single-phase being used for smaller loads and three-phase motors being used for industrial purposes [5].

A single-phase system does not deliver power at a constant rate because of the oscillations in the signal. With the peaks and dips in the voltage, the power fluctuates considerably. If two more phases are introduced 120° out of phase, it becomes a three-phase system, and the power becomes almost constant. Figure 13 below shows the difference between single- and three-phase systems [6].

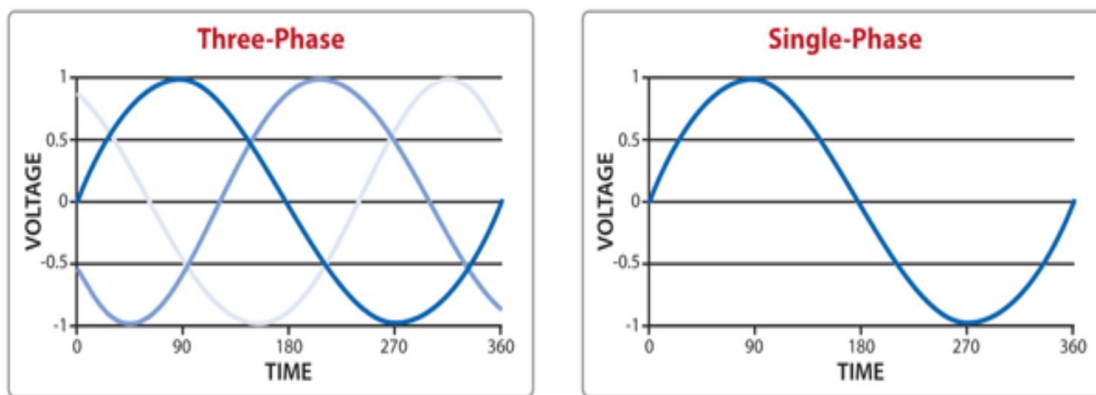


Figure 13: Three-Phase vs. Single-Phase

DC motors are the first motors that experienced widespread use. For lower power systems, their initial costs are typically less than AC motor systems. DC motors can be easily controlled by simply varying the voltage that is supplied to the motor. There are many advantages to DC motors including high-starting torque and excellent speed control. They are also quick to start and stop and easy to install. Featuring a simplistic design, DC motors can both accelerate and reverse quickly. There are some disadvantages, namely, their vulnerability to dust and their maintenance needs [5].

There are two main types of DC motors: brushed and brushless. A brush, on a motor, is a carbon device that conducts current between stationary wires and the moving parts, typically a rotating shaft. For brushed motors, there are a few different types: series wound, shunt wound, compound wound, and permanent magnet.

Series wound motors are constructed so that the field winding is in series with the rotor winding. Varying the input supply voltage controls the speed, which is not very effective; as the torque increases, the speed decreases. These motors are used in a variety of applications, including lifts, cranes, hoists, and automotive machinery [5].

Shunt wound motors use a field winding connected in parallel with the rotor winding, which helps deliver increased torque without sacrificing a reduction in speed just by increasing the motor current. Since the winding is in parallel, the starting torque is not as high as in the series wound. Shunt wound motors can be found in vacuum cleaners, grinders, and lathes [5].

Compound wound motors implement a combination of the series and shunt wound rotor design. Compound motors have a high starting torque and operate smoothly. Applications for these motors include compressors, rotary presses, elevators, and continuous conveyors [5]. Permanent magnet motors, as their name implies, use a permanent magnet. This type of motor is preferred for applications requiring high levels of precision, such as robotics [5].

There are some issues with brushed DC motors; specifically, their short life span when used frequently. Brushless motors eliminate some of these issues by using Hall Effect sensors to detect the rotor's position, which a controller can use to accurately control the motor [5]. To understand what a Hall Effect sensor is, an understanding of what is known as the Hall Effect is critical.

As electric current flows through a material, the electrons mostly move in a straight line. If you put the material in a magnetic field, the force acts on the electrons, which causes them to stray from their straight-line path. This results in one side having more electrons than the other, creating a potential difference (voltage). Measuring the potential difference will provide the Hall Voltage developed [7]. Figure 14 shows this Hall Effect. The Hall Effect sensors are then activated by an external magnetic field. As a motor turns, the magnetic field changes, making a Hall Effect sensor perfect for this application.

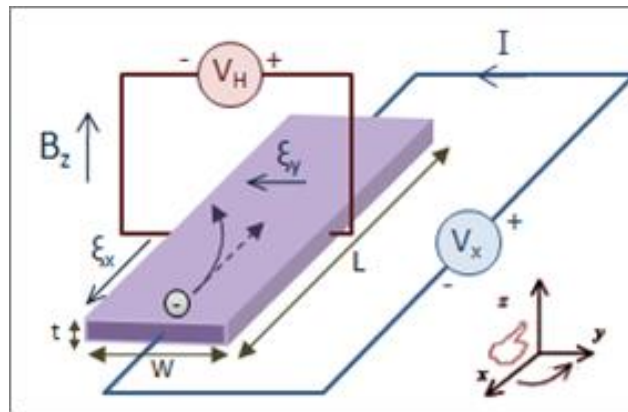


Figure 14: Hall Effect and Hall Voltage
Image available to public domain by Wikipedia.org

The two main types of brushless DC motors are stepper motors and servo motors. Servo motors contain a brushless DC motor, potentiometer, and a control circuit. Gears attach the motor to the control wheel. The motor's rotation affects the potentiometer's resistance, allowing the control circuit to control the amount and direction of the movement [8].

Stepper motors consist of a rotor with permanent magnets and a stationary stator that carries the windings. A big difference between stepper and servo motors is a stepper motor typically has a lot higher pole count, and because of this, the motor appears to have a more continuous operation. Since stepper motors generate incremental motion, they are usually run open loop, which reduces the cost and complexity of the design. This open-loop operation is not as accurate as the closed loop design of a servo motor, so if a more accurate operation is needed, an encoder can be used.

Encoders are electromechanical devices that provide electrical signals; these signals are useful for controlling speed and position. They convert the motor's mechanical motion into an electrical signal; the control system can use this signal to track how well the motor is operating and make any necessary adjustments. There are different options for mounting an encoder for control: shafted, hub/hollow shaft, and bearing less [9]. Shafted encoders connect the motor encoder shaft to the motor shaft via a coupling method. This provides mechanical and electrical isolation from the motor shaft.

Hub/hollow shaft encoders are directly mounted to the motor shaft. These encoders are connected with a spring-loaded tether. Easy to install, these encoders do not require a shaft alignment; however, it requires extra caution to provide electrical isolation using this method. A bearing less encoder, also known as ring mount, contains a sensor assembly that is formed like a ring, which is mounted on the face of the motor, and a magnetic wheel, which is mounted on the motor shaft [9].

Motor drives are important for the operation of a motor. The electronic drive gathers and directs the electrical energy to a motor. It has the ability to control how much and how often the electricity goes to the motor; therefore, the drive can influence the speed and torque [10].

3.3.1.1 Project Application

After obtaining an in-depth understanding of the types of motors and their uses, our project team was able to decide which motor was the best choice for our application. The two types of motors that we considered were DC servo motors and DC stepper motors. For our purposes, we needed higher torque with not much speed, but we also needed high position accuracy for sensing objects in the sky. Table 1 compares the main advantages of stepper and servo motors.

Table 1: Stepper vs Servo Motor

	Stepper	Servo
Torque	High at low speeds	High even at high speeds
Accuracy	High with encoder feedback	High
Speed	More suitable for low speeds	More suitable for high speeds
Cost	Low	Moderate
Lifespan	High	Moderate

Considering our needs and the features of each motor, the best option for our application was a stepper motor with an encoder. There were many options when choosing the appropriate stepper motor. Some of the more significant requirements we considered were the step angle (1.8° or less), high torque, and the cost.

After an extensive search, the motor we decided to use is a NEMA 17 Integrated Drive and Motor with Encoder. We chose this motor for many reasons. First, having a motor with an encoder and a drive that is already integrated into the motor decreases the complexity of the design immensely compared to having to locate a drive and encoder and integrate them into a motor. Also, the NEMA type motor is a low-cost and common motor that provides high torque over a lower speed range. The motor also has a wide range of settings that make it easy to adapt to any situation.

These settings can be changed by the dipswitches located on the motor in between the connections for the motor and the encoder pins. The most important dipswitch settings are changing the steps per revolution. Most motors have a fixed number of steps per revolution which ultimately limits the design capabilities. The motor chosen has the option to change from 200 steps per revolution to 25,000 steps per revolution. To be able to handle the change in steps per revolution however, the signal being sent to the motor has to be able to reach higher frequencies as the step count increases. 25,000 steps will not be able to be achieved because even sophisticated PLC's cannot provide frequencies that high for precise motor control but being able to change to 400 or even 800 steps will provide flexibility for the mechanical engineering (ME) team as well as the electrical engineering (EE) team.

The dipswitches are broken up into two sections. One set of dipswitches is used to change the steps per revolutions for the motor and the other set is to change the settings of the motor itself, such as filtering and current. Tables 2 below shows the different dipswitches and what they're used for.

Table 2: Motor dipswitches

	1	2	3	4	5	6	7	8
Current	X	X						
Idle Current			X					
Self-Test				X				
Pulse Noise Filter					X			
Smoothing Filter						X		
Load Inertia							X	
STEP/DIR								X

The current setting for the dipswitches 1 and 2 allows the user to change the amount of current that the motor uses. To obtain maximum torque, the current setting would need to be a maximum 100%, but if there are conditions where power consumption is an issue, the current level can be changed to as low as 50% to conserve power. It is important to note that a reduction in current to this level will reduce the torque output considerably.

The idle current setting allows the motor idle current to be lowered 50% or 90%. The lower level of the current for this idle setting, the lower amount of holding torque this motor will have when the motor is not turning. For large loads that could potentially overcome the holding torque of the motor, a higher idle current is recommended. The self-test switch is a simple but useful dipswitch within the motor. The switch can be turned on to test the motor to make sure it is receiving power. This could be used when diagnosing the motor issues.

Pulse noise filter dipswitch gives the user the ability to add filtering for noise that could affect the STEP signal by causing the drive to interpret pulses improperly. The two options, 150kHz, or 2MHz provide the user to toggle on whichever frequency is near the range that they will be using for turning the motor. Another filter, the smoothing filter is used for motors that are used at lower step resolutions. At lower step revolutions, the motor can run rougher than at a higher step count because of the increased production of noise. With the smoothing filter, it enables the motor to run at lower steps per revolution with the same smoothness as the higher steps per revolution.

Load inertia is an anti-resonance and electronic damping feature that improves the motor performance. The load inertia must be calculated, then divided by the rotor inertia which was given by the manufacturer, and that will determine the setting for that dipswitch.

Lastly, the STEP/DIR dipswitch is to provide two different options for sending signals to the motor. STEP and DIR can be two separate signals, or STEP and DIR can be sent on the same line. DIR determines the direction of the motor, and STEP determines the speed of the motor.

For separate operation, a pulse with a frequency corresponding to the speed is sent on the STEP line, and a logic HIGH or LOW is sent on the DIR line to determine CCW or CW rotation. If using STEP and DIR together, sending a signal to STEPCW will turn it CW and STEPCCW will turn it CCW with the speed being proportional to the frequency of that signal and the steps per revolution set with the dipswitches. A few of the options of the steps per revolution of the motor include 200, 400, 800 or 1600 steps.

There are many other options for steps, but the requirement for the input frequency to the motor increases with an increase in steps. The highest number of steps obtainable is 25000 steps. To command the motor speed of 50 RPS, the pulse frequency would need to be 1.25MHz, which is higher than we would expect an ATmega2560 to output effectively.

A comparison of options for the stepper motor used can be seen below in Table 3.

Table 3: Comparison of Motors

	STM17R-3NE NEMA 17	STM17R-3ND	4209S-1P
Steps per Revolution	200, 400, 800, 1600, 3200, 6400, 12800, 25600, 1000, 2000, 4000, 5000, 8000, 10000, 20000, 25000	200, 400, 800, 1600, 3200, 6400, 12800, 25600, 1000, 2000, 4000, 5000, 8000, 10000, 20000, 25000	400
Built in Encoder	YES	NO	NO
Holding Torque	68 oz-in	68 oz-in	31 oz-in
Length	2.64"	2.64"	1.34"
Weight	14.7 oz	14.7 oz	7.04 oz
Operating Voltage	12V, 24V, or 48V	12V, 24V, or 48V	24V
Cost	\$204.00	\$118.00	\$49.10

Between the three motors above, the reason for choosing the first one is mainly because of the built-in encoder feature. It was not possible to find a motor for less

than that that included a built-in encoder. Since the existing design at the observatory has a built-in encoder, the team and sponsors suggested using a similar motor.

See Figure 15 and Figure 16 below for the motor of choice and the torque curve.



Figure 15: STM17R-3NE NEMA 17, courtesy Applied Motion

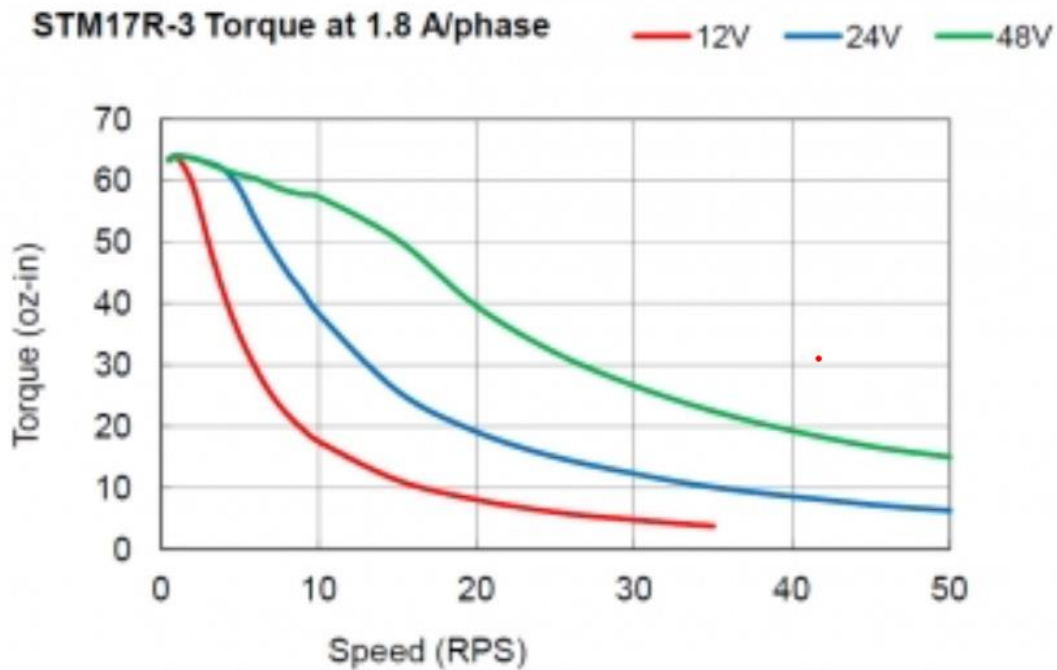


Figure 16: Torque Curves, courtesy Applied Motion

3.3.2 Power Supply Unit

A power supply unit is the part of the electrical system that converts the power provided by an outlet (normally 115VAC or 230VAC) or another source to a usable level inside an electrical device (typically 5VDC). A block diagram for a typical power supply can be seen in Figure 17 below.



Figure 17: Power Supply Block Diagram

The input transformer converts incoming voltage to the level required to convert to DC at an acceptable level. In this case, and in most cases, the type of transformer used is called a step-down transformer. A step-down transformer decreases the voltage of the incoming signal, or steps it down to a lower level. Transformers also provide isolation between the output and the supply from the incoming line.

The rectifier converts the voltage that was stepped down from AC to DC from the transformer. After the signal is rectified, it still contains unwanted ripples that can be filtered out using a filter capacitor to smooth the signal. The final device in the power supply process is the regulator. A linear regulator compares the output voltage with a precise reference voltage and adjusts the device to maintain a constant level of output voltage.

A power supply that does not contain a regulator is called an unregulated power supply. An unregulated power supply is simple and therefore cheaper than a regulated power supply, but because they are unregulated, the output voltage from this type of power supply can fluctuate.

The switch mode power supply is more complicated than the other supplies mentioned above. In this type of supply, AC voltage is converted to an unregulated DC voltage, with a series transistor and regulator. Since this DC is a constant high-frequency voltage, the transformer is significantly smaller and, consequently, so is the power supply. Transformers for a switching power supply, however, have to be custom made [11].

For the motor we chose, the STM17R-3NE NEMA 17, either a regulated or unregulated power supply can be used. One issue that could arise when using a regulated power supply—per the manufacturer—is it could cause a problem with regeneration. If the load is rapidly decelerated from a high speed, much of the kinetic energy of that load would be transferred back to the power supply, which could trip the overvoltage protection of the switching power supply, causing it to

shut down. Since we are not planning to reach very high speeds, this should not be an issue for our system. The manufacturer of the NEMA 17 motors offer power supplies that work well with their motors, such as the PS150A24 24VDC switching power supply seen in Figure 18 below.



Product Features

- 150 watts, 6.3 amps
- Universal input voltage
- Regulated output voltage
- For use with DC-powered stepper drives, servo drives, and integrated motors.



Figure 18: PS150A24 24VDC Switching Power Supply, courtesy Applied Motion

One of the biggest reasons for choosing a power supply that provides the 150 watts is because that would allow the project to be expanded in the future. The current system at the Robinson Observatory requires more power than our model system will but investing in the power supply at the beginning of the process makes it possible to broaden the scope over the different teams. There are many other reasons for choosing this power supply rather than building our own or purchasing another more inexpensive power supply. Designing and building our own power supply of this magnitude would be a complex project.

Due to time constraints, it is more sensible to purchase one with the required capabilities so that we can focus on the design and implementation of the system, which is more important than the power supply, which is only a small piece of the model. The power supply we've selected is not the cheapest we could find, but it also is not the most expensive. It is more expensive than we would prefer because of budget constraints; however, if the scope is to be expanded in the future, the initial investment is necessary.

More inexpensive power supplies could potentially provide the same load capabilities, but since the supply we chose has been designed to be used with the manufacturer's integrated motors, integration will be easier. A table comparison of options for the power supply used can be seen below in Table 4.

Table 4: Comparison of Power Supply

	PS150A24	PS50A24	PS320A48	Other
Recommended	YES	YES	YES	NO
Current Rating	6.3A	2.1A	6.7A	Variable
Voltage Output	24VDC	24VDC	48VDC	Variable
Watt Rating	150W	50W	320W	Variable
Cost	\$172.00	N/A	\$262.00	Variable

The main reason for choosing a power supply that was already designed was because the manufacture for the motor that was selected provided recommended power supply's that are compatible with the motor. Since there are two motors, the 2A power supply would be maxing out our consumption, and the other power supply that provided 320W was just more than we needed for the cost.

3.3.3 Sensors

Sensors are a key component of the original, full-scale implementation of the telescope and mount. In the telescope housed at the observatory, a pair of optical sensors have been selected for this purpose. Their functionality is two-fold.

First, it is possible for the motors to drive the mount in such a way that it could damage itself through further application of torque. Therefore, the first, and arguably most important, function of the sensor is to serve as a limit switch. When this sensor determines that the telescope mount has met a predetermined limit of movement, it engages and prevents any further movement in that direction.

The second purpose of the sensors are to detect when the telescope is in its "home" position. That is, a specific position for both sections of the mount had been designated as the home position. When the telescope is at rest (i.e. non-operational), it should be returned to this home position. Furthermore, calculations on position and tracking are dependent on allowing the telescope to start at this home position.

The existing configuration of the telescope relies on a single optical sensor for each motor. This optical sensor is then paired with an interrupter, which has been mounted to the gear. The direction and rotation of the motor is coupled with the way that the interrupter triggers the optical sensor to determine whether the telescope mount has reached its limit (i.e. limit switch application) or whether the

mount is in the home position (i.e. home position application). Figure 19 below summarizes this relationship.

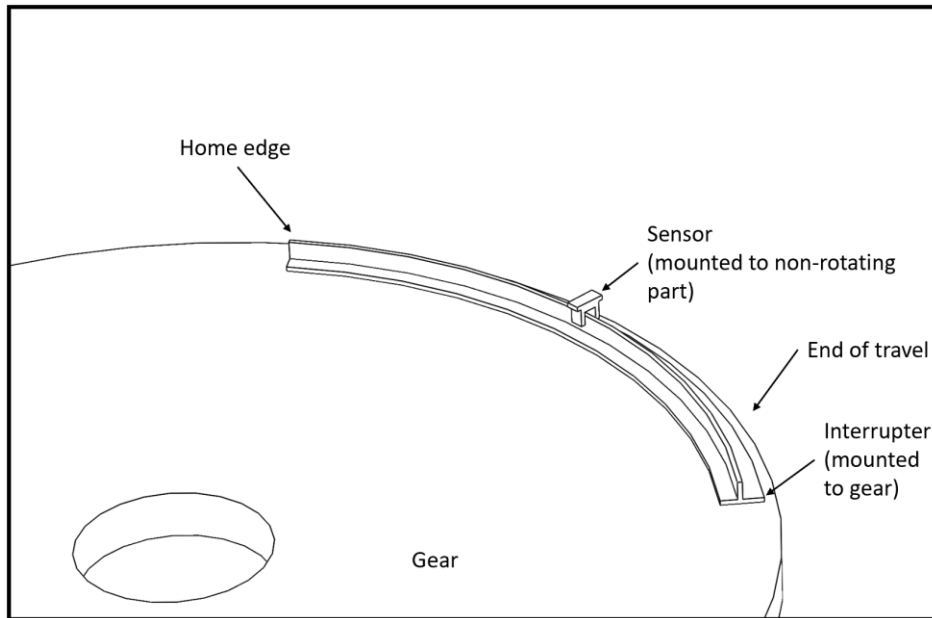


Figure 19: Existing Configuration of Optical Switch and Interrupter

There are a wide variety of position sensors available on the market, and we have considered each within the scope of our application. A brief treatment of each type is provided below.

3.3.3.1 Potentiometric Position Sensor

The potentiometer is a familiar tool to any student of electrical engineering. In brief, the potentiometer can be considered as a variable resistor that functions through the principle of the voltage divider. A potentiometric position sensor relies on a potentiometer to perform its duties. In this device, a resistive track serves as the sensing element. An element known as a wiper is attached to the part that is to be tracked, and the wiper also connects with the track. Movement of this wiper varies the resistance between one end of the track and the wiper, allowing it to serve as a potentiometer. One advantage of this type of sensor is that the resistance change with respect to wiper position is linear, facilitating ease of use. These sensors lend themselves best to applications that require linear position sensing, such as flow control valve position, injection molding and robotic motion control. Figure 20 below represents a typical linear potentiometer. Other options for potentiometric position sensors include cable actuated position sensors and rotary displacement position sensors.



Figure 20: Linear Potentiometer

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

3.3.3.2 Capacitive Position Sensor

As can be inferred from the name, capacitive position sensors use capacitance to determine position. These position sensors are highly accurate, generally offering a resolution as low as the nanometer level. The major drawback of these devices is that they are made to operate across a short distance, typically between $10\mu\text{m}$ through 2mm.

These devices are capable of operating in a non-contact environment through application on a conductive, grounded target. These sensors lend themselves particularly well to applications requiring high resolution, such as vibration measurement, semiconductor wafer surface sensing and servo system feedback for nano positioners.

In general, these devices operate through two mechanisms. The first mechanism is that of a changing dielectric constant. In this instance, displacement is measured by connecting the body to be measured to a dielectric material between two plates. The movement of the body will vary the dielectric constant, which is then measured. The second mechanism is that of changing the overlapping area. This is accomplished by connecting the body to be measured to one of the plates, while the other remains at a fixed position. The overlap between the plates changes with movement, thus changing the capacitance of the sensor. Figure 21 below represents a typical capacitive position sensor.



Figure 21: Capacitive Position Sensor

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported license from commons.wikimedia.org

3.3.3.3 Magnetostrictive Position Sensor

Magnetostriction is a property inherent to ferromagnetic materials (e.g. iron, nickel and cobalt). When these ferromagnetic materials are placed within a magnetic field, they change their size. A magnetostrictive position sensor operates on this principal.

The magnetostrictive position sensor is comprised of a position magnet, a magnetostrictive position sensor (used to measure the distance between the head of a sensing rod and the position magnet), electronics capable of transmitting a pulse, and a waveguide. The pulse is transmitted down the waveguide, and the induced magnetic field interacts with the magnetic field from the position magnet. This creates strain on the waveguide, which can then be sensed. These sensors also produce a high degree of resolution (on the μm scale) and can measure significantly greater displacements than the capacitive position sensor. The obvious drawback is the complexity of the sensor. Figure 22 below represents the operating principle of many magnetostrictive position sensors.

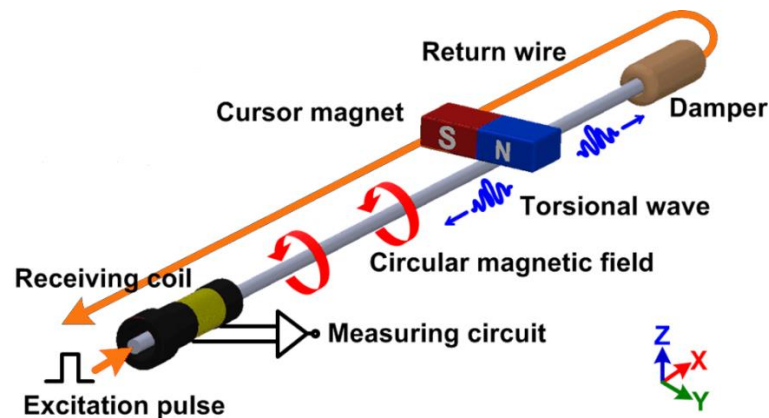


Figure 22: Magnetostrictive Position Sensors

Permission is granted to copy, distribute and/or modify this document under the terms of the Creative Commons Attribution-Share Alike 3.0 Unported license from commons.wikimedia.org

3.3.3.4 Eddy Current-Based Position Sensor

The Eddy-Current based position sensor is another non-contact position sensor that is capable of providing high resolution. These sensors are sometimes also known as inductive sensors, although this is a bit of a misnomer (generally, “inductive” is a reference to an inexpensive proximity switch). These devices are well suited to industrial applications due to a high tolerance for dirty environments.

In brief, these sensors also operate through magnetic fields. A driver creates an AC current through a sensing coil at the end of a probe. This induces an alternating magnetic field, which will induce small eddy currents in the target

material. This interaction changes with distance from the target material. As the distance varies, the Eddy-Current position sensor outputs a voltage proportional to the change in distance between a probe and the target.

3.3.3.5 Hall Effect-Based Magnetic Position Sensors

These sensors operate on the Hall Effect. That is, when the magnetic flux around the sensor exceeds a certain density, the sensor activates generates an output called the Hall Voltage. With these devices, the moving part is connected to a magnet, which is housed with a sensor shaft. This creates the Hall element. Then, when the part moves, the magnet also moves, which creates a magnetic field and induces the Hall Voltage.

The advantage of these devices lies in their reliability, small size, wide range of operating voltages, large variety of output options and relatively easy implementation.

3.3.3.6 Optical Position Sensor

The optical position sensor was the design choice made for the full-scale telescope currently operating in the observatory. The operation of this type of sensor was outlined briefly in the preceding section but is elaborated briefly here. There are two main types of optical position sensors. In the first, light is transmitted from one end to the other and changes in the characteristics of the light (e.g. wavelength, intensity, phase and polarization) are monitored.

In the second case, the transmitted light is reflected and then monitored. Optical sensors are also often referred to as encoders. Unsurprisingly, these devices operate particularly well when counting revolutions. The drawback with this type of device is that foreign particles (e.g. dust, dirt, water, etc.) can interfere and cause the sensor to fail. A common optical position sensor is represented by Figure 23 below.



Figure 23: Optical Position Sensor (Switch)

Ultimately, our team has selected the optical sensor for this implementation. There were several primary reasons for this selection, not the least of which is that the optical sensor closely matches the original equipment used in the full-scale telescope.

Table 5 lists the three strongest options for choice of sensor and shows the differences across a few electrical characteristics. Other considerations that led to our selection included cost, ease of implementation, size and the suitability of the technology for our task. The highlighted column shows which sensor was ultimately selected.

Table 5: Comparison of Sensors

	Optical	Hall Effect	Magneto-resistive
Supply Voltage	4.5 – 16V	5 V	3 – 5 V
Supply Current	12 mA	6.5 mA	16 mA
High-level Output	$V_{DD} - 2.1 V$	$V_{DD} - 0.5 V$	N/A
Interface	Digital I/O	I ² C	SSC / IIF
Manufacturer	TT Electronics	ams	Infineon
Part Number	OPB980T51Z	AS5601	TLI5012B
Cost	\$5.04	\$3.49 + Magnet	\$7.23 + Magnet

The final selection was a pair of TT Electronics OPB980T51Z optical switches. Several factors led to the selection of this specific part. First, this sensor is designed to operate across a broad range of supply voltages (VCC), ranging from 4.5V to 16V. Our ideal use case is to power these sensors directly from the Arduino, thus avoiding the need to tap an external power supply. Since the Arduino is generally configured to provide 5V power, we expect these sensors to operate normally given this condition.

The power dissipation is reasonable, rated at 300mW. A low-level output voltage of 0.4V will confirm to the Arduino's specifications. And, of course, availability is a concern – there are several thousand in stock across various distributors (e.g. Mouser, Digikey, etc.). Our selected sensor configuration includes covered apertures (to enhance robustness), flying leads (to ease integration with the PCB) and a buffer logic/Totem-Pole output driver architecture, as represented below by Figure 24.

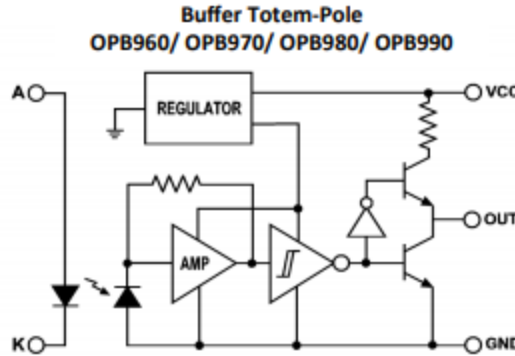


Figure 24: Buffer Logic/Totem-Pole Output Drive Architecture, Courtesy Texas Instruments

Our intention was to simulate the functionality of the full-scale telescope by using one optical switch per motor. We used feedback from the motor encoders to determine position and correlated this with the input from the optical sensor to determine whether the mount has reached its limit or its home position. An incidental benefit of this style of control was the reduced PCB footprint as compared to that needed for two sensors per motor.

3.3.4 Joystick

To control the movement of the motors manually, the project needed a joystick to serve as an additional input to the system. Although the main method of sending commands to the motors came from the Arduino, it was also desired to use a joystick for demonstration purposes as well as to best resemble the telescope found in Robinson Observatory. The main thing considered with the joystick implementation was that since the Arduino reads the output as an analog signal anywhere from an integer value from 0 to 1023, the rest position will vary each time around the 500 range. It is important to note that the joystick should only be used for rough pointing, but when testing the pointing accuracy of our system as a whole, the software should be the main source of pointing commands.

The options and technologies considered for joysticks are broken up into sections below with a summary table at the end of this section.

3.3.4.1 Mini Analog Joystick - 10K Potentiometers

This joystick features two 10k Ω potentiometers as well as a spring back system for tactile feedback. Most joysticks on the market that look like this arcade-style joystick feature only clicking switches in each direction. That is, when the joystick is moved in the positive x-axis direction, a limit switch is triggered, and the pin is set to high. In this joystick however, since there are potentiometers involved, the movement along the x-axis provides a range of varying output from -5V to 5V. The physical limits along both the x and y axes are 50 degrees, from -25 degrees to 25

degrees. The leads for the two potentiometers are easily accessible as shown in the figure below.



Figure 25: Mini Analog Joystick

Another benefit to using this joystick is that since it features two independent potentiometers, the gantry spring system in the interior of the joystick base allows for movement along the diagonal axes which is beneficial since it best resembles the existing system of the Robinson Observatory controls.

3.3.4.2 Two-Axis Joystick

The joystick model below is similar to the mini analog joystick above; however, it is sized significantly smaller and is already mounted to a circuit board. The pins shown in Figure 26 below stick out of the bottom of the board so that it can easily be inserted into a breadboard for prototyping. The joystick features two 10k Ω potentiometers and the option to wire the outputs as either voltage or resistance outputs.



Figure 26: 2-Axis Joystick

3.3.4.3 Analog 2-axis Thumb Joystick with Select Button and Breakout Board

This type of joystick is different than previous versions because it features a digital output in addition to the two analog outputs for motor control. The benefit of having a digital output is its ability to be programmed as a trigger key to automatically have the telescope perform a key position, such as the home position. In the Robinson Observatory, the homing of the telescope had to be done via software however, having another way to home the telescope could indeed be useful. Alternatively, the digital output could be used to effectively switch over control from the software program to the joystick, which was the chosen way in which the digital output ended up being programmed in our project.

Similar to the mini analog joystick, this version still requires a ground connection, a V_{CC} source of 5V, and two analog pins from the microcontroller. Additionally, it requires a digital pin for the select function. Figure 27 shows where these five pins are located on the breakout board that comes with this joystick.



Figure 27: Thumb Joystick with Select Button

Additionally, this joystick model is not a switching arcade style of joystick and features two potentiometers of a 10k Ω resistance value, similar to the other joystick considered. Another advantage of using this joystick is that it is ubiquitously used in multiple projects found during research, and therefore already has existing EagleCAD schematics and can therefore be directly incorporated into the PCB that we design. However, as purchased, it is able to be used as a standalone feature.

In summary, the pros and cons of each joystick are easily enumerated in the comparison table below.

Table 6: Comparison of Joystick Models

Feature	Mini Analog Joystick	2-Axis Joystick	Thumb Joystick with Select Button
Potentiometer resistance	10k Ω	10k Ω	10k Ω
Select button	No	No	Yes
Analog outputs	2	2	2
Digital outputs	0	0	1
Maximum operating voltage	5V	10V	5V
Breakout board included	No	Yes	Yes
Size	2.7 x 2.1 x 2.1	1.64 x 1.2 x 1.1	1.25 x 1.5 x 1.5
Price	\$19.95	\$6.95	\$5.95

Looking at the table above, and after thorough research, it was concluded that the joystick that was used in this project was the thumb joystick with the select button. This was chosen firstly because of the added feature of a select button which none of the other joysticks had. Also, it had a preferable breakout board which makes it easy to wire up without the need to mount it into a breadboard and can instead be directly wired through our PCB, however we opted to use the breakout board for portability reasons. It is also RoHs compliant, which met our environmental constraints outlined in section 4.2. A last reason for the selection of this joystick is that it is almost a quarter of the price of the larger joystick which allowed for the ordering of more spares to stay within the same budget. Meeting our budget requirements was one of our economic constraints set upon ourselves.

3.3.5 Microcontroller

The software design that was selected was highly based upon the microcontroller chosen in this section. The particular microcontroller we chose was the Arduino Mega 2560 with 54 Digital I/O pins (15 being PWM capable), 16 analog input pins and a standard set of power pins. This particular brand of microcontrollers uses their own version of the C language and is primarily handled completely by Arduinos own brand-built IDE. We also had to factor which signals the motor expects from our board and how to control that from our board. From these restrictions, it was required that we wrote our code and logic in the modified C language of the Arduino and therefore used its very specific IDE to upload and compile our files. These files need to be .ino files so it's very restrictive of what you can use to program.

3.3.5.1 MSP Option

To arrive at the decision of which microcontroller to use, we first started with a couple of different options on the table. The main microcontrollers we looked at were MSP, Arduino, and Raspberry Pi. Texas Instruments develops the MSP series of controllers and boards that are fairly basic and utilize UART connection and terminal to send and receive data in a serial fashion. Sending data via serial communication is one of two main ways to send data, with the other one being parallel. UART is a fairly simple set up and easy to use so there isn't going to be much restriction there in terms of what type of connection to the computer we use. It also exists on a plethora of boards out on the market. The hard part about MSP is the complex amount of work needed to be put in to use and setup the MSP. Along with its not so friendly programming practices we could see a lot of time sitting around and debugging the software, trying to make it work. MSP's generally use C, which is easiest for Electrical Engineers, but we decided that there were better options out there.

3.3.5.2 Raspberry Pi Option

The Raspberry Pi was the next choice in our list of possible microcontroller/board combination to use. The Raspberry Pi is fairly simple to set up and even comes with its own OS which is Linux based. This is all great but is not exactly what we want or need for the project. Its default language is also Python and being that none of our team members are super proficient in writing Python, it's probably best to leave that one out and look for other microcontrollers. The only reason why we would have chosen a Raspberry Pi would have been if the entire interdisciplinary team decided to use ROS, a robot operating system. The ROS architecture could have been pretty useful for the project as there are multiple components talking to each other and relying upon data being sent such as motors reaching a maximum turn, the need for a meridian flip, LED's and the feedback being sent to the computer science team. Along with a potential camera that can have openCV on it, there would be multiple files of different languages operating and at the same time, and ROS would have been a really good organizational system to use. However, the teams concluded early on that ROS wouldn't be necessary to use, therefore the Raspberry Pi was eliminated as a potential microcontroller option.

3.3.5.3 Arduino Option

The final entry on the preliminary list was the Arduino, which is precisely the one we chose to use. The Arduino is written in C and comes with its own premade IDE that is extremely easy to use. It has its built-in compiler as well as an uploader. The Arduino product made the process of outputting any desired code or signals extremely easy and quick. This is a great advantage because it sped up testing and debugging time and saved a lot of potential time troubleshooting possible things that could go wrong with the overall design. In terms of extreme flexibility in

what you can do, the Arduino probably has the most restrictions out of all of them, limiting our ability to really take advantage of the hardware. However, even with these restrictions it still ended up fitting our needs perfectly.

The serial connection was very easy to set up and maintain from a programming standpoint, while also making the programming much more streamlined with its built-in functions. Arduino microcontrollers/boards also have several material advantages that we can utilize. Arduino offers a particular product called an Arduino Mega that offers a bolstering 54 digital I/O pins. Our project had a profuse number of devices that needed to be sending input and output signals back and forth with the board and through the USB connection. The controller has pins that have a built in analog to digital converter (ADC) which is a sub-controller that takes any analog signal between a restricted 0 and 5 volts and converts them into discrete digital values ranging from 0 to 1023 (1024 values) as seen in Figure 28.

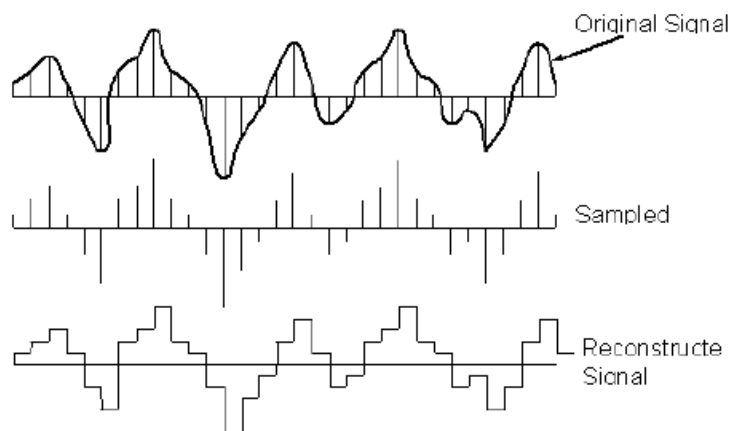


Figure 28: Signal Sampling and Digital reconstruction

These values are then programmable in code. Obviously, the original signal isn't perfectly reflected in the sampled signal as you can see in the image above but given a high enough bit resolution it can effectively be the same, in terms of what applications we ended up using it for. It so happened that our telescope also incorporated a joystick that a user can use to control the direction of the telescope. This joystick outputs an analog signal so we ended up using the analog to digital pin on the Arduino.

A final positive for Arduino was the built-in ability to increase the turn on voltage from not using the USB. The USB limits the starting voltage to 5 volts per USB standard, but there is a port that allows us to go up to 20 volts. Our project is definitely going to need it because of how many appliances that are going to be powered from the board. If we were to just use a USB 5 volt, the board would start struggling to stay on and provide the required power. If we perhaps used another board that only had a dedicated pin to power the board such as the raspberry pi, a transformer, rectifier, and voltage regulator circuit would have to be made. Otherwise the circuit would not be able to plug into a wall outlet in the United

States. The United states have power outlets running at 120 Volts AC, and if that was fed into a microcontroller board a lot of magic smoke would start to appear as the board is catching on fire. From this we decided that our best course of action would be use 9 volts input, as the recommended input voltage is between 7 and 12 volts.

In Table 7 below, a summary of each of the microcontrollers considered for this project is listed. As explained previously, the ATmega2560 was selected for the main microcontroller of the project due to the larger amount of PWM and total pins, and also due to the customer's request. To meet the requirements of UCF for substantial PCB design, also used an ATmega328 on board the PCB to check the functionality of the main microcontroller.

Table 7: Comparison Table for Microcontrollers

	PWM pins	I/O pins	Total pins	Primary Programming Language	Clock speed	Price
MSP430G2	~	24	24	C/C++	16MHz	\$17-23.75
Raspberry PI 3	~	40	40	Python	900MHz	\$35.68
ATmega328	6	14	20	C/C++	16MHz	\$22
ATmega2560	14	54	70	C/C++	16MHz	\$38.50

3.3.5.4 Programming Language

As touched on a little above, our choice as to what language we chose mattered greatly on what microcontroller we chose. However, programming languages and compilers today are so robust and advanced that a team could chose practically any language they wanted and still be able to get a functioning microcontroller. An obvious comparison would be that the Raspberry Pi uses Python as its primary programming language, even when Python is known to be very high level and not offer much control over the physical level of programming. That being said, the Raspberry Pi and Python language allow you to intake serial commands as well as pin registration, employing some low-level work. Yet, the base C language still remains the go to and best programming language for low level programming projects. It's definitely the fastest language out there (unless the assembly languages are accounted for) and has pretty much every single low-level capability that could be asked for. One of the big reasons for this is because you can directly optimize the space used for the program and work at a very base level from what the hardware is doing naturally. A Python compiler must go through an abundance of steps to end up producing a large machine code file that gets upload to the microcontroller, whereas a C compiler is almost just a syntax rewrite into assembly code and then easily assembled into machine code.

3.3.5.5 Integrated Development Environment

Most popular microcontrollers tend to have an IDE that's associated with them. In this case, the Raspberry Pi doesn't have one associated with it. However, there were three IDE's that came up as the first choices for our microcontrollers, two of which are the default ones and one is an open source IDE. The MSP series of controllers typically used an IDE called Code Composer which is actually based upon another open source called Eclipse. Code Composer featured a large set of tools for the programmer which included a debugger, a built-in console, a file management system, and exporter. It had a huge set of tools and features giving the user an endless amount of options. Albeit these options, it actually seemed to be too many options for us and too many ways for things to go wrong. Our team valued speed of application more than the immense list of possibilities, and it became clearer that we'd rather use an Arduino which contains a much simpler and streamlined passageway to get to our goal.

Energia is the other IDE that could be used for both MSP and Arduino; however, it was too similar to Arduino's custom-made IDE, and would just involve extra steps to set up. As a matter of fact, they are so similar, there is a lot of documentation comparing the use of Energia on an MSP430 to that of an Arduino. Even the layout and functionality of the two IDEs are remarkably alike. However, in terms of the speed and procedure of getting things working Arduino still takes the win with having effectively no installation process. That and a more user-friendly architecture regarding the actual practice of programming on it made Arduino a better choice than Energia.

3.4 Parts Selection Summary

As a result of the strategic part selection process, the main components for the design were collected by the end of the Senior Design I semester. An image of the main components can be seen in Figure 29 below. The numbers in the image correspond to the numbered list on the following page.

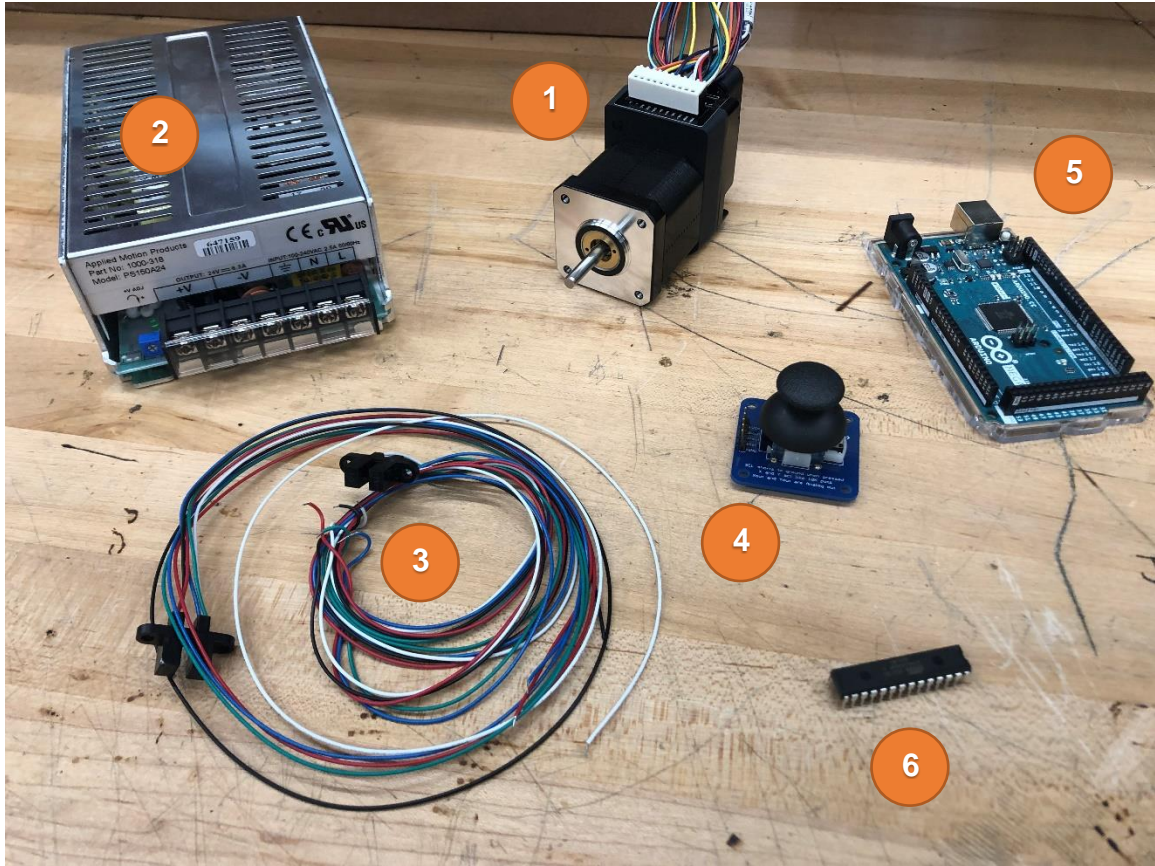


Figure 29: Major Components

1. Motor
2. Power Supply
3. Sensor
4. Joystick
5. Microcontroller (ATmega2560)
6. Microcontroller (ATmega328)

The following table provides an overview of what major components were selected for our scale model prototype. Specific details as to why each component were selected can be located in the strategic parts selection subsection found in Section 3.3. In the following section of this paper (Section 4) we will talk about how related standards and realistic design constraints affected the parts selection process. Constraints in the areas of cost, manufacturability, and environmental friendliness were considered in the selection process of the parts as well as which standards such as communication standards the relevant parts in Table 8 below use or do not use. We also considered some sustainability constraints in selecting parts such as if the desired parts were readily available and in stock.

Table 8: Parts Selection Overview

Item	Part Number	Manufacturer	Price
Motor	STM17R-3NE	Applied Motion Products	\$204.00
Power Supply	PS150A24	Applied Motion Products	\$172.00
Sensors	OPB980T51Z	TT Electronics/Optek Technology	\$5.04
Joystick	512	Adafruit	\$5.95
Microcontroller	ATmega2560	Amtel	\$12.21
Microcontroller	Atmega328	Amtel	\$3.21

4. Related Standards and Realistic Design Constraints

Having realistic design constraints and adhering to realistic standards is a very critical aspect of any deliverable product to a customer. In our case our customers include both our professors and the supporting personnel that work at FSI and Robinson Observatory. The following sub-sections outline the relevant standards used in our project as well as the design constraints that were considered throughout the part selection process following all the way through integration and testing.

4.1 Standards

This section outlines how ANSI Related Standards impact the design process of our project and lists the relevant communications standards used such as the very well-known USB standard. “Standards are documents that describe the important features of a product, service or system. For example, CSA Standard Z262.34-00 Ice Hockey Pucks specifies a hockey puck's material, size, mass, hardness at room temperature and test methods” [12]. The idea of standards really came about when it started hindering the safety of humans. ASME was in response to several steam boiler pressure vessel failures [13], and at some point, a collection of engineering institutes at the time came together to create an organization called ANSI. ANSI now watches over the creation of standards in the U.S., but also translates them to international standards so that they can be used worldwide.

4.1.1 ANSI Related Standards and their Design Impact

Standards can be easily interpreted as the backbone to modern technologies and lifestyles as they perform very supportive roles. They facilitate communication between designs without actually communicating data. This universal communication between designs takes time and effort to build which is where IEEE-SA, ANSI, NSSN, and various other organizations come in to develop the standards to be used. The IEEE-SA and ANSI aren't actually run by governments, but by communities that come together to create international or national standards to be adopted by governments, consumer groups, and the like. These standards aren't required by law to be met, but more a sort of checklist that allows products to be compatible across all types engineering. Essentially, by saying if something is up to a certain standard, the user knows that the product meets listed requirements and guidelines.

The University of Canterbury describes standards as “documented agreements containing technical specifications or other precise criteria to be used consistently

as rules, guidelines, or definitions of characteristics, to ensure that materials, products, processes and services are fit for their purpose” [14]. From the definition, standards can be used as rules, guidelines, or definitions, which illuminate what standards can be used for. Noting the ability to become rules, the engineering codes definition is then created. Engineering codes can be standards that if not adhered to can become punishable by law. For our particular project and product, there will not be any engineering codes we must adhere to, however we will be taking advantage of already existing codes implemented.

Our project had a set of prebuilt devices and products that were all incorporated together, which all came with their own set of standards. For connecting with the CS team’s software, a USB standard had to be taken into consideration as it was how we communicated between the computer and our Arduino Mega. The Arduino Mega has its own set of standards that it follows. It has its own built in USB and C based language which is then compiled and sent through the Arduino brand IDE.

4.1.2 USB Standards

The standards for the Universal Serial Bus were actually created with companies instead of just established standard organizations such as the ones listed above. It became a joint effort between Compaq, Intel, Microsoft, NEC, and a couple of others to create a port device that was easily compatible with a plethora of devices. The Universal Serial Bus is arguably the most prominent example of how standards help develop the world with the interoperability between devices as a common goal. It is the flagship of the digital world in terms of a standard that wants to be met by designers. The parties associated with creating this standard also released the document for free, detailing how developers should go about creating a USB device that is compatible with other USB devices.

The original motivation behind the USB 2.0 had three parts associated with it. The “connection of the personal computer to the telephone” begins as the first one on the list. During the starting age of the PC and communication systems, each were both being developed separately. The developers saw this and agreed that a cheap and universal solution would be needed to link the two realms of technology if the future generation of products wanted to be upgraded from the previous generation. The second motivation was the “Ease-of-use” idea. Computers used to be in a very niche market due to their high learning curve and lack of understanding from the public on what can and can’t be done with a computer. It was detrimental to the users if certain aspects of the computer they owned didn’t allow other devices to work together, thus raising the cost and required understanding to be able to reconfigure the computer to suit a user’s needs. Not only did this ease of use benefit the user and variety of the computer, but it also played a significant role in the marketing of computers as the accessibility for them opened up a large number of peripherals and spawned scores of products that could be bought. The third and formally final motivation is the “port expansion”.

Expanding on this means that instead of having two ports, one for input and one for output, a single USB port could be used for both in terms of connecting the device. This halves all the required ports needed on a computer, essentially doubling the number of connectable adaptors, furthering the market for peripherals. A fourth and quieter motivation is that as time moves on the PC began getting more powerful and capable of higher performance. So, a new, faster type of port needed to be developed. Conclusively, the developers put a very strong summation to what USB's meant for computer with the quote, "Thus, USB continues to be the answer to connectivity for the PC architecture".

From a physical side the port itself has specific requirements that must be met not only electrically but physically. Starting electrically, the USB 2.0 requires a 5 volt to ground connection with a maximum current at 1.5 Amps (Figure 30). This VCC doesn't provide any data and is actually specifically to be used to either supply or deliver power to a circuit. This ended up providing the power to our Arduino mega and is labeled pin 1 on the figure. The opposing pin 4 is the ground which everything will be referenced to. The USB provides 2 data pins called D+ and D-. These are serial pins that feed in data one-bit rate at a time in the form of a square wave.

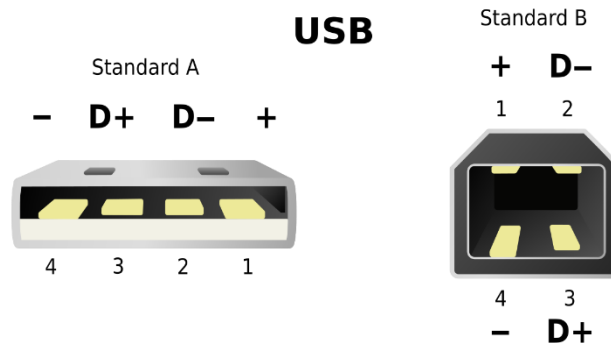


Figure 30: USB Description Graphic

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

4.1.3 C Standard

Before the invention and fabrication on modern compilers and computer languages, specified hardware-based assembly was the only way to program a computer to get it to perform the tasks desired of the programmer. However, as a giant reflection of the engineering world, each hardware and setup had their own specified version of assembly that needed to be used. And just as what happened in the engineering world, it became very difficult to properly share and conform certain products and projects from one to the other. Then the C language was made. The function and establishment of the C language is so similar to other standards from engineering where ANSI and ISO came together. They formed a

standard of C with employing the idea of “Software developers writing in C are encouraged to conform to the standards, as doing so helps portability between compilers” [15].

As it stands there are multiple different versions of the C standard as the years went on. They are aptly named based on the year they were created, such as C99 in 1999, and C11 in 2011. The current iteration of the C standard is C18 which was just released fairly recently and is the one our Arduino will be based on, as Arduino language is not actually C, just heavily inspired by it. C18 is extremely similar to C11 as it mainly just addresses certain defects in C11, which means no actual language changes.

One major change that could affect us is the removal of the `gets()` function in C11. This function reads in bytes from a standard in operation, but what known to be unsafe and cause a plethora of problems for programmers. The second major change in the language of C11 would be the addition of anonymous unions and structures that allow you to not identify the data type. However, unions and structures were not used at all with our program as these advanced data types didn't serve as much use to us. In the end we will unfortunately not be able to say our code is C18 standard compliant due to Arduinos special functions. Arduino's C based language comes with some extra functions for the programmer to use that invalidate our code from being portable to anything other than an Arduino microcontroller.

4.1.4 ATmega Standards

The Arduino brand of boards and microcontrollers is known and used worldwide. It is a marketed device that heavily is advantaged by the standards it uses. With it being this largely used device it obviously has standards that it must abide by. The user manual for Arduino design boards has a couple of notable listing of standards they use and follow. The JEDEC standard is a memory ram standard outputted by a committee that oversees semiconductor microelectronics, which AVR uses to store temporary memory. Another standard is the RS-485 standard which defines serial communications in electrical systems. Arduino moves by this standard in setting up its transmit and complete flag that can be found as a port on the Arduino Mega board and microcontroller, as well as in the USART. It is also known to be used in USB connections. A third standard AVR uses is the JTAG industry standard of “verifying designs and testing printed circuit boards after manufacture” [16]. This is mostly for quality assurance after the board is built to make sure the final product actually is the intended final product, which doesn't apply to the actual Chip AVR makes. However, the final one is closely tied to the JTAG standard as it is applied to quality assurance rather than a core design of the microcontroller and board. The 1149.1-2001 IEEE standard test access port and Boundary Scan Architecture standard just tests the integrated circuit for faulty silicon components and works in tandem with the JTAG testing.

The ATmega has a significant number of standards that it must adhere to before having the microcontroller and board go on the market to be purchased. Nonetheless, these standards don't quite apply to our specific project. We most certainly took advantage of them, but the standards themselves had very little to do with our design of the telescope.

4.1.5 NEMA ICS 16

This standard published by the National Electrical Manufacturers Association (NEMA) governed the design and implementation of our two NEMA-17 motors. The standard referenced for this document was last updated in 2001 and is available on the NEMA website [17]. Although there is a great deal of specifics covered by this standard, only the items most relevant to our design will be covered here.

The first conditions relevant to our design pertain to operating conditions for those motors. For example, the NEMA standard designates that the rated value of ambient temperature shall be 40°C (104°F) unless otherwise specified by the manufacturer. Additionally, these temperature ratings are based upon operation at altitudes of 1,000 meters (3,300) feet or less. In both cases, these constraints are well within the design parameters for our device and did not need to be considered.

There are a number of size and other mechanical considerations implemented in this standard. However, as the motor selection had been discussed and approved with our mechanical engineering counterparts, we verified that the NEMA specification was reasonable for our use case. Therefore, a detailed examination here is not needed. Similarly, there are a number of testing and acceptance criteria specified in the standard (e.g. dielectric withstand, step accuracy or insulation resistance). Again, since these tests are undertaken by the manufacturer, not the end user, these were not of specific concern to our design.

Of particular interest are the various specifications listed under the controls section of the standard. First, we consider the range of operating voltage and frequency. The standard allows a deviation of +/-10% for RMS input voltage and a frequency deviation of +/-2%. Since our power was provided by a supply sold by the manufacturer of the motors, these tolerances were well within spec and not a concern for our implementation.

The NEMA standard specifies the use of rotary encoders to serve as the position and velocity feedback device. Operating supply voltages can adhere to a range of values from 5V DC through 28V DC, but in the case of our motor implementation the manufacturer has selected 5V. Outputs can conform to the following options: line driver, TTL compatible, open collector, amplified sine wave and triangular

wave. Our device utilizes a line driver and, in addition, is TTL compatible. Additional sections specifically relevant to our device and application include specifications on quadrature error. Since our encoders operate on the quadrature principal, we see that a specified measurement is performed relative to the offset in quadrature from the expected 90° .

Indexing is accomplished via a square pulse. As an interesting aside, the standard also references indexing as a home position or zero reference. This was especially applicable to our design, as the index was used to confirm when the device has reached its home position. The index can be either gated or ungated. Our particular application employed an ungated index pulse – that is, a pulse that occurs once per revolution.

Finally, there is a suite of information on testing the various outputs available with lab equipment. Again, this was a relevant section for our project as much of the testing was done in the lab using oscilloscopes and function generators (in addition to the waveforms generated by our controller – the Arduino). Testing procedures included items such as line count verification for encoders with index and testing the index using a two-input oscilloscope. Each procedure includes a step-by-step breakdown of the necessary test setup (e.g. step 1 – connect two $1\text{k}\Omega$ resistors in series to form a series resistor network, etc.).

4.1.6 Communication Interface Standards

When considering the different motor options, it was found that some of the STM17 models had RS-232 communication interface, some had RS-485 communication interface, and some had no communication port. Although we ended up choosing the motor model which did not make use of the RS-232 communication port, this communication standard is still important to discuss in the context of our project because of how ubiquitous it is in the Robinson Observatory on Ara Drive. In the observatory, the dome rotation controls, door controls, and positioning controls are all controlled through a computer and series of control boxes via USB and RS-232 interfaces. Understanding the RS-232 standard will be key for future team's progress towards replacing the control box in the observatory as well as ours so that we can try to mimic the communication protocol as close as possible in our scale model.

The recommended standard (RS-232) was first introduced in 1960 by the Electronic Industries Association (EIA) for data transmission via serial communication [18]. The standard gives recommendations for the electrical characteristics and the timing of how signals are sent and received. In addition, like many electrical standards there are also mechanical standards associated such as the physical size and pinout of the connectors. The female and male connectors are commonly seen on older equipment, particularly personal computers, printers, mice, power supplies and other computer peripherals. Since

the Observatory has been around for a long time, it would make sense why they still have peripheral equipment that is reliant on the RS-232 standard for serial communication.

As for the electrical characteristics of this standard, there is a maximum open-circuit voltage limit of 25 volts and the output signal level usually varies from +12V to -12V [18]. The data transmission limit is 256 kb/s and the cable line limit is 50 feet. As far as logic control, there are procedures to control the flow of information in both directions. The signals are represented as voltage levels with reference to a common ground or power in the system. Certain pins have certain functions like sending signals meaning 'Request to Send' and 'Clear to Send' which control data from the transmitting computer to the receiving data set. There are also confirmation pins indicating whether the data was transmitted, or received, and which data carrier to detect.

Some reasons this standard has become outdated are that it is limited to lower data transmission speeds and works best with shorter cable lengths. The connectors are also fairly large and take up a lot of space on the devices that use them. The USB and RS-422 standards which serve as the RS-232 successor in a way are further discussed in detail in their respective sub-sections in this chapter.

4.1.7 Industry Standard 26C31 Differential Line Driver and 26C32 Receiver

When considering the different motor options, one of the key differences between the top three considered was whether they had a built-in encoder or not. As seen in Section 3.3.1, the encoder option was selected. The incremental encoder specifications state that there is an internal differential line driver which can source and sink 20mA at TTL levels. The specifications recommend using the industry standard 26C32 as the receiver. The corresponding datasheet to the AM26C32 Quadruple Differential Line Receiver which was used to assist in digital data transmission lists that the receiver meets or exceeds the requirements of ANSI TIA/EIA-422-B, TIA/EIA-423-B, and ITU Recommendation V.10 and V.11.

The TIA/EIA-422 standard, which is also known as the RS-422 standard, is a technical standard which was created by the Electronic Industries Alliance (EIA) which specifies the required electrical characteristics of a digital signaling circuit. This standard imposes limits on the transmission rates of data and the length of cables that use this standard. The data transmission rate limit is 10 Mbit/s and the limit on cable length is 4,000 feet. Our cable length need in our application is well under that limit.

Though this specific standard only applies to the definition of signal levels, there are also other properties of the serial interface such as mechanical properties

applying to the connectors and properties of the pin layout and wiring. Those properties are part of the RS-449 and RS-530 standards.

As far as communications wiring, the RS-422 standard indicated that the wiring should be made of two sets of twisted pair cables where each cable pair is shielded. Shielded cables have the advantage of being more noise immune because of how they couple the noise better and more symmetrically than their non-twisted cable counterparts [19]. Compared to the RS-232 standard, the RS-422 standard overcomes the limitation of using single-ended standards. These limitations also include the limited data transmission rate as well as the lack of ample noise rejection capability.

The second standard that is applicable the differential line receiver we used was the TIA/EIA-423 standard, also known as RS-423. This standard is similar to RS-232 but features higher data transmission rates. Compared to the RS-422 standard one of the main differences is that it defines an unbalanced or single-ended interface with a sending driver which is unidirectional. A table comparing both RS-422 and RS-423 standard specifications is below [20].

Table 9: Comparison of Telecommunication Standards

Specification	RS-423 Standard	RS-422 Standard
Operating Mode	Single-ended	Differential
Maximum data transmission rate	100kb/s	10Mb/s
Max. Driver Output Voltage	+/- 6V	-0.25V to +6V
Driver Load Impedance	Less than 450 Ω	100 Ω
Slew Rate	Adjustable	N/A
Receiver Input Voltage Range	+/- 12V	-10V to +10V

The last standard which our differential receiver used is that of the International Telecommunication Union (ITU) which coordinates standards for telecommunications. There are a series of ITU recommendations established for data communication and two of them, specifically V.10 and V.11 are met or exceeded by our differential receiver. Recommendation V.10 states that for unbalanced electrical circuits the data communication can be up to 100 Kbit/s which is in agreement with the RS-423 standard. The recommendation under V.11 states that for balanced electrical circuits, the data communication can be up to 10 Mbit/s which is in agreement with the RS-422 standard.

4.2 Realistic Design Constraints

As mentioned previously, the realistic design constraints that were set upon our project had a big impact on how we went about choosing parts and integrating a working product. The design constraints used can be broken down into the following categories: economic constraints, time constraints, environmental constraints, social constraints, political constraints, ethical constraints, health constraints, safety constraints, manufacturability constraints, and sustainability constraints.

4.2.1 Economic and Time Constraints

Since our project was funded in part by the Florida Space Grant Consortium, there was an allotted \$1,000 that was generously provided and to be split between the three interdisciplinary teams. Anything outside of that had to be covered by the individual members. The procurement of some parts for the electrical team had begun before receiving formal funding as waiting for funding would have caused serious delays in the project progress. Our team did not wish to spend too much more over the potential allotment of \$500, considering that the computer science team did not need their portion of the grant money, therefore this served as an economic constraint towards our project. This economic constraint consequently affected the part selection process for our various subsystems. For example, there were options for more precise motors and encoder feedback but that came with a price. It was carefully considered for each of the subsystems whether the extra cost associated with alternative parts could not have been overcome via software or another method.

In addition, there were harsh time constraints on the project that were also considered during the project scheduling phase. All our members on our team were taking at least three engineering classes in addition to senior design. At least two members had part-time internships in addition to their University of Central Florida collegiate course load. Therefore, to maintain good grade point averages for the semester, there had to be a time limit on the total time dedicated towards this project. Another timing constraint we considered was that since this effort was an interdisciplinary project, our sponsors were not available for meetings all the time so there needed to be a comfortable amount of cushion time to allow our sponsors and significant outside contributors enough time to adequately address any concerns or questions that arose.

Another factor we had to consider was that since our teams' advisor was in the Mechanical and Aerospace Engineering department, any purchases made with FSI or FSGC funds had to be made through the MAE purchasing office. Although our team was required to receive all major components by the end of the semester according to Senior Design 1 rubric, in order to actually receive those parts on time, we had to submit appropriate order forms with a reasonable time allotment

for the purchasing office to place the order, receive the order, process the order, and for our purchasing lead to retrieve the order. Because of those reasons, any purchases made needed to consider this large time constraint when selecting parts and vendors.

Even though our group was one of the few groups that elected to take Senior Design II in the fall as opposed to the summer, our group still felt as if there would still be a significant time constraint felt throughout all semesters. This was because over the summer, all members of the team were pre-occupied with both in-state and out-of-state full time and part time internships. However, through the use of our milestone planning tools later discussed in the Administrative Content section, our group showed that we overcame the obstacle of limited time.

4.2.2 Environmental, Social and Political Constraints

The final product of the scale model telescope was aimed to be geared towards hobbyist and amateur astronomers use. Therefore, it was important to consider the social constraints of those who will end up using our product if it were to be re-produced at any volume. Having an easy-to-use user interface was one way to achieve this. In addition, building the scale model telescope system at a low cost would also adhere to the social constraint of consumer budget.

As for environmental constraints, it was our teams desire to use as many RoHS components as possible in our design. RoHS stands for Restriction of Hazardous Substances and is implemented under the RoHS Directive that restricts the use of certain hazardous substances under European Union transposed laws. To be compliant with the RoHS Directive, each component must have minimal percentages of concentration by weight of a handful of substances in any given material. These substances are Cadmium, Hexavalent Chromium, Lead, Mercury, Polybrominated Biphenyls, and Polybrominated Diphenyl ethers, or any of their compounds. The percentages of concentration must all fall under 0.10% for all substances except for Cadmium or Cadmium Compounds which have an even tighter restriction of 0.01% by weight [21].

4.2.3 Ethical, Health, and Safety Constraints

As discussed in the parts selection section, one of the components used in our system to verify pointing accuracy was a laser pointer. Different laser classes have different levels of safety to be considered. In the United States, most lasers that are used in astronomy and by the general public to point at the sky and certain constellations are Class 3a lasers which emit a green neodymium diode laser beam [22]. This laser beam has a wavelength of 532 nanometers and has an output power of just under 5 milliwatts [22]. Since this laser class is higher than a

Class 1 or Class 2 laser, it requires a warning label reading “danger” by the US Food and Drug Administration (FDA). As for safety, it is critical that the laser beam of this class is not pointed into anyone’s eye. Since higher class laser beams are much more concentrated than lower class laser beams, every precaution must be taken from a safety standpoint to avoid accidental pointing into someone’s eye. Precautions like constantly ensuring the laser pointer is off unless everyone is standing behind the horizon line limits of the telescope will ensure everyone’s safety. There should also be affixed appropriate IEC-Compliant laser safety labels placed on the outside of the laser. The chosen laser pointer used ended up being only a Class II, however the same precautions (aside from the safety labels) were taken during testing.

Another safety constraint was manufacturing a PCB that does not generate too much heat so that it would be unsafe to touch any voltage regulation components. A 15V linear voltage regulator for instance, has a maximum operating temperature of 125 degrees Celsius, or 257 degrees Fahrenheit. Temperatures of only about 80 degrees Celsius are needed to cause severe burns when touched for less than one second. Therefore, it was critical that any components that can reach that temperature when operating be either dissipated with a large enough heat sink or completely concealed and distanced on the PCB from any components that a user would interact with such as a joystick.

In addition to ensuring the components on the PCB did not burn anyone, the manner in which soldering was used posed another PCB related health and safety constraint on the project. For one, soldering irons can reach a temperature of 400 degrees Celsius so care was taken to never touch the tip of the soldering iron. If a tip needed to be replaced for one with more surface area, the replacement only occurred once the iron had reached a safe temperature. In addition to the burning hazard from the soldering iron tip, soldering certain materials heats them to a point where fumes can be inhaled by the person performing the soldering. Additionally, flux which is used to aid in the soldering process is known to contain rosin which also produces dangerous solder fumes [23]. It was important that the proper soldering conditions were used which included ventilation by means of a fan that could prevent the soldering fumes generated from being inhaled. The type of solder used was also a constraint on this project from a health and safety perspective. Solder which contains traces of lead is considered to be toxic, especially if ingested. Therefore, firstly the use of lead containing solder was avoided.

4.2.4 Manufacturability and Sustainability Constraints

For a product such as our senior design project to be manufacturable, it must be able to be easily reproduced. This included us trying to use commercial off-the-shelf parts when possible. Using commercial off-the-shelf parts with short lead

times and large stock quantities also helped our project meet sustainability constraints since it would be less likely that those parts will go obsolete in the near future. Another way we made our project more sustainable was to be robust in the design of the PCB so that components were attached professionally and securely. Also, any wires that came into contact with moving parts on the telescope were to be wired appropriately and with enough slack so they were not strained during any particular movement or combination of movements. Having optical sensors to determine when the telescope reaches the pointing limits helped prevent overstretching and straining of the wires leading to the motors from the power supply and control system at the base of the telescope.

5. Project Hardware and Software Design Details

This section on project hardware and software design details outlines the process at which the team designed each particular subsystem. For our complete design, we had a total of six subsystems, with one of the subsystems being broken down into a set of two systems due to the small size of the subsystems. The subsystems are; DC to DC converter, Sensors and LEDs, Motors, ATmega2560, Joystick, and the ATmega328.

5.1 Initial Design Architectures and Related Diagrams

There are many options when it comes to choosing a microcontroller for a project. A considerable amount of thought is required to choose the right microcontroller, such as how many and what type of hardware interfaces (communication and I/O), as this will dictate the number of pins and space is required of the microcontroller.

Software architecture is another important piece of the puzzle to consider. This will determine how heavy or light the processing requirements are. Cost and power constraints are also important factors to consider. With that being said, choosing to use an Arduino Mega 2560, appears to have satisfied all of our requirements needed in a microcontroller because it provided the functionality we needed for our design but also satisfied the requirements from the sponsor.

To also satisfy the requirements from UCF, an on-board microcontroller that performs simple functions was used. The on-board microcontroller of choice was the ATmega328. This microcontroller was added to the shield which attached to the ATmega2560 development board.

The ATmega2560 provided the ability to attach a shield to the existing board to satisfy our requirements of constructing a printed circuit board (PCB). A shield is modular circuit board that piggybacks onto the ATmega2560 to instill it with extra functionality as seen in Figure 31 below.

It was determined from our design that one shield will suffice, however, if future teams would like to add complexity to our system, there will be limited space on our shield. As we continued our design, we factored in that in the future, additional shields may be attached to the ATmega2560, so the routing of signals needed to be assessed.

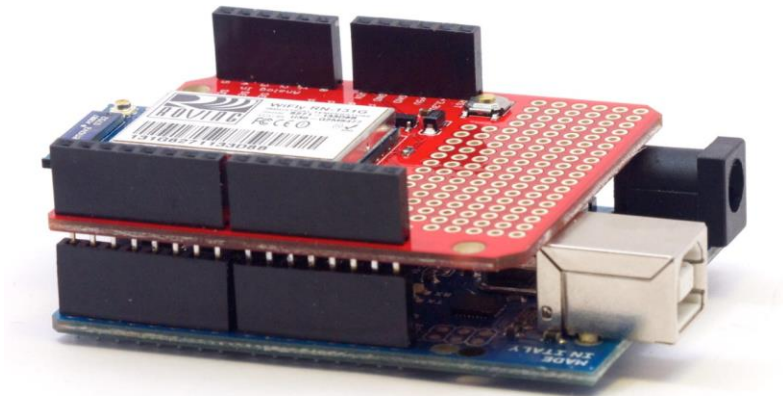


Figure 31: Stacked Arduino shield

Permission is granted to copy, distribute and/or modify this document under the terms of the Attribution-NonCommercial-ShareAlike 2.0 Generic license from creativecommons.org

As far as the structure and design of the system is concerned, the team made up of Electrical Engineering students, Computer Science students, and Mechanical Engineering students was mostly concerned with providing a more accurate model of the current system that is in place at the Robinson Observatory. Because of the added features, such as the focus, dome control, and shutter control, a complete replication of the system was unattainable for a single senior design team to complete in a two-semester project. The goal was to foresee some of the changes required in the future two-semester projects to aid the future senior design teams in their design without requiring them to completely redesign the system.

By choosing a microcontroller with many extra inputs, this will provide the future teams with the ability to add additional inputs and outputs to the system and easily alter the software of the system using the Arduino IDE. Also, by ensuring that the designed system provided an ample amount of power will also give the teams enough room to increase the power consumption of the system without fear of overconsumption.

Our plan was to purchase a power supply that was capable of driving the two motors (right ascension and declination) and the rest of the system. The output of this power supply unit was then to be split to the motors and we would use a DC to DC converter to provide all the lower power electronics with a source that is more than able to handle any load required. However, after realizing that we would have to have the Arduino Mega plugged into the computer USB port to communicate with Stellarium, we decided to skip the implementation of the DC converter and just use the Arduino's built-in 3.3V and 5V power pins to power our sensors, joystick, and lower power electronics.

5.2 First Subsystem, Breadboard Test, and Schematics

For the first subsystem of the project which was designed to be a DC to DC Converter, the following two sections describe the schematic layout of the subsystem, factors that went into designing the subsystem, any tools used to assist in the design of the subsystem (such as TI's Webench Power Designer), as well as how to verify the subsystem design was working as expected via breadboard testing. Though we ended up not electing to use a DC to DC converter as a subsystem in the end, the design is applicable should future teams need to power additional low voltage electronic components from the power supply.

5.2.1 DC to DC Converter Design and Schematic

Starting with the DC to DC converter for the system, a schematic layout of the design can be seen in Figure 32 below.

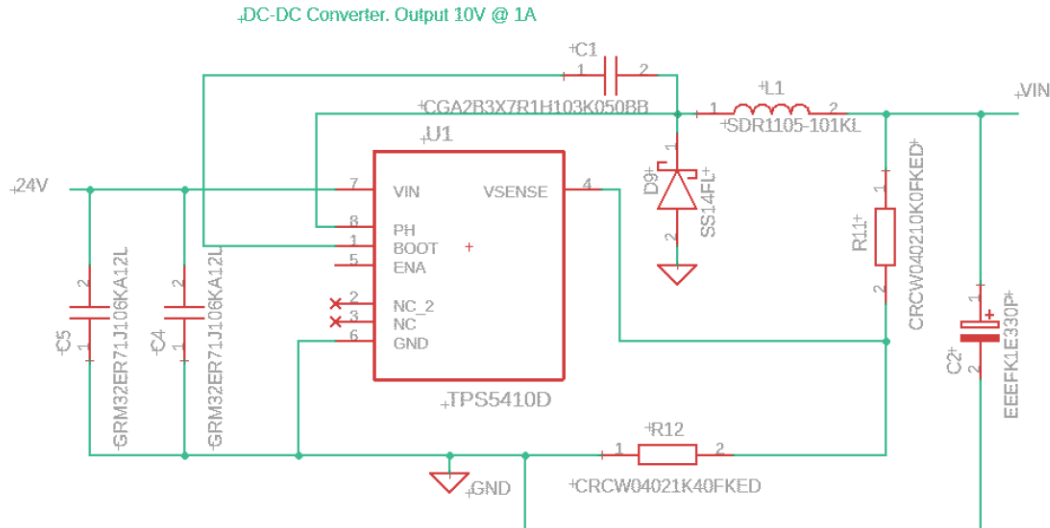


Figure 32: DC to DC converter Schematic

As one can see, the input voltage to the DC to DC converter was intended to be 24VDC. This initial voltage would have been from the PSU (PS150A24 24VDC switching power supply) that was purchased to supply power to this DC to DC converter for the operation of the ATmega2560, and all of the other lower power electronics in addition to the motors being used to control the telescope. The motors did not receive their power through this DC to DC converter because the V+ of the motor requires 24VDC, but they would have received their control signals

for steps and rotation through the signals from the ATmega2560 that would have received its power from the DC to DC converter. Though as mentioned previously, this design was not implemented on our PCB, and the control signals received their power from the existing power pins on the ArduinoMega2560 dev board beneath our shield.

When we designed this circuit, Texas Instruments (TI) provided a power designer tool, Webench, that aided in the design of an end-to-end power supply. Figure 33 below shows the website and the parameters required for the end-to-end design.

The screenshot displays the Webench Power Designer interface, divided into three main sections:

- Input:**
 - Supply type is: DC (selected), AC
 - Vin Min *: 21.6 V (range: 0 - 1000)
 - Vin Max *: 26.4 V (range: 0 - 1000)
 - Advanced:
 - Vin Nominal: 24 V (range: 0 - 26.4)
 - Add an Input EMI Filter:
- Output:**
 - Vout *: 10 V (range: -80 - 500)
 - Iout Max *: 1 A (range: 0 - 180)
 - Isolated Output:
 - Advanced:
- Design Consideration:**
 - I want my design to be:
 - Balanced (selected)
 - Low Cost
 - High Efficiency
 - Small Footprint

Figure 33: Webench Power Designer, courtesy Texas Instruments

The acceptable range for the ATmega2560 via the V_{in} pin is 7-12VDC. If the V_{in} is less than 7V, the 5V output pin on the ATmega2560 may supply less than 5V and become unstable, but if supplied with more than 12V, the voltage regulator may overheat and damage the board. To eliminate the possibility of either issue, a V_{in} of 10V was chosen. After reviewing the PSU datasheet, the minimum, maximum, and nominal output voltage was obtained, which provided the necessary parameters for the power designer tool from TI. With all of the required information, Webench was able to output multiple DC to DC converters that met our requirements. When narrowing down the selection of converters, there were many things to consider: efficiency, BOM cost, footprint, BOM count, component footprints in Eagle. The efficiency of the converter tells us how close to the 10V output at 1A we will be. This is important, but the majority of the designs provided an efficiency of 88% or more, so that still did not narrow down the search much.

The BOM count and footprint was an important consideration because our shield has a limited area and Eagle, the PCB software we are using, has a free version that only allows PCBs to be of a certain size. Also, the more components, that is the higher BOM count, the more items there will be to solder onto the PCB, which increases the probability of board issues. The BOM cost is not much of a concern because the DC to DC converter we are needing would not require costly components. The components that are used for this low-level power are quite cheap to manufacture. The BOM costs in consideration were under \$2.00.

Lastly, the footprints in Eagle played a major role in the selection of a DC to DC converter. When designing a PCB, the physical layout of the devices must be known for the layout of the board. Although our combined experience using PCB design software was limited, we realized that finding parts that have existing footprints are hard to come by in most cases. So, before narrowing down our design decision, the individual parts, mostly the regulator were researched to ensure that the footprint was available for those parts. Footprints can be constructed by the user, but that can add a level of complexity and uncertainty that does not seem to be worth the risk since there are many options for DC to DC converters.

5.2.2 DC to DC Converter Breadboard Test

The breadboard testing for the DC to DC to converter was quite straightforward. Since TI's Webench program provided the design for the team, any error in design should be eliminated. To test the converter, a 24V DC signal was applied to the input of the converter by means of a DC Power Supply and the output voltage was checked to ensure a 10V DC signal at the output by using a Digital Multimeter (DMM). In addition to verifying the system outputs a proper voltage, the current also had to be checked. The current capabilities of the DC to DC converter was designed to handle 1A. To test this, there had to be a load attached to the output of the converter and the current along with the output voltage were tested.

This subsystem could be easily tested by itself because of the general capabilities a DC to DC converter needs to operate as intended. In the next subsystem, the LEDs and sensors were tested with and without the DC to DC converter to not only verify the design of the subsystem was correct, but also that the subsystem integrated well with the DC to DC converter subsystem.

5.3 Second Subsystem, Breadboard Test, and Schematics

For the second subsystem of the project which was composed of both the status LEDs and sensors, the following two sections describe the schematic layout of the subsystem, factors that went into designing the subsystem, any tools used to

assist in the design of the subsystem, as well as how we verified the subsystem design was working as expected via breadboard testing.

5.3.1 Status LEDs Design and Schematic

The second subsystem designed for our system were status LEDs. Status LEDs was an idea developed by our team as a way to show the user the current status of many of the devices within the system. When speaking with the team that operates the telescope, it seemed as if the team was confused as to the current status of the telescope because of the lack of feedback from the system. The current system at the Robinson Observatory has beepers to beep when the motors were ready, but that was all that it offered. In addition to keeping the user informed on the operation of the system, adding other LEDs for other purposes could assist in diagnosing issues, or aid in testing of the system. The LED system we designed can be seen below in Figure 34 and Figure 35. Four LEDs were tied to the ATmega2560 and three were tied to the ATmega328.

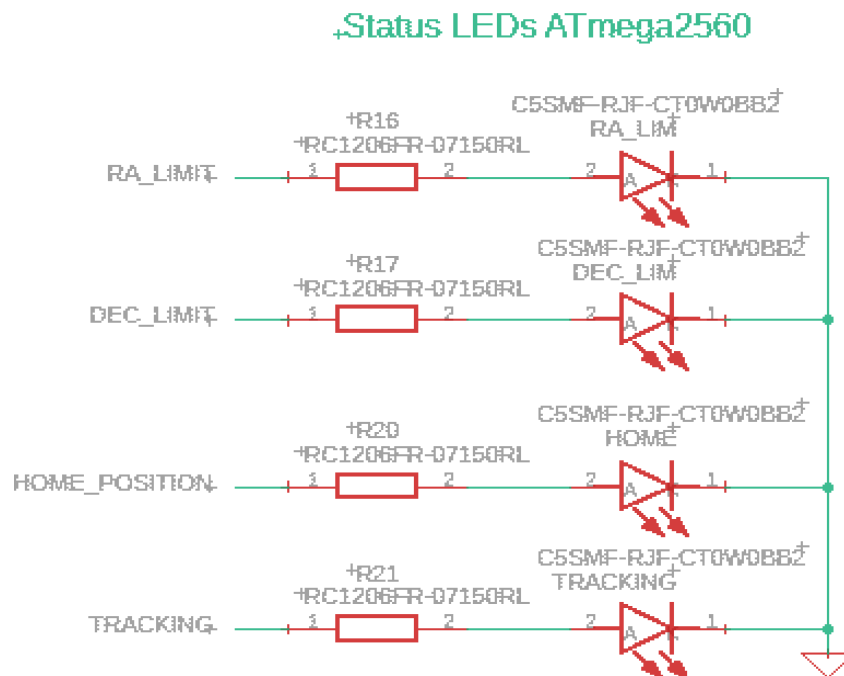


Figure 34: Status LEDs from ATmega2560

±Status LEDs ATmega328

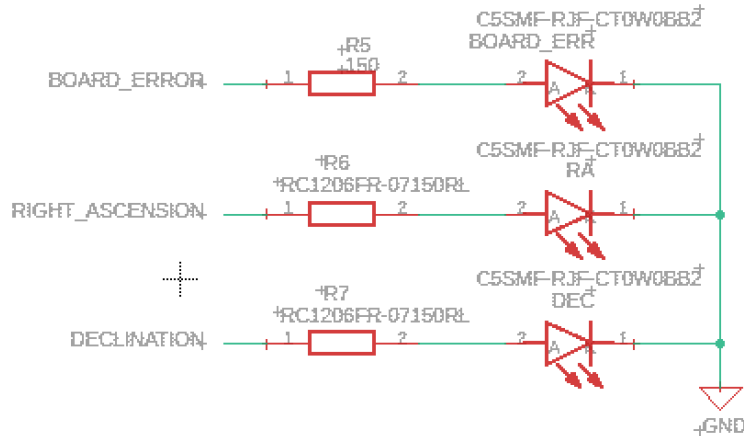


Figure 35: Status LEDs for ATmega328

When designing the LED circuit, it must be understood that a current limiting resistor is required to be in series with the LED. When an LED is forward-biased, the internal resistance is extremely small. Using a form of Ohms Law, $I = \frac{V}{R}$, it can be seen that with an extremely small resistance, the current will increase to a level that is not suitable for the system. The LED cannot handle that level of current, therefore it will blow the LED immediately. To design the circuit with the resistor, an understanding of the forward voltage V_f and forward current I_f of the LED must be known. The V_f is different for different color LEDs. We used red LEDs for all of the status lights. The V_f of red LEDs are around 2V with an I_f of 20mA. Using Ohms Law, and the understanding that the input to the resistor will be 5V because that is the output from the ATmega2560, the required resistance for the resistors was found quite simply via the following application of Ohms law: $R = \frac{5-2}{.02} = 150\Omega$.

The “RIGHT_ASCENSION” and “DECLINATION” LEDs were introduced mainly for diagnostic purposes. If the motors are running, the user can obviously see the motor turn, but if the motor is supposed to be turning but is not, having an LED in the circuit could help the technician determine whether it is because the motor has failed or that the signal is not reaching the motor, resulting in other internal issues within the PCB, or even the signal from the PC.

The “RA_LIMIT” and “DEC_LIMIT” LEDs were for both the user and diagnostic purposes. These LEDs will turn on when the telescope has reached its limit on how far it has turned. If the telescope turns beyond that point, damage will occur to either the mount, the telescope, and/or the system itself. If the LED is illuminated and the motor is nowhere near its physical limit, the user will be able to see that the issue is not that the motor has reached its limit, but that there is an electrical/software issue. This could help speed up the diagnostic process to find the underlying issue.

The “HOME_POSITION” LED was added since we believed it would be helpful for the user because it will let them know when the system has reached its resting home position. At the Robinson Observatory, the system always starts at the home position and turns off at the home position, and if it does not reach the home position, it will not follow through with any commands. In our program, we also had the scale model always turn the motors to reach home before continuing with reading serial commands or user input.

The “TRACKING” LED was used to inform the user that the telescope has found the position in the sky that was requested by the computer and it is locked in and following that target. Lastly, the “BOARD_ERROR” LED was used to indicate to the user that communication has failed between the PCB shield and the ATmega2560 dev board underneath. This serves as a ‘heartbeat’ monitor, as mentioned before and can easily alert the user to fully seat the PCB shield within the ATmega2560 board before performing any other diagnostic practices.

5.3.2 Status LEDs Breadboard Testing

Testing the LEDs was a simple task that could be quickly verified. There were two ways to test that the LEDs worked as designed.

The first way was to build the circuit on a software program that could provide a means for analyzing electrical circuits. The most common used in the education department is Multisim. Multisim, created by National Instruments (NI), is an easy to use circuit simulation tool that Universities use to teach students how to simulate electrical systems. Multisim unfortunately is not open source and therefore is not free to use. There are student editions for purchase, but usually a university with an engineering program will have computers with the software for laboratory experiments that are required for certain classes.

There are other simulation tools that are open source such as LTspice. LTspice is also fairly easy to use, but the capabilities of the program are limited. Some special designs such as non-ideal op amp oscillator analysis is tricky because the majority of the op amps within the program have ideal characteristics therefore do not behave as they do in practice because of the complexity it adds to the analysis. This is fine for most cases, however, if a more in-depth analysis needs to be done for very sensitive systems, Multisim seems to be the better option. There are many more simulation tools for electrical analysis, but access to those options is limited due to mostly cost. These other programs would be used by engineers outside of the education system.

Figure 36 below shows one LED being tested in Multisim. For this test, the voltage across the LED and the current going through it are tested to ensure that it is operating within the limits as defined in the datasheet by the manufacturer. Before

testing the entire eight LEDs in the circuit, one was first tested to eliminate complete failure if the system was designed improperly.

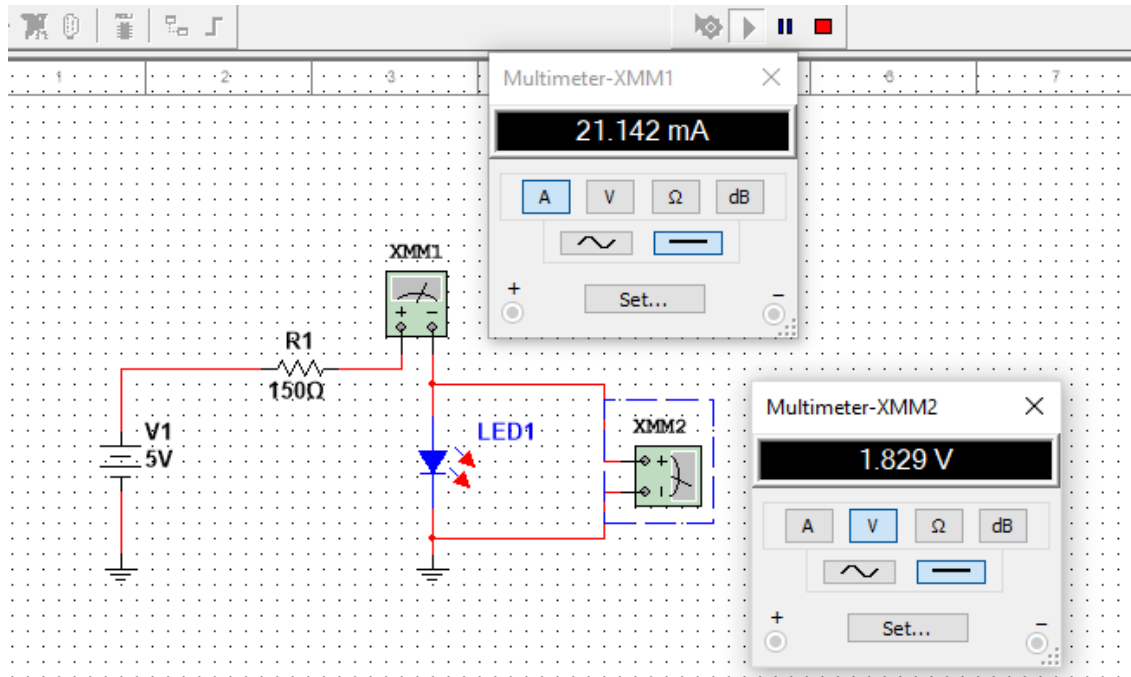


Figure 36: Multisim testing of LED

Once the simulation of the single LED was completed and verified, the simulation could then be carried out with the additional LEDs connected in parallel, however, as long as the voltage source was producing the correct voltage, and the current capabilities of the source could withstand the addition current from each LED, the operation of the system would remain the same. Since the voltage source and current capabilities of the system does not depend on the software, but of the hardware (ATmega), it is unnecessary to simulate the system any further.

Testing the LEDs on the breadboard began in the same manner as it did for the simulation testing by starting first with one LED and testing the circuit to ensure proper voltage and current before moving further in the testing process. This time, the voltage and current capabilities and values were not the ideal values given with the calculations or simulation tools but greatly depended on the many differences in the hardware and testing devices. Before beginning to understand the analysis, the tester had to have understood how their measured values could differ from expected values.

The testing equipment readings varied among the equipment used to test the circuits. As it is with everything else, the higher the cost for an item usually means the higher the quality, and in this case, the higher the accuracy. The factor with the biggest effect on accuracy is temperature. The same goes for the power supply. The power supply will heat up with continued use and request for more power, but

most systems these days are quite efficient for lower cost, but for more accuracy, higher priced power supplies can be purchased. Additionally, the values for the components used are not exact. All components have tolerances that tell the user how much the values can vary. Typical tolerances for resistors are +/- 5%.

The testing of the LED circuit was carried out by connecting it to the ATmega and sending a digital output from one of the pins. This did not only ensure that the LED circuit worked, but that it worked with the current and voltage capabilities of the ATmega. When testing the single LED circuit, the voltage and current was verified for the LED to make sure that it was operating within range and would not fail. The brightness of the LED was also checked to see if the supplied power was enough for the LED. A dim LED is undesirable because the end user would like to clearly see the status of the system which would be achieved by providing a well-lit LED.

After verifying the single LED circuit, the next step was to connect the remaining LEDs to other pins of the ATmega and verify that even when all of the LEDs are on, the current and voltage provided by the ATmega is sufficient enough to power the LEDs simultaneously. This was verified after connecting three LEDs to the power pins on the ATmega328 and four LEDs on the ATmega2560.

5.3.3 Sensors

The sensors in this system can be broken down into two parts, the input diode, and the actual sensor part. The input diode shines an IR light to an optical sensor that senses if the light is reaching the sensor or not. If something goes in between the input diode and the optical sensor, the signal is broken, and there is no output signal. Because of the nature of this design, the input diode needs a constant supply of power to always shine its IR light.

5.3.3.1 Input Diode Design and Schematic

The input diode is a plastic infrared emitting diode within the sensor that shines an infrared (IR) signal. This diode also has a forward voltage and forward current as seen with the status LEDs in the previous section, therefore a current limiting resistor was also needed to keep the current to an acceptable level. Table 10 below shows the electrical characteristics of the device we needed to design the circuit.

Table 10: IR Diode Characteristics

	PARAMETER	MIN	TYP	MAX	UNITS	TEST CONDITION
V_F	Forward Voltage	-	-	1.8	V	$I_F = 20\text{mA}$

Using an input voltage of 3.3V coming from a constant output from the ATmega2560, V_F of 1.6V, and an I_F of 20mA, the value of the resistor could be found for the IR diodes. Once again, using Ohms Law, we found that $R = \frac{3.3-1.6}{.02} = 85\Omega$. Figure 37 below shows the basic circuit that makes up the IR diodes. Since there were two sensors, there were two diodes that needed powering.

+Supply to Input Diode of OPB980 Sensors

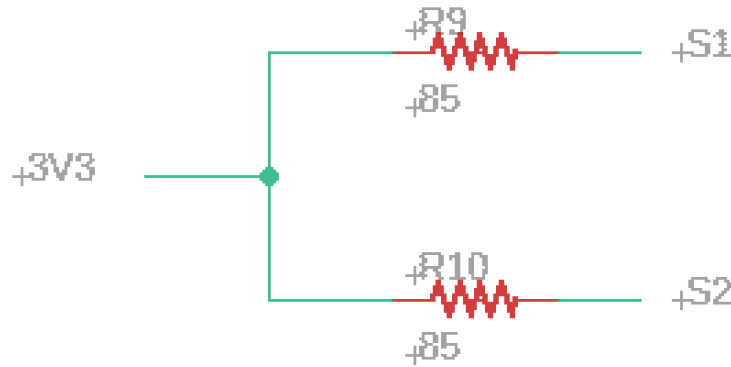


Figure 37: IR Diode Schematic

5.3.3.1a Input Diode Breadboard Testing

Because the input diode is an IR LED, testing of the operation of the LED cannot be done visually. The current and voltage of the LED could be verified by the same process as performed in the previous section of testing the LEDs. Once this testing is verified, the remaining components of the sensor could be connected, as seen in the next section, allowing the entire sensor to be tested and verified.

5.3.3.2 Optical Sensor Design and Schematic

The design of the optical sensor portion of the sensor was also quite simple. Figure 38 below shows the internals of the sensor, including the diode.

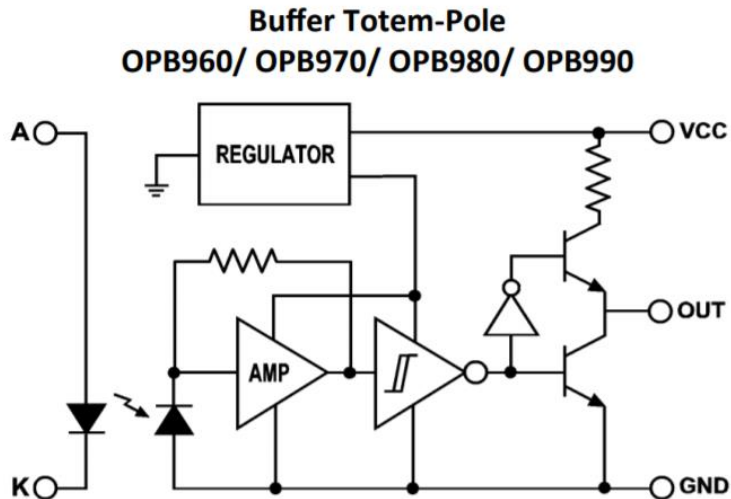


Figure 38: Optical Sensor Internal Components, Courtesy Texas Instruments

It can be seen that the three terminals that make up the optical sensor portion of the sensor are V_{CC} , OUT, and GND. The electrical characteristics for V_{CC} and OUT (V_{OL} and V_{OH}) shown by two cases of the output being high or low, can be seen in Table 11 below for OPB980 which was found within the datasheet.

Table 11: OPB980 Electrical Characteristics

PARAMETER	MIN	TYP	MAX	UNITS	TEST CONDITION	
V_{CC}	Operating D.C Supply Voltage	4.5	-	16	V	
V_{OL}	Buffer Totem-Pole	-	-	.4	V	$V_{CC} = 4.5V$ $I_{OL} = 12.8mA$ $I_F = 0mA$
V_{OH}	Buffer Totem-Pole	$V_{CC} - 2.1$	-	-	V	$V_{CC} = 4.5V$ to 16V $I_{OL} = 800\mu A$ $I_F = 15mA$

The OUT for the sensor was found to be a minimum of $V_{CC} - 2.1$, which was enough for the ATmega2560 to recognize a high or low signal. The V_{CC} we used was the output of the 5V pin on the ATmega2560 to provide a constant voltage to the sensor because it always needed to sense the state of the system. Figure 39 below shows the schematic that was developed to implement these sensors into our system.

+Sensor Connectors

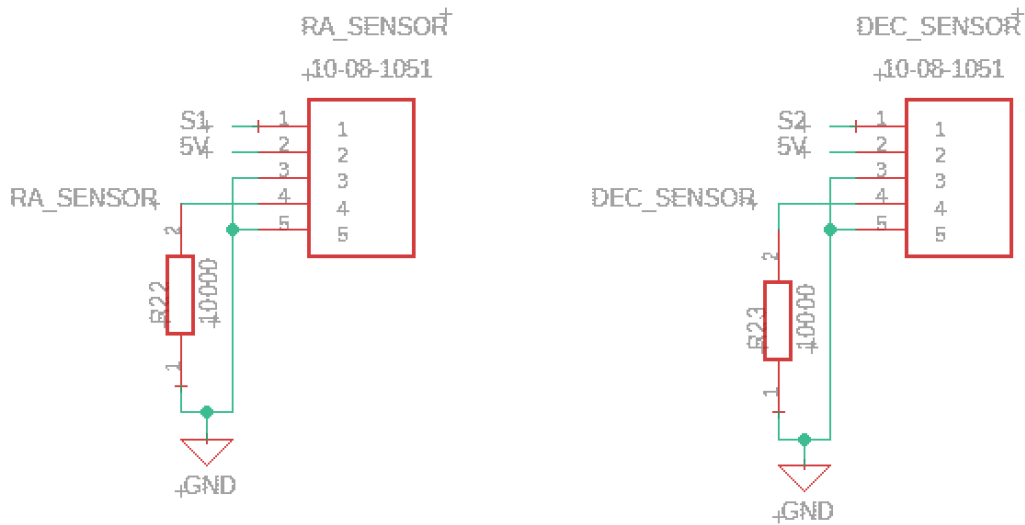


Figure 39: Optical Sensor Schematics

These connectors connect the wires to the board by a means that can be disconnected easily for replacing or testing. The inputs, S1 and S2 are the inputs to the IR diodes, the 5V input is to V_{CC} , the pin 4 labeled “RA_SENSOR” and “DEC_SENSOR” is the output from each sensor, and the rest of the pins are to ground. The reason for the 10k Ω resistor in the schematics was to use it as a pull-down resistor. This enabled the ATmega2560 to accurately measure the state of the sensor as a high or low.

5.3.3.2a Optical Sensor Breadboard Testing

The final testing piece for this particular subsystem was the optical sensor. After testing the IR LED, the remaining wires could be installed. The first step was to simply wire the sensor per the datasheet and test the sensor, then wire the sensor to the input and output (I/O) pins of the ATmega. When testing with the ATmega, not only the sensor capabilities were verified, but also that the sensor worked with the ATmega to signal a change in the state of the sensor.

To test the sensor, the gap in between the sensing element and the IR LED needed to be broken. This could be done by simply introducing any object that is not transparent, because the sensing element is looking for that IR light from the LED. Once the obstruction is placed in the path of the IR LED, the state of the sensor will switch, and in this case, the output would become a high (5V), and when the obstruction is removed, the output would be a low (0V). After verifying the functionality, by introducing the sensor to the ATmega, the tester could verify that the ATmega accepted the voltage output levels as high or low voltages and could also send a signal to the LEDs that were tested in a section above to trigger an LED when the switch opened or closed.

Figure 40 below shows the status LEDs and the sensors wired up with the breadboard the ATmega2560 for testing.

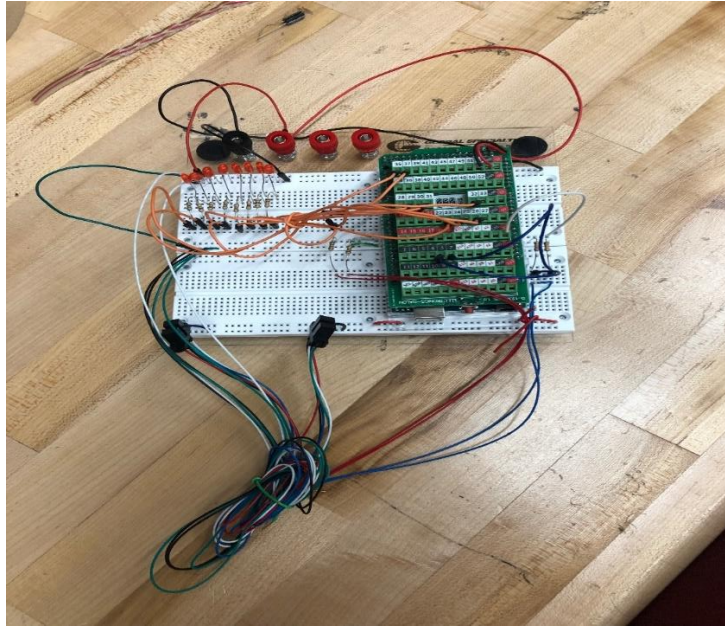


Figure 40: Status LED and sensor testing

5.4 Third Subsystem, Breadboard Test, and Schematics

The third subsystem in our design, generally, consists of the motors. At a practical level, this subsystem can be broken down into several additional subsystems. That is, the motor subsystem consists of the motors, the encoder, the power supply and an AM26C32 differential line receiver. The operation of each of these subsystems will be detailed here along with a summary of the integrated operation of the motor subsystem at the end of this section.

The overall schematic for this subsystem as designed in Senior Design I is included here as Figure 41, and each component and interconnect will be detailed in the appropriate subsection. It is important to note that we have two motors and encoders in use in this application; only one is presented below to simplify the design discussion. In Senior Design II, a different footprint for the differential line receiver was selected as well as a different part was used for the motor and encoder connectors, making the updated schematic look a bit different. The updated schematics are found in the different subsections.

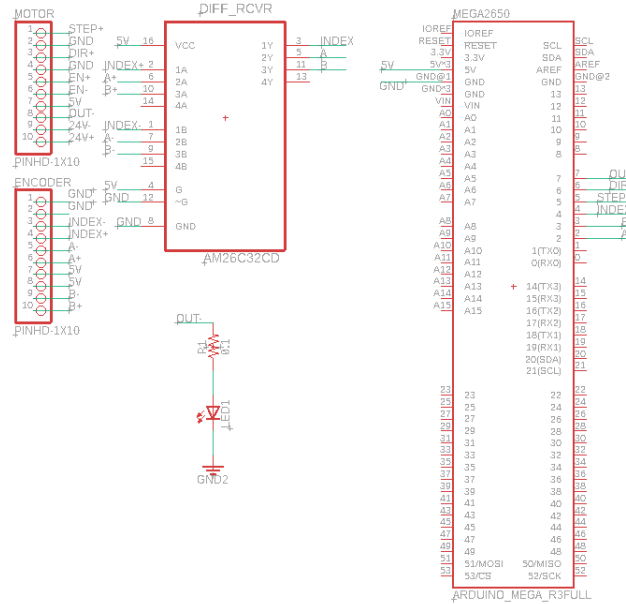


Figure 41: Complete Motor/Encoder Schematic from SD1

5.4.1 Motors

Our application employs a pair of STM17R-3NE, NEMA 17 units that incorporate an integrated driver, encoder and motor. This motor was selected for several reasons, including robust configuration options (user selectable current, idle current, load inertia, step size, pulse type and noise filter), a minimum native step size of 200 counts per revolution (CPR) and the availability of an integrated encoder.

The electrical connections for this motor are summarized by Figure 42 below. These are reproduced in the above schematic, with this connection located at the top left corner of the layout. It is worth noting that these connections have been reproduced in numerical order, from top to bottom (1 through 11) on this connector with the exception of pin 9, which is designated as no connection (N.C.).

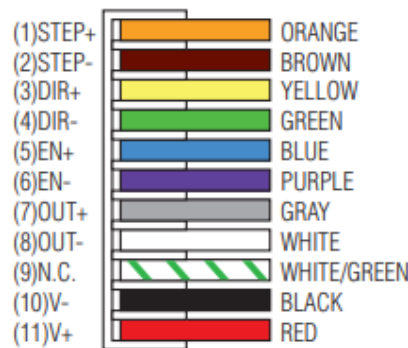
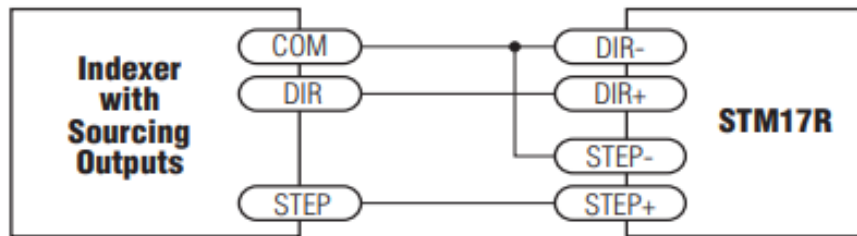


Figure 42: STM17R Motor Connections, courtesy Applied Motion

One of the first considerations on this design was whether it was necessary to implement differential outputs from the motor (STEP/DIR) to the Arduino MEGA 2560. Since the motor and mount were ultimately housed less than a few feet away from the microcontroller, and a large amount of environmental noise was not present, it did not seem necessary to implement the added complexity of an additional differential line driver chip in our design. Therefore, the motor was tied to the Arduino (via our PCB/shield) as represented by Figure 43 below.



Connecting to indexer with Sourcing Outputs

Figure 43: Connections from Microcontroller to Motor, courtesy Applied Motion

The enable (EN) wires could be used for three primary applications. First, they could be used to enable the drive if a secondary drive becomes faulted. Next, they could enable or disable the motor in response to a proximity sensor. And finally, they could be used to implement a kill switch to shut down the motors. The first two applications were clearly unnecessary in our design. Additionally, since our overall unit included the ability to cut power to the motors, the kill switch option was also not used. Therefore, these EN connections were unused in our design.

The OUT outputs close in response to a motor fault condition, providing a digital signal across the output. Our design did initially utilize these outputs. The use was twofold. First, we had an LED connected to the output. This meant that if a fault condition occurred on the motor, the LED would be illuminated. Additionally, we had this output line connected to one of the digital I/O pins on our microcontroller. We would occasionally poll this line to determine if there had been a fault condition. Unfortunately, the Arduino MEGA 2650 only supports 6 I/O pins with attachable interrupts, and these were all required for other applications. Therefore, polling this input would be required. During Senior Design II, we ended up removing this fault LED from the board when changing connectors to save more space on the board. We came to this decision because there was already a built in LED on the motors that could tell us if there was a fault, which occurred rarely.

5.4.1.1 Configuration Options

This motor supports two primary means of control for step and direction. In the first application, the controller pulses one signal for each step in the clockwise (CW) direction and another signal for each step in the counterclockwise (CCW) direction. This is an unconventional control scheme, although it is supported by

the motor and can be enabled through dipswitch #8 on the motor. The second scheme, and the one that we selected, is referred to as “Step and Direction.” In this scheme, the step signal pulses once for each motor step and the direction signal is used to determine direction with a simple high/low input.

The step size on this motor is dipswitch-configurable (dipswitches #1 - #4) and can range anywhere from 200 CPR to 25600 CPR. The ultimate selection for this variable would be dependent upon the needs of the mechanical engineering team (i.e. dependent on the gear ratio that they select), but for our initial testing configuration we elected a step size of 400 CPR. After later integrating with the mechanical team, it was discovered that a step size of 800 CPR was optimal.

At lower step resolutions, the manufacturer advises that the motors can run more roughly than in the higher CPR configurations and produce audible noise. Therefore, an option called micro step emulation (or step smoothing) is available on dipswitch #6 and needed to be enabled unless we elected a CPR of 2000 or higher. There is a minimal lag associated with the command filter used for this process. This ended up not being an impact on our application, and if it had (e.g. in the earth’s rotation tracking capability of the system), it would have been an easy adjustment to disable this feature. Figure 44 below graphically demonstrates the lag associated with this option.

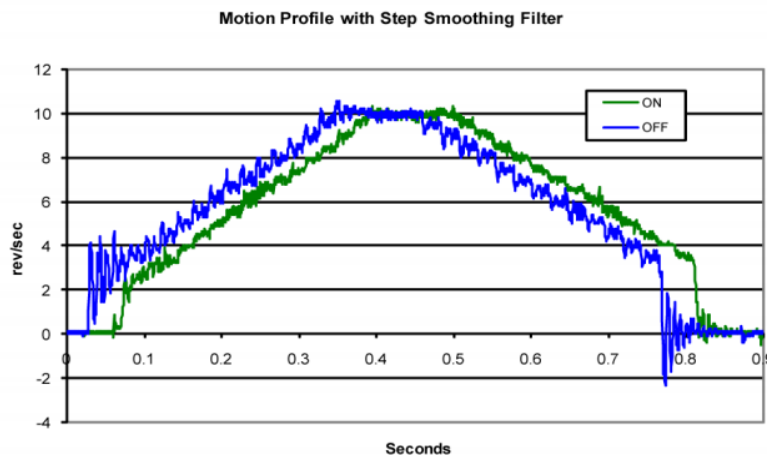


Figure 44: Motion Profile with Step Smoothing Filter, courtesy Applied Motion

The STM17R includes a digital noise filter on both STEP and DIR inputs. The purpose of this filter is to prevent noise that may cause the drive to interpret a single step pulse as multiple pulses, thus negatively impacting the motor. The selection for this setting is determined by the speed at which the motor is being driven. This will be addressed in further detail in the following section, but we estimated that our needs would require a pulse of less than 150kHz to drive the motor. Therefore, this option was set to 150kHz.

The additional configuration options were dependent on the needs and input of the mechanical engineering team. These will be addressed briefly here, and our

selections for design and testing purposes will be enumerated. However, these selections did not impact the design work of the electrical engineering team and could easily be adjusted upon request of the mechanical engineering team.

The current provided to the motor can be controlled through dipswitches #1 and #2, and defaults at 100%. Lower current equates to reduced torque alongside reduced power consumption and motor temperature. The best practice here was to work with the mechanical team to confirm the torque needed for their design and select the lowest reasonable option.

Similarly, dipswitch #3 controls the idle current. Idle current can be set to either 50% or 90% of the running current, with a reduced idle current reducing holding torque in conjunction with reducing motor heating. Again, the lowest functional value for this variable was elected after the mechanical engineering team completed their design. The value we used was 90% which did have motor heating, however the design was ventilated enough that this wasn't a problem.

Finally, we consider load inertia. This is a simple calculation (load inertia divided by STM17 rotor inertia – 82 g-cm²) and was ultimately determined by the final design of the motor mount. For testing purposes, we elected the lower of the values, setting it at 0-4X.

Therefore, the overall settings of the motor are summarized below in Table 12. The table shows how the settings were changed based upon the final needs of the telescope mount. These settings were the same for both the right ascension motor and declination motor, but did not impact the programming or design of the electrical engineering team.

Table 12: Motor Configuration Selections

Dipswitch #	Description	Test Setting	Final Setting
1 – 2	Current	100%	90%
3	Idle Current	90%	90%
4	Self-Test	Off	Off
5	Step Pulse Noise Filter	150kHz	150kHz
6	Smoothing	On	On
7	Load Inertia	0-4X	0-4X
8	Command	Step and Direction	Step and Direction
1 – 4	Step Size	400	800

5.4.1.2 Control

The configuration that we selected for our motor control was that of Step and Direction. Direction is expressed very simply – that is, we send a high signal for

CW rotation and a low signal for CCW rotation. Step is slightly more complex, and follows the equation presented below.

$$\text{Pulse Frequency} = RPS_{desired} * \text{Step Count}$$

This equation allows us to specify the rotations per second (RPS) of the motor. This, in conjunction with the gear ratio determined by the mechanical engineering team, allowed us to determine the constant speed required to compensate for the earth's rotation.

However, in practical purposes, the commands that we received from the PC's software package were to rotate a specified number of degrees (or, perhaps with more granularity, arcseconds). In this instance, we relied on a relatively simple calculation of 1 pulse being equal to one step, at a given step size. We had the ability to command the movement at a faster rate for a period of time and slow it down (via a lower frequency) as it approached the set point. On the Arduino, 1ms is equivalent to 1 delay, so our calculation for movement was centered upon how long to pulse the motor at a given frequency to achieve a desired angular displacement. Precise control of the pulse frequency on the Arduino is achieved through the use of an external PWM.h library, detailed additionally in the software section of this document.

5.4.2 Encoders

The STM17R-3NE supports a built-in quadrature incremental rotary encoder. The encoder follows the industry standard 26C31 differential line driver for output, meaning that the channels (A, B and Index) are differential signals. Therefore, a differential line receiver (industry standard 26C32) was required to translate this output before it hit our microcontroller. A brief summary of this differential line receiver is included here before a more detailed discussion of the encoder operation.

5.4.2.1 AM26C32 Quadruple Differential Line Receiver

Since the two encoders support three differential signals each (A, B and Index), a total of two AM26C32 differential line receivers were required for this design. They are designated in the introductory schematic via this same part number. This means that channels 1 through 3 were used on each differential line receiver with channel 4 being unused.

The operation of the differential line receiver is straightforward. The 26C31 differential line driver sends a pair of differential signals – that is, a square waveform and its inverted pair (accomplished through the use of an onboard inverter). These signals are tied directly to the AM26C32 differential line receiver, which then only considers the difference between the two channels and ignores

any signal that is common to both (i.e. noise). This is done to enhance noise immunity. Our encoders were placed relatively close to our microcontroller, so this level of noise immunity was not necessary; however, as it was built-in functionality of the receivers, we elected to support it through the use of the differential line receivers.

Two enable signals exist on this chip, allowing for active high or active low outputs. Although it was only necessary to use one of the enables, we elected to enable the active-high select and disable the active low-select. This had no impact on the functionality of the device, only enforcing an active-high output – which was our desired signal. The active-high enable and V_{CC} both received +5V DC from the Arduino. These chips are designed to operate in the range of approximately 10 mA I_{CC} , so the additional current draw was within a reasonable range.

Several additional notes bear mentioning and factor into the design and troubleshooting of this device. First, if the input lines are inverted, the output will also be inverted. That is, an expected high will be a low and vice versa. Since most code used to determine direction considered the quadrature of these signals, inverted high and low would have led to an incorrect direction determination.

Next, we elected to terminate the differential receiver per the recommendation of the manufacturer. A pair of 110-ohm resistors were placed in series and a 0.0047 μ F capacitor was placed across each differential pair. The purpose of the resistors was to increase noise immunity, which, again, was not an extremely high priority – however, the cost of these additional resistors was quite low and there was little reason to not include them. The capacitor, however, was an important design consideration. The capacitor conserves power on the order of 20 mA per pair, or 120 mA across our 6 pairs. Power consumption was a concern when elements of the design were to be powered through the Arduino.

5.4.2.2 Quadrature Incremental Encoder

For position control, there are three primary signals of interest being sent by the encoder. After they have been interpreted by the differential line receiver, those signals are A, B and Index. The A and B signals are the quadrature signals. That is, depending on the direction of rotation, one signal will lead, and the other signal will lag. Our task was to use the microcontroller to determine which signal is leading the other.

In brief, this was accomplished through the use of interrupts. Therefore, it was essential that the A and B channels of each encoder were tied directly to a digital I/O pin on the Arduino that supports the ability to attach an interrupt. In our case, the Arduino Mega 2560 supports this capability on pins 2, 3, 18, 19, 20 and 21. The specific pin used for the interrupt was inconsequential – the important takeaway here is that four out of our six available interrupt pins had to be dedicated to the pair of A and B outputs coming out of the quadrature encoder.

The encoder bundled with the STM17R-3NE supports 1000 lines of resolution. This means that for each rotation of the motor, 1000 “counts” are sent from the encoder. Our microcontroller code tracks a variable, here referenced as “count,” that will tick upward in increments of one for each clockwise signal from the encoder and down in increments of one for each counterclockwise signal from the encoder.

Therefore, when this “count” variable reached 1000, we knew that we had completed one full clockwise revolution of the motor. Conversely, when the count reaches -1000, we had completed one full counterclockwise revolution of the motor. This position information would need to be cross-referenced against the gear ratio determined by the mechanical engineering team. For our initial design purposes, we selected a temporary gear ratio – which was able to be easily changed later by a simple #define statement. The result of this calculation then informed us on how far the mount has rotated in the clockwise or counterclockwise direction. This was an absolute value, measured in degrees but potentially also had the resolution capability to be measured in arcseconds.

The encoder also provides a third differential signal that was interpreted by our differential line receiver. That is, Index. The Index signal is very straightforward, it sends a pulse each time the motor completes one full revolution. This Index signal was cross-referenced against our count variable to double check the accuracy of our positioning data, with a flag raised if a conflict arose between the two.

The final connections of concern were the +5VDC and ground. There are two +5VDC connections, and the intention was to supply this power directly from the Arduino. The manufacturer specifies that the requirement is 5VDC at 56 mA as typical with 59 mA as max. This, along with the other components drawing power from the Arduino 5V V_{CC} , was calculated to be within the specified 200 mA maximum current draw. However, a possible contingency if this should have proven too high of a current draw would have been to include an additional DC to DC converter on the PCB and draw this power from the dedicated 150W power supply, discussed below. The ground pin shared the common GND pin on the Arduino.

5.4.3 Power Supply

Our final choice for the motor power supply was a product provided by the manufacturer of our motors – Applied Motion Products. We selected their model PS150A24 Power Supply (PSU), a 150W/24V PSU that includes a built-in, active power factor correction (PFC) filter.

The initial design of our board specified that this power supply should be used exclusively for the motors. That is, the PSU has built-in +V and -V connections that were tied directly to the +V and -V inputs on the motor. However, this PSU is capable of a maximum output current of 6.3A at a nominal output voltage of 24V. The motors operate at a peak current draw of 2A each, meaning that this power supply was capable of providing more than enough power for our specified application. Figure 45 below shows the front panel of the selected power supply which lists the input and output specifications.

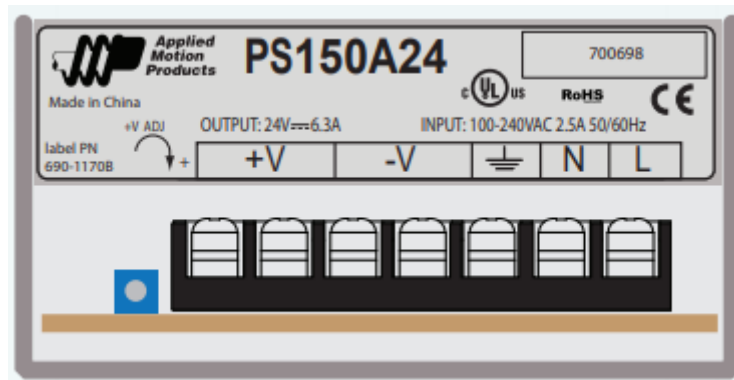


Figure 45: Power Supply Front Panel, courtesy Applied Motion

Should the power demands of our components (joystick, encoders, LEDs and sensors) have exceeded the 200 mA (microcontroller) / 800 mA (5V regulator) that the Arduino was able to provide, a secondary option was to include a DC to DC converter on our PCB and draw additional power from this PSU. However, that option was not necessary in our final PCB design.

5.4.4 Motor and Encoder Breadboard Test

The integrated breadboard test for this subsystem required the motor, encoders, differential line receiver and power supply to work in tandem. In general, the connections followed those outlined on the schematic at the start of this section, but the details are summarized briefly below.

The motors should have their +V and -V inputs tied directly to the power supply. The OUT+ on the motor was tied to the V_{CC} pin (5V regulator) on the Arduino. The OUT- on the encoder is tied to one of the digital I/O pins on the Arduino. In parallel, we had a resistor and an LED, which would trip if the motor was faulted. The EN+ and EN- connections on the motor were unused. DIR- and STEP- were connected to the common ground of the Arduino, while DIR+ and STEP+ were connected to a digital I/O pin.

On the encoder, we connected both ground pins to the common ground of the Arduino. A+, A-, B+, B-, Index+ and Index- were connected to their respective differential line receivers (channels 1 – 3), and the individual outputs of these

receiver pairs were connected to a digital I/O pin on the Arduino. In the case of A and B, pin #s 2, 3, 18, 19, 20 or 21 were selected due to the interrupt capability. Both +5VDC connections were tied to the 5V regulator pin on the Arduino.

For the purposes of our breadboard testing, it is important to understand the power requirements (and thus, current draw) of the various components. During this testing, we connected a digital multimeter (DMM) in series with each of the individual components to determine current draw at idle (for the component) and when they are in use. Similarly, we placed a DMM in series with the 5V regulator on the Arduino so as to have an understanding of the combined current draw of all components connected to this pin. Our goal was to keep all combined current draw under the specified 800 mA that this pin would support.

In general, the breadboard testing was to determine the functionality of each component – not to simulate the complete code package that will be available later in this development cycle. Therefore, we used our PWM.h library in conjunction with simple inputs from the Arduino Serial Monitor to vary the frequency of the pulses output at the digital I/O pins, thus driving the speed of the motor. In addition, we set the DIR pin high and low, to confirm that the motors changed direction as intended. An oscilloscope was connected in parallel with these digital I/O pins to provide visual confirmation of the varied pulse frequencies. It was unlikely that we would be able to force a fault condition on the motor to test the OUT output; therefore, we simply applied 5V to confirm that it was being received by the Arduino and triggered the LED as expected. Figure 46 below shows the breadboard test of our motor with trigger LEDs connected to indicate the various motor conditions.

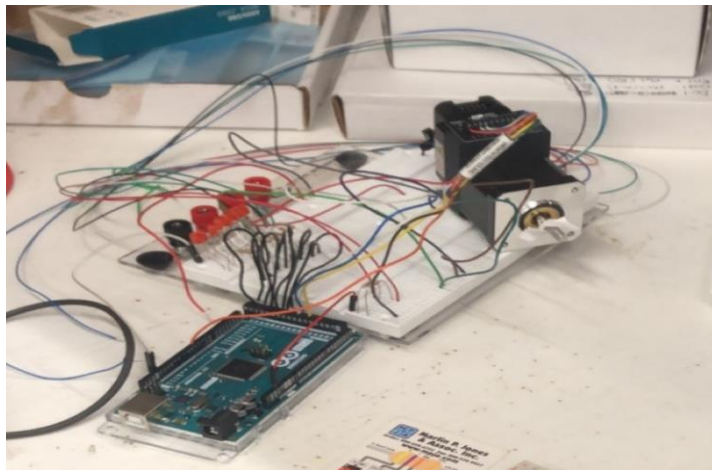


Figure 46: Motor Breadboard Test

Similarly, we would be monitoring the output of the encoder. As the differential signal would have already been parsed by the differential line receiver, we were interested primarily in the A, B and Index signals of each respective encoder. We monitored the output via oscilloscope, similar to how we monitored STEP and DIR going into the motor. In addition, we executed simple code on the Arduino to read the inputs coming into the respective I/O pins. The test case also involved code

that executed an interrupt and performed the necessary computation to determine the direction of the motor based on the feedback from the A and B channels.

5.5 Fourth Subsystem, Breadboard Test, and Schematics

The ATmega microcontroller subsystem is the system that contains the brains and commands for the entire project. It's the connector between different subsystems and facilitates the communication between them. This particular board and microcontroller is made by Arduino and testing was performed on the board directly as the feature set was already built into the board.

5.5.1 Integration of ATmega

To reiterate, the ATmega microcontroller is the brains of all the components of this project meaning all the subsystems report back to the ATmega. The motors, joystick, and sensor and status systems all run through the microcontroller. However, if we had used the DC to DC converter, this would have been the only subsystem that would not require a microcontroller to operate as it would have been more of a support device to help power other components of the project.

The motors, joystick, and sensor/status systems were connected through pin ports on the ATmega PCB, utilizing the boards given features. Features as mentioned were the varying frequency square wave output used by the motor, and the analog input and analog to digital converter built into the ATmega microcontroller used by the joystick. Also, the interrupt capabilities of the ATmega were used for the sensors and status indicators of the telescope.

5.5.2 Testing of the ATmega

The ATmega is a series of microcontrollers manufactured primarily to be used with their user-friendly PCB. Their PCB/development board provides all the functionality of the microcontroller, making any sort of product test and check easy. Essentially the features are baked into the PCB board, which allowed us to test the board which in turn tested the ATmega microcontroller (Figure 47). The board did have some of its own extra features that we could have used to also help with the testing of the ATmega microcontroller.

There are many connections to the ATmega to perform the necessary operations as listed below:

- COMM – Communication with the ATmega328. If this communication fails, the ATmega328 would realize there's an error with the ATmega2560 connection.
- A0/A1 – These are the inputs from the joystick that the ATmega2560 receives. The ATmega2560 read these inputs with the ADC and told the motors to turn based off of the displacement from the origin
- RA/DEC sensors – The sensors that limit the movement of the telescope communicate with the ATmega2560. These sensors were inputs to tell the ATmega2560 when either motor has turned too far and were used to signal LEDs and stop motors.
- RA/DEC A/B/I – These are input encoder signals from each motor. The ATmega2560 used these signals for understanding the position of each motor to adjust speed or send back communication.
- RA/DEC OUT – These are outputs from the motors when faults occur within the motor. The ATmega2560 would then signal LEDs and stop motors to prevent any further damage.
- HOME POSITION – This is an LED output that is turned on when the ATmega2560 receives a change in input from the sensors. Both sensors had to be triggered for the Home LED to illuminate.
- TRACKING – This is an LED output that is turned on when the ATmega2560 knows the motors are in their correct position and are turning at a rate to keep up with earth's rotation.
- JS – This is an input from the joystick used to signal for the ATmega2560 to start tracking. Once a high is obtained, the ATmega2560 will send signals to keep up with earth's rotation.
- RA/DEC STEP – This is a pulse output from the ATmega2560. It will send outputs when it receives an input from the software or the joystick.
- RA/DEC RUNNING – This is an output LED turned on by the ATmega2560 when it is sending signals to the motors.
- RA/DEC LIMIT – This is an output LED that the ATmega2560 turns on when the sensor sends a signal to the ATmega2560 that a motor has turned too far.

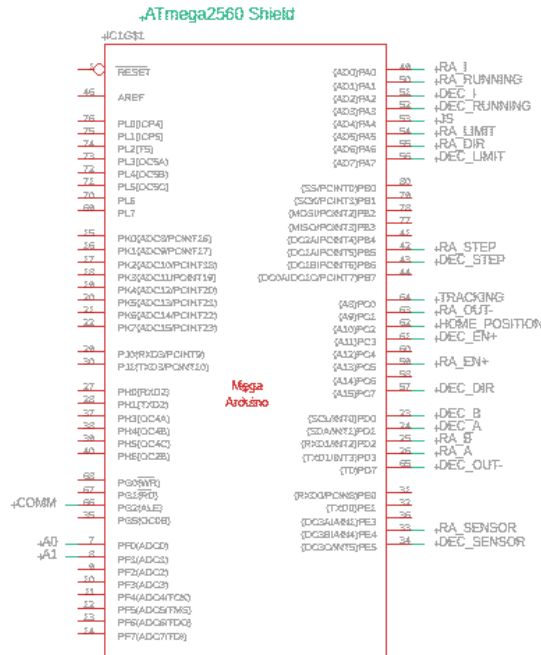


Figure 47: ATmega2560 with Connections

Briefly mentioned above, the ATmega has a built-in analog to digital converter that can be easily tested with a computer and a function generator. The function generator can create most common types of waves which was more than enough to test the ADC. The analog to digital converter is a straightforward device that takes an input signal and divides it up between the listed resolution of the pin (usually 0 to 1023). So technically a DC source that is manually manipulated can be used as well. After wiring it all up, there was a little extra work that needed to be done on the software side. To be the most accurate for a real time display of what the ADC pin is doing we needed a serial output to a computer. Setting up a small serial monitor allowed us to see the analog wave deconstruction which is made by the function generator. The next test involves the ISR's and interrupts. A simple circuit could be made for testing an ISR by wiring up a button from the output 5 Volts on the board, to the selected pin capable of performing the interrupt, and then putting a button in-between 5 volts and the selected trigger. The button could be pressed and if the interrupts triggered, then the corresponding ISR would activate. A serial monitor could be used again to monitor the output.

Another test we performed was a test to determine if the writing of our pins would actually equal a designed frequency that we desired. These pins can be written to with digitalWrite and analogWrite to produce an output. Using an oscilloscope to measure the output directly from the pins, we could reliably see how the pin is reacting to our code. This both checked the board connections and the ATmega's pin capability and functionality for the motors. In summary, testing the ATmega's board and microcontroller itself would be simple but crucial to the project and we could have even used the tested ATmega to help system test other parts of the project.

5.6 Fifth Subsystem, Breadboard Test, and Schematics

For the fifth subsystem of the project which is the joystick, the following two sections describe the schematic layout of the subsystem, factors that went into designing the subsystem, any tools used to assist in the design of the subsystem, as well as how to verify the subsystem design is working as expected via breadboard testing.

5.6.1 Joystick Design and Schematic

Although the selected joystick came with a breakout board, it is still important to understand how the joystick functions by designing the electrical schematic in EAGLE. Figure 48 below shows the connections that are made on the breakout board of the selected joystick.

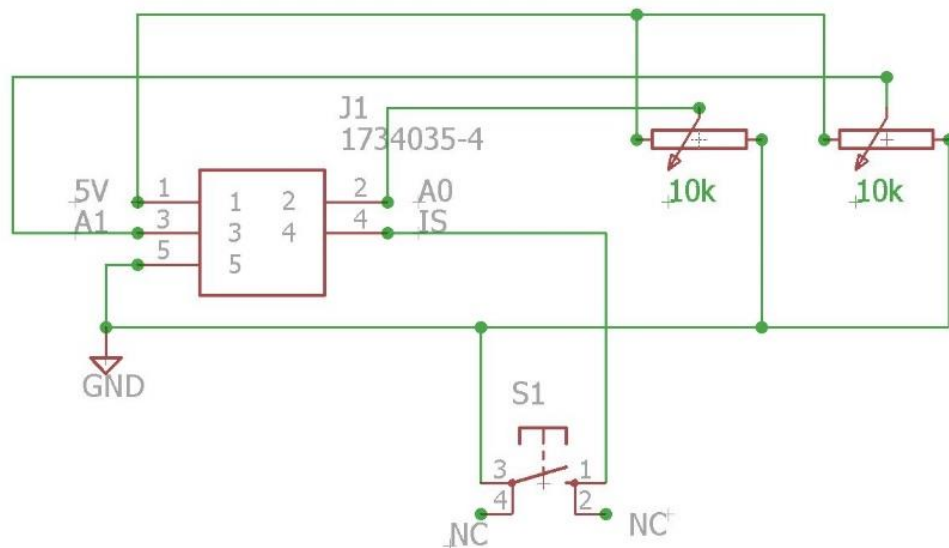


Figure 48: Joystick Schematic

The joystick schematic consists mainly of two 10k Ω potentiometers and a push button. The 10k Ω potentiometers use the middle pin as an analog output and the outer pins are 5V and GND, respectively. The push button shorts the digital output to ground when pressed.

The other component on this EAGLE schematic is the Mini USB Connector. In the actual implementation of the joystick sub-system, there was a converter cable in-between the joystick components (potentiometer and push-button) and the first revision of the PCB mounted Mini USB connector. The connector on the joystick

side of the cable was a 5-pin female 0.1" pitch header. This header plugged in directly to the joystick breakout board. The pros of using the mini USB connector on the PCB as opposed to another 5-pin header was that it allowed for quick connect and disconnect capabilities and they were common enough to have EAGLE footprints available for use. Even with those advantages in mind, in the second revision of the PCB, we elected to not use the mini USB connector and instead a 5 pin connector similar to those used for the motors and encoders, because we realized that with the USB connector cable, one of the needed five wires was terminated at the connector's shield, making that wire unusable.

The next schematic in Figure 49 shows the addition of three LEDs and current limiting resistors to show the functionality of the joystick. As the potentiometer connected to A0 was increased, the LED2 shown below increased in brightness. Similarly, as the potentiometer connected to A1 was increased, the LED1 shown below increased in brightness. For the push button, when it was pressed, LED3 shown below was set to active low and turned on.

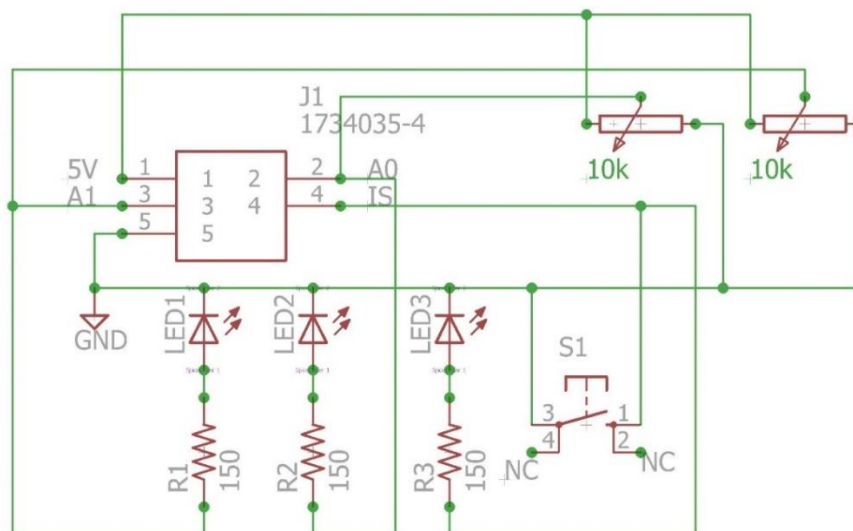


Figure 49: Joystick Schematic with LEDs for Testing

5.6.2 Joystick Breadboard Test

To test the joystick operation, a simple breadboard test was performed using the Arduino Mega 2560, breadboard wires, LEDs, and resistors. Before wiring it up, first we had to solder the joystick components to the breakout board. There was a total of four solder points for mounting the base of the joystick, three solder points for each potentiometer, four solder points for the push button and five solder points for the header which was what the breadboard wires were connected to. Care had to be taken to not hold the soldering iron too long on any of the solder points so as to not burn the plastic of the breakout board surrounding the solder points. To

access the underside of the breakout board for soldering, first the joystick body and components were aligned and pushed into the breakout board holes. Then, a rubber band fixture was wrapped around the assembly so that when held upside down, the joystick body would not detach from the breakout board. Next, the order in which the components were soldered began with the four mounting solder points, then the potentiometer solder points, then the push button solder points, and lastly the header solder points.

The program written in the Arduino IDE was loaded onto the Arduino MEGA shown below and the joystick's operation was verified. In Figure 50 below, the red LED is off because the select button was not pressed. The other two LEDs were at medium brightness since the joystick was in the center position along both X and Y axes. In the actual implementation however, the center position of the joystick actually correlated to zero speed on the motors instead of medium speed and the direction of the axis correlated to the direction the motors would spin.

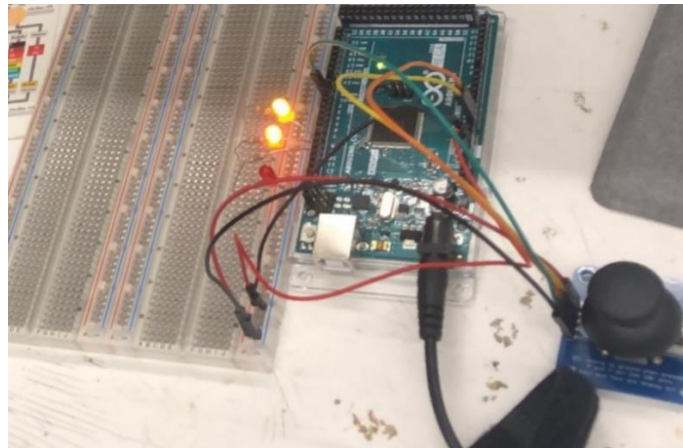


Figure 50: Breadboard Test for Joystick

5.7 Sixth Subsystem, Breadboard Test, and Schematics

The sixth subsystem for our design is the ATmega328 microcontroller. The reason for the additional microcontroller in our system was because the university required a microcontroller that is not a part of a development board. Even though we also used the development board because the sponsor requested it, we also had to satisfy the requirements made by the university for sufficient PCB design. The operation of the microcontroller was quite simple. The purpose of it was to monitor the operation of the main board (ATmega2560) and notify the user if there was an issue with the operation by illuminating an LED. The schematic for the circuit can be seen in Figure 51 below.

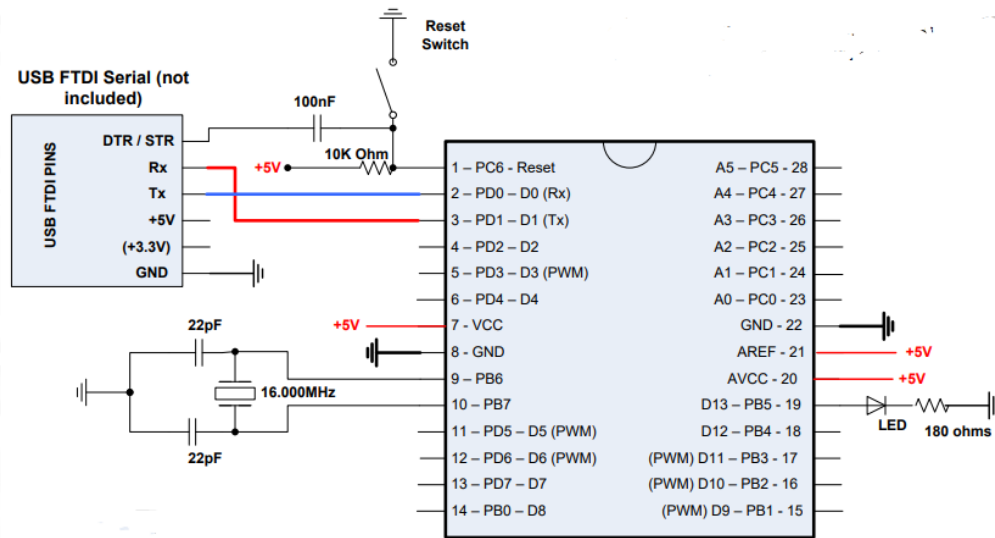


Figure 52: ATmega 328 Programming

5.8 Software Design

The design of the software had to be something that took all the listed restrictions into account and then organized them and executed itself to achieve the desired goal. Our goal of this project as stated before, was to create a telescope that tracks celestial bodies per the user's request. This incorporated motors, optical sensors, analog joystick, and a large external program that would be provided by the Computer Science team. These components are what form up what most of the code had to be capable of handling. The motors had to have been sent a signal on which direction to turn and how fast, and the optical sensors were sending signals to the microcontroller which would cause interrupts. The joystick provided signals to the microcontroller as well as the CS teams program which provided instructions on how to move and where. In the end, the entire telescope would be able to track objects in the sky.

5.8.1 Basic Arduino programming

One of Arduino's goals was to simplify the debugging process and make it easy for programmers to understand quickly. Normally microcontrollers give the designer a blank slate to create something on and then everything else is left up to the user. However, Arduino decided to provide some preliminary functions for users to work with. Instead of just a simple main function that the user must then create their own while loop to have the function repeat itself, Arduino provides a function that takes no arguments called `loop()`. This loop function does exactly what the name gives away, which is looping indefinitely if the programmer so desires. Any code that goes into this function will continuously be looped over and

over again until only exiting if a return value is set or an interrupt service routine is triggered. Even with the ISR, the loop will continue right where it left off after the ISR is finished executing its commands. Due to this, the intriguing part about the `loop()` function is that it doesn't have a conditional argument that needs to be made. The default state of the Arduino is to continuously run the loop function forever. So, to get the polar opposite of it, we would use the second function Arduino provides: the `setup()` function. Again, this function does not take any arguments in its initializer and has a default return type of `void`. The purpose of this function is to run only once in the entire code. After that, it is never run again unless the microcontroller is reset. Seeing that it only gets run once, programmers usually adhere to this as the function tells them to and set up the board for the rest of the program. This includes pin assigns, library starts, timer choices and an assortment of other things are located in the `setup()` function, but nothing heavily related to the brains of the program.

5.8.2 Integrated Development Environment

An IDE stands for integrated development environment and is essentially a tool that can be used by programmers to write code that can be tested and released. Without the use of IDE's programmers have a much more manual job to do when writing code. Programmers would often just write code in a text file or maybe a colorful text editor which would then be compiled in the computer console. The console would have to have a system environment pathway set up to be directed towards the compiler so that it could be able to understand the commands the programmer put in. Compile commands, `grep`, and possible other features the programmer had to manually install like a debugger or a memory leak finder such as `Valgrind`. Furthermore, aside from all the extra useful tools that were used, the programmer would then have to properly set up a method of porting the compiled machine code to the actual microcontroller. It was a huge process that involved a steep learning curve to properly get everything working, and even then, things could always go wrong. Then came IDEs serving as a toolbox for all the programmers' tools. Containing all these features previously listed and much more, it became the go to method of coding for programmers.

5.8.3 Interrupt Service Routines

The final main element of how the programming would be designed and utilized is something that was alluded to previously. Interrupt service routines or ISR's are specialized functions that are only supposed to be executed when a configured bit flag is raised (Figure 53). What internally happens involves a LIFO stack, a pointer, and some bit flags. When a particular desired bit flag gets triggered, it causes the microcontroller to immediately move its internal SP counter to the ISR location address and start running the ISR functions internally. While that's happening the preceding SP counter location is put on a Last in First out (LIFO) stack to keep track of what the program was doing before. The ISR completes its processes and

then moves the SP to next address on the stack and continues executing from there.

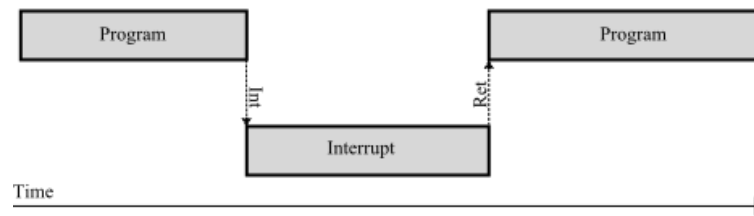


Figure 53: ISR Priority Diagram

This mechanic in the software is translated into pre-organizing functions, aptly named ISR's in Arduino. Arduino allowed us to set up certain triggers on pins that would then immediately activate the corresponding code functions when the flag was pulled. This was fantastic towards the goals of our project because we could then effectively respond instantaneously to real world stimuli and take action in our code to correspond to it. As stated antecedently, the telescope had optical sensors attached to it to prevent harmful actions to itself throughout its movement. ISR's were a great way of relaying real time information of the telescopes position to the program controlling it.

The programming design of ISR's in Arduino have an important list of things that need to be setup properly before being able to properly use it. Most of the work that is involved utilizes Arduino made functions that are specifically tailored to the functionality of ISR's. To get ISR's to work we need to use some specific function calls named, `pinMode()`, `attachInterrupt()`, and `digitalPinToInterrupt()`. The `pinMode()` function call is actually not specifically tied to ISR's, it just needs to be used to tell the pin that it's going to be used as an INPUT. The function takes two arguments, the pin under question and the type of pin it is. Doing so will set that pin to be an INPUT or OUTPUT for the rest of the program's life, unless reassigned elsewhere.

Beginning the second, most significant function, is the function call `attachInterrupt()`. This function is where the actual interrupt and its details are written on how to flag the interrupt pin and what to do. First off, the initial argument of `attachInterrupt()` is an integer value delineating the pin number to be assigned. Be that as it may, the integer value is not actually mapped to the specific pin value associated with the rest of the board. Arduino has an internal interrupt mapping of pins for each board that represents the true value of the interrupt integer (Figure 54).

BOARD	INT.0	INT.1	INT.2	INT.3	INT.4	INT.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
32u4 based (e.g Leonardo, Micro)	3	2	0	1	7	

Figure 54: Actual Digital Pin Interrupt Mapping

Arduino offers a table to illustrate the actual correct value of the integer needed to be inputted, and also offers a built-in function called `digitalPinToInterrupt()` that essentially does the correct mapping of pins regardless. This fills out the first argument.

The second argument is where the name of the operation comes from, the ISR function. This argument takes the address of the function name that is to be executed when the flag is raised. A programmer can name this function anything they want, but the function must adhere to certain principles. The ISR function cannot take any arguments, nor return any value forcing it to have a void return type. This is because ISR functions can't return a value to something that can't ask for a value and can't take any arguments because the flag raised doesn't have any data arguments to give.

The third and final argument of `attachInterrupt()` is the flag type. This argument explicitly describes when the flag should be raised to run the ISR function. Values of this argument take the form of events in a signal such as, CHANGE, RISING, FALLING, LOW, and sometimes HIGH. When a signal is read from an input, certain features can appear in the signal themselves. For such instance, if a binary signal such as a square wave is sent, then at some point in the signals timespan there is going to be its low point as well as a high point. Transitioning between these two points also can be expressed as a rising or falling edge. These characteristics of the signal are better illustrated in the following picture (Figure 55).

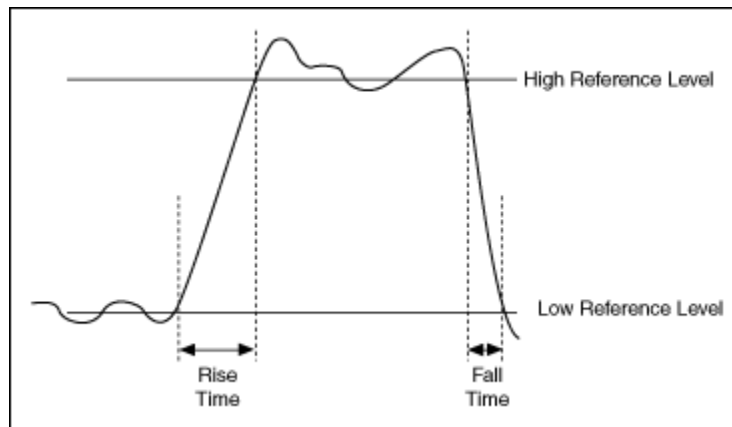


Figure 55: Digital Deconstruction and Characteristics of a Wave

Since our event that ties to an interrupt can only take on the form of a binary signal, the signal then must be a square wave with an even more discrete Rising, Falling, Low, and High position. Being that the signal is generated by an optical sensor which sends a high value when it senses obstruction, we would put a RISING flag type to be associated with it. The reason we don't use HIGH is because the Arduino MEGA doesn't actually offer HIGH trigger responses. And even if it did, a problem could occur within the code if this did happen. Setting the interrupt to respond to a HIGH trigger means that the ISR associated with that will be ran whenever as long as the signal remains HIGH, which is terrible. This correlates to the ISR being ran multitude of times instead of just once like we want to, causing potentially unknown side effects.

Another unfortunate restriction to ISRs is that it is highly disapproved to use clock delays within the ISR. The problem boils down to the simple fact that an ISR is supposed to be as fast as physically possible, and a time delay counts the values of millis() which is translated into a literal delay meant to burn time as requested by the user. The millis() function is a very basic function that principally just counts how many milliseconds have passed since the program started on the Arduino. It holds this value in an unsigned long variable which means that it can't go negative. It also means that after approximately 50 days, the number will overflow leading back to zero.

The delay() function is just a function that monitors the millis() function, effectively stopping all work on the microcontroller. Not only does this cause problem such as not being able to respond to other interrupts, but there is a possibility that the delay() function stops counting the seconds and hard locks your microcontroller. From all this, the software design that we created never used a delay inside of an ISR, instead opting to change a value somewhere else in the code and then that prompts a delay function. This design was used with the optical sensors because their signal is based on a mechanical reaction with a possible scenario of having two interrupts at the same time. If not properly coded, the telescope might turn too far and cause damage to the mount, even possibly causing an alignment error.

5.8.4 Meridian Flip

When discussing the features of an automated tracking telescope, there is one feature that always is included. The feature is called a Meridian flip and is named that because of what it does. For observing and tracking celestial objects on a telescope that has two major axes of rotation, there is a certain edge case that require you to turn the telescope on its head. Both the directions for the motors have to be flipped as well as having to perform an operation beforehand. The best course of action is to explain the problem first.

The problem stems from the limitations of how far the motor and gears can turn in one direction. A telescope that has any significant weight to it usually has some

sort of counter balance to it or way of keeping the center of mass in the center of the telescope. From this arises the limitations that a certain motor/gear can turn before the telescope starts turning into itself, causing physical damage and losing tracking. If an object that is being tracked goes directly over head of the telescope, the telescope will have run into this problem and be unable to continue tracking the object.

The way how this is solved is through the Meridian Flip. The Meridian Flip must essentially swap the position of the counterweight and the telescope. This is good because it puts the telescope in a position that now has the entire gear/motor length to use to track the object in question. The telescope stopped at its max turning angle, performed a meridian flip, and then now can continue starting at the beginning of the turning angle for the axis it was being stopped at. However, aside from the maneuver, there are some more complicated things that arise. Since the telescope is not in the opposite side of everything, all the directions need to be reversed to keep tracking the object. Of course, the speed is going to be the same essentially, but another side effect that will occur is that the image will now appear flipped, and the telescope essentially turned upside down.

In our final implementation, we did not actually utilize the Meridian Flip because our testing mount didn't have a need for it due to the nonexistent possibility of mount damage as well as the nonresistance of heavy counterweights. If we had received the mechanical mount earlier, we would have been able to implement a flip using their physical limits, however they indicated to our team that a meridian flip wouldn't be necessary due to their design choices also not having vulnerability for damage to the mount.

5.8.5 Varying Frequency

The Nema 17 stepper motor that we had ordered is a motor that is controlled by two main inputs. A speed input and a direction input. The direction input is a very simple signal that will be sent when we want to have the motor turn in a particular direction, essentially a binary signal. The speed input signal is a little different. The Nema 17 motor requires the user to input a square wave which is just a binary wave with a 50% duty cycle. This means the wave will just oscillate between two voltages for a similar amount of time spent at both ends. The interesting way how the motor actually values its speed is based on the frequency of the square wave being inserted. Higher frequencies correspond to higher speeds and vice versa. This feature of the motor is actually a very dynamic feature that we had to program around, and we were very grateful that the signal had to be a square wave.

To commence programming of a square wave of varying frequencies we had to tackle two problems. One was what the max frequency of the pin that we would use was, and the other was how to program a changing frequency of a pin.

Arduino's core clock frequency is 16 MHz which was more than enough to allow our motors to track something in the sky. Trackable objects in the sky don't require us to move that fast so we probably anticipated plenty of room to work with in terms of frequency ranges. However, before we even began programming there already seemed to have been a problem. It turned out that while the core clock frequency is at 16 MHz, the output speed of pins on the Arduino Mega clock in at around only 490Hz. This is much slower than 16Mhz and limited our output square wave to essentially half of that at 245Hz. However, there was a fix for this that allowed us to increase the base frequency level of the pin output. Instead of being able to send only 490 signals a second, we could change that level by tying it to another internal timer that outputs at a faster speed. Once it could outputs at a faster speed, the program could then divide the frequency to match which speed we wanted. Arduino Mega has 6 internal clock timers that can be used to change the pins output frequency and get the variable speed we want from there. The motors speed would then be directly controlled by us, allowing us to track stars accurately.

5.8.6 Bit Banging Square Wave VS Direct PWM Outputs

After we had a much higher and usable pin frequency, we could start outputting a square wave of variable frequency that would control the motor speed precisely. The default Arduino library has a function called `digitalWrite()` which allows a user to define a binary signal to be sent to a specific pin. The first argument is a pin number and the second argument is the digital equivalent output. HIGH and LOW appear here again with HIGH being a 5-volt output and LOW being a 0-volt output, which is particular to our board. As used before, it was necessary to utilize the `pinMode()` function to specify the output of our pin. As this output is just a simple square wave all we really needed to do was write a small function that oscillates a HIGH and LOW output. Since the Arduino comes with a built-in delay function, we could cascade the code to have a digital write to high, a delay of X time, a digital write to low, and then a delay of the same X time. Looping these four lines of code would end up simulating a square wave to emit from a port of our choosing. This was great because then we had a way of varying the frequency based on a single variable with a duty cycle of 50% (Figure 56). This method was tested and showed some pretty interesting results. For certain pins the square wave worked nearly perfectly and had a variable speed based on the pins. However, some pins that were used still resulted in a square wave, except the square wave had a natural decay response when oscillating down to 0. This looked very scary at first but ended up still working well enough for the motor to vary its frequency. We also could electrically attach a sort of 5-volt regulator to the output that only allowed two voltages to pass, being 5 or 0. Regardless of all this, there seemed to be a slightly bigger problem discussed on the following page.

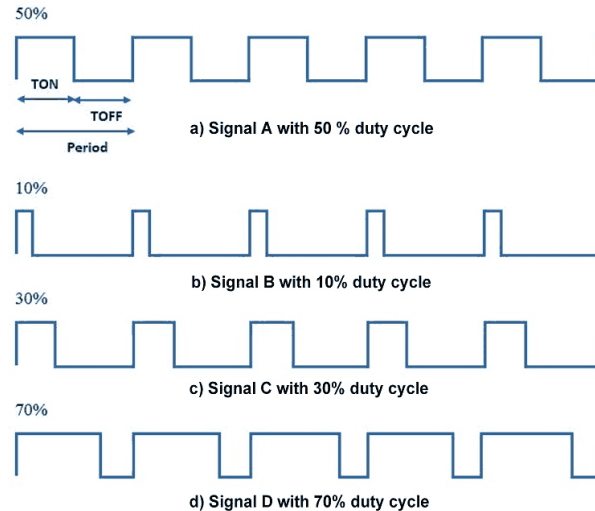


Figure 56: Duty Cycle of a Square wave

The problem that we ran into next was the processor use. As it stood with that code, it meant that the processor would be using 100% of its resources to toggle a digital pin on and off over and over again. This would have been really unfortunate. We could squeeze in two different pins that output at the same frequency but for tracking and optimization it would have been terrible. The other part that could have really ended up being a bad play was that while moving the telescope, the Arduino would be blind to everything that's going on around it aside from interrupts. No serial commands could come in nor commands from a joystick. However, several different ways could be used to have the Arduino send out a square wave that doesn't take up all the resources of the microcontroller.

AnalogWrite() is a command in Arduino that allows you to write a variable square wave from a pin that can output a PWM signal. To properly explain what a PWM signal is, the following image makes it really easy to understand (Figure 57).

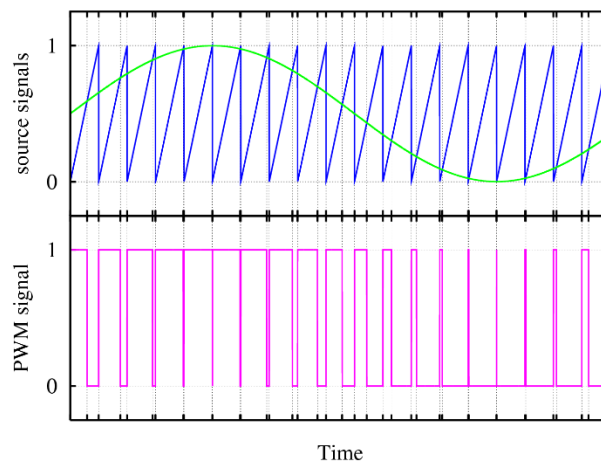


Figure 57: Analog signal transformed into PWM signal

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation license from commons.wikimedia.org

If there is an analog signal that we would like to reproduce but only have access to it in a digital form, the microcontroller will not be able to reproduce the analog signal as shown in green. What a PWM signal does is it matches the analog signal with a triangle or sawtooth wave (in blue) that essentially samples and compares the analog signal. Both sawtooth and analog signals are fed through a comparator circuit which then spits out a binary signal telling the viewer which signal is greater for the given time. When the analog signal is larger than the sawtooth the binary value tends to be more of a 1 than a 0. When the analog typically is smaller than the sawtooth the binary value tends to be more of a 0 than a 1. This output just looks like a square wave with a varying duty cycle. The magical part is that when some device needs an analog signal to operate, it effectively can't tell the difference between an analog signal or a PWM signal. This is good, but some work was needed to change the output of a PWM pin to a constant duty cycle and a varying frequency. The manipulation of `analogWrite()` would get us to what we want. The function takes two values, one of which is the pin number, and the other is the duty cycle number. This is the caveat though, it doesn't have a method of varying the frequency of said square wave. For that though there were several different ways of possibly fixing this issue, most of which involve changing the pins internal timer to have a higher default frequency. Once we set a particular PWM pin to have a specific timer frequency the program can then do some division to land the pin on the desired frequency, upholding the initial task of having a varying frequency square wave. The enormous difference now was that we could actually have multiple pins, reacting to multiple commands to run at multiple different frequencies, while also performing other tasks alongside.

5.8.7 Analog Inputs from an Analog Joystick

The design of our telescope was heavily influenced by what the UCF observatory telescope is like, and that telescope has an analog based joystick that allow users to free move the telescope around. In attempting to replicate that telescope our project incorporated a joystick that would produce an analog signal that was fed into an analog pin in our Arduino. Utilizing the `analogRead()` function in Arduino, the signal was sent through as an analog signal and then processed by an ADC. An ADC has been described before as an analog to digital converter. How it does that is it has a set resolution to read 1024 different values from a range of 0 to 5 volts. Anywhere in-between there is set to a specific value that we can read from the return value. Note the resolution values can actually be changed as we see fit meaning we can go higher or lower in the voltage range (Figure 58).

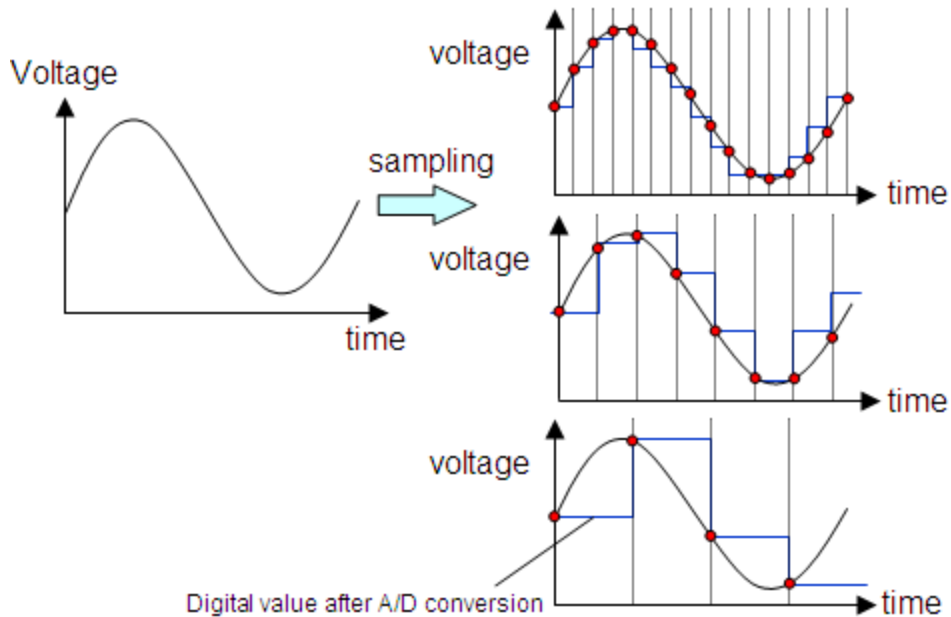


Figure 58: Analog Signal Deconstruction into a Set Digital Resolution

The analog stick that used would send out two analog signals representing the two axes that the user can control. These two signals would then have to be stitched together by us to determine the direction the controller wants to move. The code would have to interpret two numbers between 1023 and 0 that represent the user's position. We can conceptually illustrate this as a Cartesian coordinate system with the two values equaling a point. This point can then be treated as a triangle with a magnitude and angle substantially turning it in to a vector. The vector would then be interpreted by another part of our code to translate that data into workable information for our motors.

5.8.8 Serial Parser

A significant part of the telescopes programming was performed by the Computer Science team. More specifically, they were in charge of taking an open source application and transforming it into a usable star map that any person can just point to a star and immediately begin tracking it. The process would result in a series of motor controls made to get the motors to move the telescope to the intended location. Plenty of calculations must be done on the CS teams' side, but what was guaranteed to our team was a serial input string that would come through a USB with the direction and distance for each motor to move along. Being that the data is sent through a string data type, we must develop a parser to properly interpret the string.

String manipulation is a very detailed task in the C language as it requires the programmer to go byte by byte through the data. This is because a string in C isn't actually a single variable, but rather a set of memory allocated characters with a

single address leading to all the characters that are sequentially listed in memory. What the address pointer will do is read all the characters from the memory address and only stopping when a specific type of character is reached called a null terminator. Every string has these null terminators as the last byte of the string, otherwise the program wouldn't know when to stop reading byte data and stop displaying the corresponding ASCII values seen in Figure 59.

ASCII Table															
Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Figure 59: ASCII character code used for common American computers

At this point the Computer Science team would have to provide us with the method they have of differentiating parts of the string for the values. If the values ended up being differentiated by a period character or a pip character, then we would have used that to find out when they are relaying different information. The plan was to read in the characters of the string one by one and look for a period or character to symbolize the end of one part of data. This data will most likely be a number. This causes a minor problem that is easily fixed by the C language. Since these numbers are coming in as strings the number we get is actually a string that can't be interpreted as a number. If the user attempts to interpret the string number as an actual number, the program will just read out the ASCII value of that string or character rather than the real intended number. Alternatively, it will just emit some garbage value or crash.

However, the C language has a built-in function that allows our program to use to translate a string to a number directly. The `atoi()` function takes a string and processes it to return the real value the string represented as an integer instead of a char array. This is great and can be highly efficient when we attempt to parse our string for data. Unfortunately, there is one small possible problem that we can run into. Certain standard C functions aren't compatible with microcontrollers as they work upon different components of the computer. For instance, the highly

versatile C function `malloc()` doesn't work in a microcontroller setting because it requires an Operating System to work. `Malloc()` is a memory allocator that is designed to work with a computer's memory storage system to create a heap of virtual memory for the program to play around in. Since our microcontroller doesn't have an operating system, it can't generate those heaps of memory in the same way an OS can. The fear is that the `atoi()` function might come under some problems as well, but further inspection showed that the `atoi()` function did indeed work on microcontrollers, allowing us to use this shortcut when parsing the string. If this function did not work, we would have to manually assign the ASCII values to numbers and do the math ourselves if another function could not be found.

5.9 Summary of Design

Many of the design decisions made here were influenced directly by the needs of the customer or a desire to replicate the existing set-up of the observatory. For example, our decision to employ an Arduino in tandem with a shield was driven by the request of the customer. In another example, our decision to use optical switches to ascertain the mount's home position and act as a limit switch was almost entirely because this mirrors the existing setup at the observatory.

Other decisions, such as the NEMA-17 motors, were born out of a desire to retain maximum flexibility. The specific motor that we have elected to use provides a suite of built-in customization options. For example, max and idle current can be adjusted, allowing for more, or less, torque, as needed. In another example, the step count can vary in the range of 200 counts per revolution (CPR) through 25,600 CPR, varying by a factor of more than 100. Maximum flexibility was desired with components such as this, and the microprocessor, that would serve as our primary interface between our peers on the mechanical engineering and computer science teams.

Our overall design can be considered as two broad categories. Input and output (I/O) as well as processing and motor control. The first link in the chain is the I/O. Here, we receive control signals from the software designed by the computer science team. These control signals provide commands to the motor, directing a number of degrees to rotate the mount from its home position, as well as other commands, such as a reset to home position or to enable tracking. These signals are received directly onto the Arduino board via the differential USB connection, and translated into serial inputs which can be processed by the microcontroller.

The microcontroller receives these signals and processes them into motor commands. The motor is driven by a frequency modulated pulse train, with higher frequencies translating into more frequent steps, and thus, higher revolutions per second (RPS). The gear ratio determined by the mechanical engineering team played a key role, here, as our microcontroller would need to provide intermediary

calculations accounting for the gearing so as to drive the mount into the desired position.

As the motor moves, feedback is provided by way of the encoder, as well as the OUT pins on the motor itself. All external connections (with the exception of the USB interface to the computer and external power for the Arduino) are made to the Arduino shield, which has the appropriate I/O accommodated in the PCB design. If implemented, the OUT would have been used only to signal a fault in the motor (and includes the additional illumination of an LED, signaling the fault), whereas the encoder provides data on the direction of rotation, speed and overall position.

The encoder, in particular, uses a quadrature incremental rotary encoder. A differential line receiver sits on the PCB between the encoder and our microcontroller to translate these differential commands. Our coding design required that these inputs be tied to Arduino pins allowing interrupts to be attached. This feedback on positioning is then relayed to the software on the PC per the requirements of the computer engineering team. At any point where the mount's traversal interrupts the optical switches, corresponding position information on "home" or "limit" is sent to the PC and (in the case of a limit switch) motor movement is halted.

In addition to direct commands from the PC software, motor commands can be provided through an external joystick. The current planned implementation of the joystick is to use its "button" input (a simple high or low signal) to transfer command of the motors from the PC to the joystick and send appropriate feedback to notify the software. The joystick is analog, and functions on a 2-axis input. These inputs were connected to the Arduino analog inputs through the shield. The analog inputs include built in analog to digital converters (ADCs). These ADCs index the signal into 1024 distinct levels, allowing our code to interpret the positioning commands input by the user.

Status LEDs exist to provide visual feedback to the user informing them of the state of the mount controller. Examples include an LED to indicate whether the mount has hit a limit or is in a home position, or whether the motor is running and/or tracking. These LEDs are connected to digital I/O pins on the Arduino and driven by setting that output to high. In some instances, such as with the LEDs, it is necessary to send power directly from the microcontroller. In nearly all other instances (such as power to the encoders), the 5V output is provided through a 5V regulator, which has a significantly improved maximum current rating as compared to the Arduino.

The two final considerations are the power supply to the motors and an additional level of supervision due to our implementation of the Arduino shield. The power supply was selected due to the recommendation of the motor manufacturer and meets all appropriate specs. It would have been reasonable to design a power

supply for this purpose, or buy a third party option, but since clean power is an important element of any electronic design, and there was some concern expressed by the manufacturer about regeneration, we elected to go with the recommended power supply.

We had also implemented a second microcontroller on our shield to supervise the connection between the main Arduino board and the auxiliary shield. A “polling” feature was implemented so that if communication ever fails between the two boards, a flag will be raised, and an LED will be illuminated. Additional feedback was a possibility but was determined not to be a necessary part of the design. In addition, the second microcontroller allowed us to offload some of our more power-hungry peripherals (such as LEDs) to this ATmega328. This was to lessen the impact on the 200mA (maximum) current rating of the Arduino 2560.

6. Project Prototype Construction and Coding

In section 6, we have the documented state of the project as it was reaching its design completion. This section discusses the integrated schematic that was developed, along with the plan we had for the software design, and lastly the plans for obtaining the PCB and assembling the required parts.

6.1 Integrated Schematics

This section is used to discuss the process of integrating the subsystems and how they impact each other. After testing and verifying all of our subsystems, the next step was to integrate the subsystems into one system for testing and verifying. Figure 60 below shows the integrated schematic developed in Eagle for the first revision of our PCB.

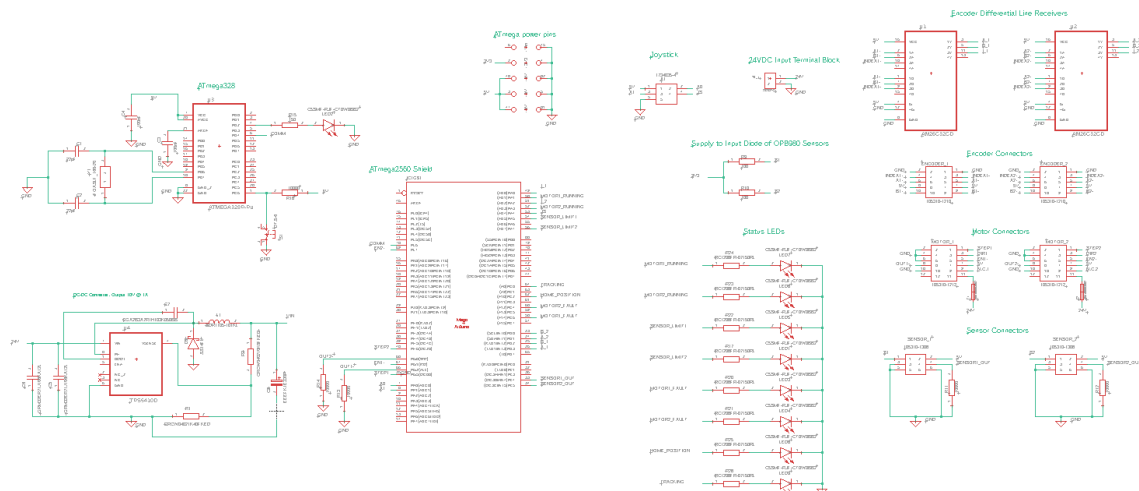


Figure 60: Integrated Schematic, First Revision

At the top left is the ATmega328 subsystem. This subsystem communicates with the ATmega2560 subsystem to determine if there is an issue with the ATmega2560 since it is the brains behind the entire operation. The ATmega328 will illuminate an LED if there are any issues with the ATmega2560.

The bottom left subsystem is the DC to DC converter subsystem. This subsystem, though not used in our second revision of the PCB connects to the ATmega2560 to distribute the power to all of the other subsystems including; encoders, motors, sensors, LEDs, and ATmega328. The big component in what is almost the middle of the image is the ATmega2560. This subsystem communicates with the other subsystems and sends commands or receives information. The ATmega2560 subsystem sends commands to the ATmega328, LEDs, motors, and the astronomy software. The ATmega2560 subsystem receives information from the astronomy software, sensors, motor, joystick, and encoder subsystems. The part

of the schematic above the ATmega2560 are the voltage pins for the shield. The voltage pins contain a V_{CC} which is the power in from the PSU, a regulated 5V, a regulated 3.3V, and a ground connection.

Next to the voltage pins is the joystick. The joystick sends data to the ATmega2560 to control the motors in a manual way when not communicating with the astronomy software. The ATmega2560 will interpret the commands from the joystick and move the motors appropriately. To the right of the joystick is the input 24V terminal block. The terminal block will accept the input from the PSU recommended by the manufacturer for controlling the motors. This 24V input will split between the DC to DC converter and the inputs to the motors.

Below that terminal block are the IR LEDs to the sensors. The IR LEDs accept a constant 3.3V input from the ATmega2560 to constantly power the LEDs to sense for faults. Below the IR LEDs are the regular LEDs for user feedback. The LED uses are defined in a section above. These LEDs are turned on and off by the ATmega2560 for certain conditions. For example, if the sensor faults, the corresponding LED will illuminate. If the system reaches its home position, another LED will illuminate.

Moving to the top right of the integrated schematic are the differential line receivers. These receivers receive a differential input from the encoders and output a signal to the ATmega2560, which then the ATmega2560 uses to understand more about the position of the motor and adjust the commands to the motor as necessary. Below the differential line receivers are the encoder connectors. These connectors connect to the motor encoders and receive and send out data. The encoders receive a constant 5V signal and output a differential signal pair pulse train to the differential line receivers.

Below the encoder connectors are the motor connectors. These connectors connect to the motor for sending and receiving data. The motors accept inputs from the ATmega2560 to tell it where to turn and how fast. The ATmega2560 can also send a command for the motor to disconnect for removal. The motor also accepts a 24V input from the PSU from the manufacturer that works its way through the terminal block section before reaching the motor. The motor outputs a fault signal when there is an issue. This output will be sent to the ATmega2560 to process and illuminate LEDs to show there is a fault. It is important to note that there is also a fuse in between the 24V PSU and the input to the motor. The last piece of the schematic are the sensor connectors with their pull-down resistors. The sensors output a signal to the ATmega2560 when the telescope turns too far and trips the sensor.

Below in Figure 61, we have the integrated schematic design for the second and final revision of our PCB. Changes made included tying the reset button to both microcontrollers, eliminating the DC to DC converter, changing the footprints for the differential line receivers as well as the connectors for the motors, encoders,

sensors and joystick. Lastly, there was an elimination of some LEDs to make room for the larger connectors that were necessary for a robust connection.

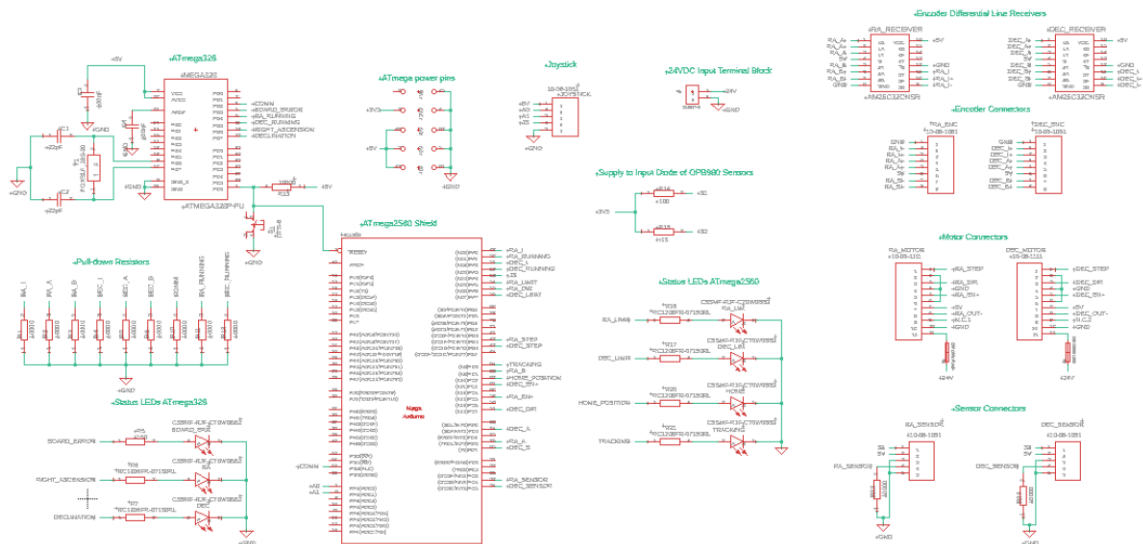


Figure 61: Integrated Schematic, Final Revision

The associated board layout for the final revision of the PCB is shown below in Figure 62. The default trace width was increased for the 24V path for the power supply to the motors, and there were mounting holes added to securely connect as a shield above the ATmega2560 development board. The LEDs, sensors, motors and encoders were all labeled on the board to aid in assembly and troubleshooting.

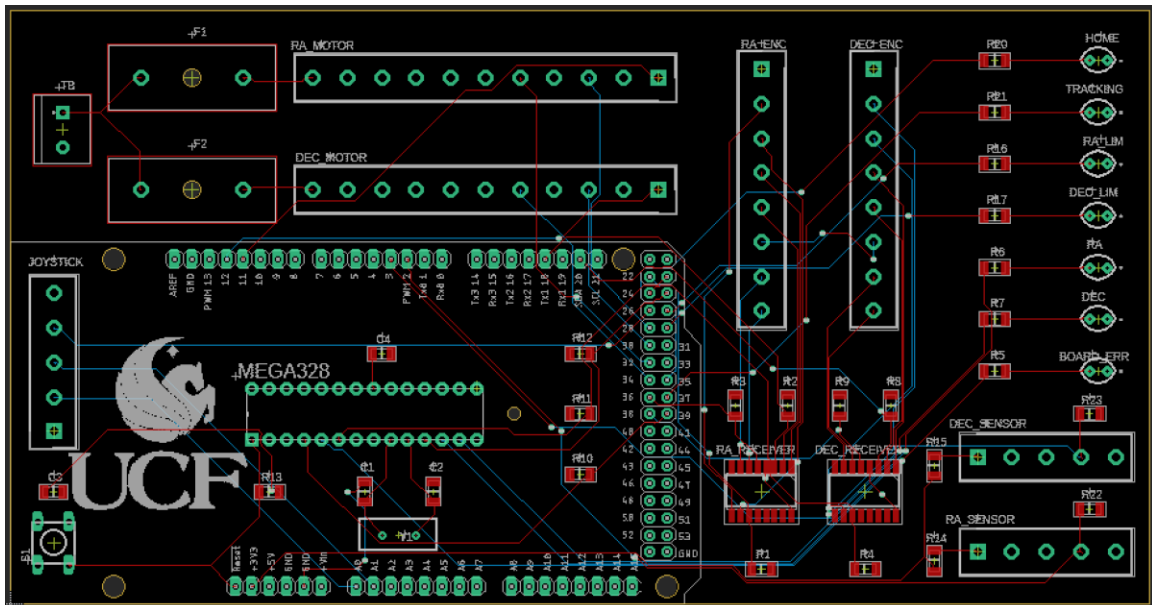


Figure 62: Board Layout

6.2 PCB Vendor and Assembly

The PCB design on this project was a two-part process. That is, the PCB itself had to be manufactured, and then the various components assembled (i.e. soldered) onto the board. Therefore, this section is split into two parts: PCB Vendor and Assembly.

6.2.1 PCB Vendor

Several vendors were under consideration for the PCB fabrication portion of this process. A brief discussion of each vendor follows. Ultimately, this decision was made based upon the quoted turn-around time and price provided by each vendor, as well as an overall determination on quality and reputation. Initially, only PCB vendors based in the United States were considered for this portion of the project, primarily due to the increased lead time associated with an overseas vendor. However, it was discovered that with the added cost of DHL shipping, the lead time could be greatly reduced from 15-20 days to 3-5 business days.

6.2.1.1 OSH Park

OSH Park is one of the most ubiquitous names in PCB fabrication. As specified by our design criteria, this is a U.S. based manufacturer. OSH Park accepts files in the KiCAD, EagleCAD or zipped Gerber format. Since our design work was done in Eagle, this compatibility was a significant advantage for us. No conversion needed be done, which could eliminate any potential errors that may arise during this process.

OSH Park offers a number of different pricing and turn-around packages. The most general package, a two-layer board with a 12-calendar day turn-around time, is priced at \$5 per square inch. For this cost, three PCBs are produced and shipped. The cost of shipping is included in the cost of the boards. In addition, they offer a "Super Swift" service, which offers a 5 business day time to ship. This service may be highly desirable if we require a second PCB as we transition into Senior Design II. Four-layer boards are also available, although we did not anticipate the need for this service.

6.2.1.2 Express PCB

Express PCB is another PCB fabrication vendor that offers boards manufactured in the U.S., again meeting this criterion of our design. The pricing scheme for Express PCB is more complex than that offered by OSH Park. Several levels of service are available and are primarily dependent upon the size of the board that is required.

First, we consider their MiniBoard Standard option. The size of this board is static – the only option is a rectangular 3.8 x 2.5-inch board. The cost is reasonable, at \$51 for three identical boards. The turn-around time is also quick, boasting a 1-day lead time for a two-layer board.

A second, more versatile option is their ProtoPro service. This service is significantly more costly, weighing in at \$169 for 4 boards. However, it also boasts increased flexibility. The size constraint here is that the board must be a rectangle that is 21 square inches or less, with the longest dimension no more than 12 inches. The lead time for this service is 2 days.

The most significant concern with this company is the requirement to use their proprietary software. Three versions are offered: ExpressPCB Classic, ExpressPCB Plus and ExpressSCH Classic. Although the software does not look overly complex to use, importing EagleCAD or Gerber files is simply not an option with this manufacturer. While their lead time is desirable, this significant drawback weighs heavily against them.

6.2.1.3 4PCB

Our next consideration for a PCB fabrication vendor is 4PCB / Advanced Circuits. Although 4PCB offers their own software package (PCB Artist), similar to Express PCB, they also accept and work with standard Gerber files. Since Gerber files are easily produced by EagleCAD, this is not necessarily a shortcoming for this vendor.

This vendor prices their boards at \$33 for a board size at a maximum of 60 square inches and a two-layer board. This pricing is made available especially to student groups. This cost provides a single PCB, which is, ultimately, all that will be required by our team. The low cost and student discount make this a somewhat attractive vendor. Their turn-around time is listed as 5 days for this service. Four-layer boards are also available, although, again, this will likely not be required in the scope of this project. The student cost for the four-layer board is \$66 for a 30 square inch board.

6.2.1.4 JLC PCB

Our last and chosen vendor for PCBs was JLC PCB. They took Gerber files as an input which allowed us to use EAGLE to design our board layout. The pricing was very inexpensive at \$2 for a board of 1-2 layers under 100x100mm. The quote for our first revision PCB was \$7.60 for five boards of a size of 120x88mm. The final cost for our second PCB was \$8.90 for five boards of a size of 168x87mm. There was an additional shipping charge of \$16.81 to guarantee our boards would arrive within 3-5 business days, however there was a discount of \$8.00 for first time users

of their service which made this cost seem worth it. In addition to the low price, the other reason why our team selected this vendor was due to its reputation from other students which had recommended this vendor to our team.

6.2.1.5 Comparison of Vendors

In Table 12 below, the different vendors considered for the PCB fabrication stage of our project are listed with each vendor's respective board size, turnaround time, file type and cost. Our team officially decided on JLC PCB as our PCB vendor, which was a decision that was made during the summer before Senior Design II.

Table 13: Comparison of PCB Vendors

	Board Size	Turn Around	File Type	Cost
OSH Park	N/A	12 days	EagleCAD	\$5 in ² (3 boards)
Express PCB	21 in ²	1 day	Proprietary	\$169 (3 boards)
4PCB	60 in ²	5 days	Gerber	\$33 (1 board)
JLCPCB	48 m ²	3-5 days	Gerber	\$8.90 (5 boards)

6.2.2 Assembly

Where available, our intention was to use surface mount parts in our design. In cases where surface mount was not available, or, such as in the case of the ATmega328, where we had existing parts available, we would use through-hole as needed. Two clear options were initially apparent for the assembly of our board: soldering the parts ourselves in the lab or employing an assembly house. As the process of self-soldering the parts is fairly self-explanatory, this section will focus on the potential use of an assembly house.

6.2.2.1 Quality Manufacturing Services

Quality Manufacturing Services (QMS) is a local assembly house based out of Lake Mary, FL. Specific pricing for assembly services is not immediately available on their website, but in the past, the company has offered assembly services to UCF senior design teams at no charge. If this service was still available, it would have made sense to move forward with QMS. Even if there was a cost associated with the service, it would have still been something that in consideration. Having a professional assembly house handle the reflow and soldering pieces of this project would easily rule out human error on our end, in terms of component assembly.

One of the key concerns regarding outsourcing the assembly of our boards is communication and readability. That is, it is imperative that the PCB silk screens accurately describe the associated components and that all files provided to the

assembly house are well documented. Additional time and scrutiny spent in this area would pay dividends in the quality of the final product that is produced.

6.2.2.2 Lockheed Surface Mount Technology Lab

Though we heavily considered going to QMS to get our boards assembled, it turned out that one of our group members had an acquaintance with a PCB assembly technician at their workplace, Lockheed Martin. The mentioned coworker offered to solder all the components for us at no charge with a two-day turnaround time. We were extremely grateful to her for offering this quality service to us. Our team designed a 3D model of our PCB to provide help in determining the proper orientation of the chips on our board. Also, a detailed parts list was provided that corresponded to the labeling of parts on our board.

6.3 Final Coding Plan

The final programming of our project took place near the end of the project as all the parts were understood and calibrated. Preliminary code was written up to test the parts and concepts, which then the results would dictate how the final result would look and work. The code that came out of this was a very fast and resource efficient program that got the objective done per requested by the user. Fortunately for our project, the hardware and tools that were used provided a solid foundation to a seamless program that got the job done in reasonable real time.

6.3.1 Information Flow

When creating a coding plan the first step that should always be considered is how to design the method and direction of the information of the parts. How each part of the project interacts with another part of the project and how each part can interact with another part is truly the molding process of the software design. The parts can only relay certain information and respond to certain information, with which the information is dictated by other parts culminating in the software controlling all this information and part integration. Our specific project was a user-controlled tracking telescope that contains motors, joysticks, optical sensors, and specially designed program all giving and receiving signals from the Arduino.

A good place to start is with how and what information we would be receiving regarding our devices. The Computer Science team that we worked with created a program that sends a signal to us telling us how to move the telescope, which in turn meant that they would be sending our Arduino a descriptive string containing the distance and direction our motors need to turn to. The motors will then turn to the requested direction and move until the position is reached sending back a signal that the movement was complete. If an optical sensor is triggered however,

the motors will stop and send back a failed signal depending on what the failure means. During all this, the Arduino will be also susceptible to move into the joystick control mode where the user can manually move the telescope. Taking all this into consideration, the summary of information flow is the computer inputs information into the Arduino which then sends out information to motors. The motors then can respond back through the Arduino to the computer completing a sort of information cycle. All while the cycle is happening, the Arduino is also listening to input commands from an optical sensor or a joystick. These two devices would send out information into the Arduino, subsequently sending out information into the motors and the computer (Figure 63).

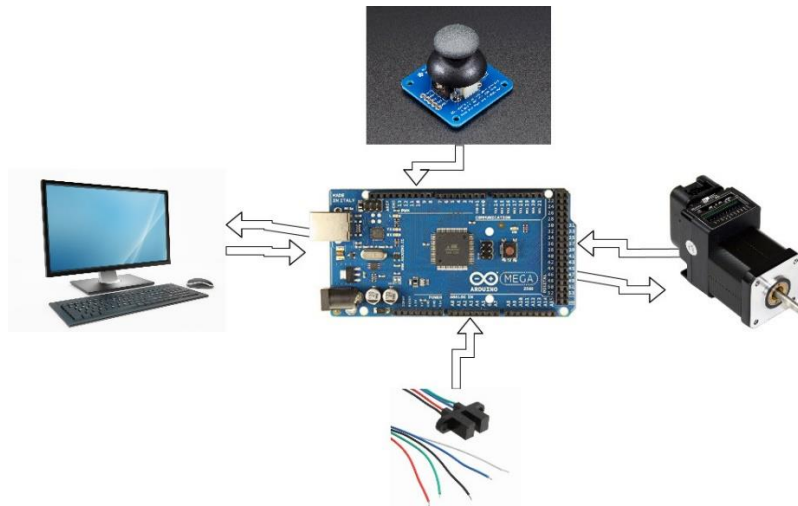


Figure 63: Direct information flow of components in the telescope system

6.3.2 Controlling the Motors

The control of the motors is simple conceptually but required a little bit more effort to execute. The motors functionality comes directly from the Arduino which will be spitting out two values. A direction to turn and at what speed. Initially the computer will be telling the Arduino some different values and it will be left up to the parser to correctly interpret the values, but the transformation of the values had to be done as well.

From the given directions and speed to travel at, the program would have to handle the calculation of the distance traveled to a corresponding frequency to be outputted. That frequency is then dictated by the calculations done with the stepper motor and its steps. Several calculations and testing had to be done to achieve this. This is because the final purpose of the motors is to move a certain amount of distance across the sky. These distances are called arc-seconds, and testing will have to be done to determine how much a step on our motor will correspond to an arc-second in the sky. Once that was found out a calculation would then have to take place to associate the speed of the motor with the number of steps the motor takes. This will relate the speed of the motor which in turn will relate the

frequency of the signal with the distance to travel across the sky. As shown before, the frequency is the speed of the motors, but this doesn't solve the whole problem. In the equation for distance traveled which is $Distance = Speed * Time$, the speed is only one part of going the required distance.

To go a certain distance the motors need speed and time. Speed has been covered with the only remaining thing as time. Using the equation above however can give us the time needed to move at a certain speed, resulting in the specified distance we would like to move. With these two elements we can set the motor to move a calculated speed with `analogWrite()` and a calculated time to reach our desired destination. The calculated speed or frequency when tracking was around 640Hz when paired with the proper step count.

Immediately there comes an interesting coding complication in the form of organizing the motors and processes. Initially controlling two motors at the same time can actually become fairly difficult if not handled properly by the method of bit-banging. Bit-banging is a method where the programmer manually sends out discrete signals to simulate a sinusoidal wave. Each motor will need a specific frequency of a square wave that controls the speed. Since the Arduino doesn't support simple multithreading capabilities which would allow us to run multiple processes at the same time, we're stuck executing code a single line at a time. This means that if we were to spend the time creating a signal via bit-banging, the program wouldn't be able to take on other tasks while this was happening unless it was an interrupt. Even then, the interrupt would end up stopping the signal for the motor causing further issues. It is possible to bit-bang out two signals at the same time in the `main loop()` function, but their frequencies would need to be delicately handled to the point of detriment. And it still wouldn't solve the problem of the program not being able to do anything else while sending out those signals.

Fortunately for Arduino users, the library comes with a built-in function as described before called `analogWrite()`. `AnalogWrite()` just continuously outputs a square wave of requested duty cycle at 490Hz. That's a positive feature at our finger tips, the issue is that we must have a higher variable frequency for the `analogWrite()`. Our `analogWrite()`'s frequency can actually be changed fortunately, by directly overriding some innate timers in the Arduino. Now the problem is almost completely solved regarding the processing management. It's almost completely solved because even though a well-defined frequency square wave can be outputted, the program will still have to wait for a specified amount of time before it shuts off the signal when the telescope reaches its destination. This is where some proper management of coding and data came into use.

The program has the ability to set and forget a square wave out of a PWM port on the Arduino, but it still will have to wait a specified amount of time before the processor can move on. If the immediate choice is used, namely the `delay()` function or `delayMicroseconds()` function, the problem will not be solved. The processor will fall asleep for the given amount of time and not perform any other

actions during this time. The method that was used to work around that is instead of using the built-in function for time delays, the internal clock on the microcontroller was used. `Millis()` is function in Arduino that just simply returns the number of milliseconds in time passed since the board began running. Our program used that to calculate the current time and duration to equal the final time we want to reach. This final time will then be passed as a return argument from the function call to be put into a variable in the main loop. A conditional if statement was checked every iteration of the loop to see if the time desired has been met. If the time desire has been met, then the if statement will call another function to stop the signal from being sent.

Conclusively, the program would then be able to effectively continuously output square waves of varying frequency that turn on and off at desired times, while still allowing the program to perform other tasks. Alternatively, the motor functions could just simply reassign a global variable value and the `loop()` function just check the global variable. However, global variables make code less portable between systems, reduce readability and sometimes cause unintended behavior in microcontrollers. To say the least, it is a good idea to not use global variables as much as possible.

A final part of the programming for motor control was the calculations for continuous tracking. The concept is that since the earth rotates, the period at which a user is looking at a celestial body will see the object move out of sight after a certain amount of time. Working in tandem with the Computer Science team and the Mechanical team, the calculations were made to be able to set a constant speed for each motor to track the celestial object until another command is received.

6.3.3 Joystick control

The joystick is a small device that the user has the ability to plug in and physically turn the telescope where they want to point it. To briefly explain, the joystick outputs two DC signals whose voltage represents the strength the user moves the controller. One signal is the x-axis and the other one is the y-axis. In spite of the signal technically be a DC signal, the actual signal effectively acted like an analog signal, changing the way we handled the input. The signal would change so often that it wouldn't ever really be steady at a single DC value for a particular amount of time. The `analogRead()` function that comes with Arduino is a function where if a value is read into an analog input port, it will use an analog to digital converter to change that signal into 1024 discrete values, outputting the value that the input signal is at. This gave us a range of discernable numbers to use when programming the analog stick.

Programming a set of 1024 values was fairly straight forward. Typically, with joysticks there is a dead zone that is programmed into the signals that allow the

joystick to move a slight amount without correlating to an action. It was useful so the user didn't have to be extremely careful and concerned with oversensitivity if we programmed in a small buffer that didn't relate to movement.

As stated before, Arduino proves us with a function called `analogRead()`. When the analog joystick is in a stable position it will be providing the port with a 2.5-volt signal translated into 512 as a discrete value. Through the process of testing we shall find a sort of dead zone range of values that the Arduino will just ignore, assumingly between 612 and 412 values. An if statement will be running to check the output of the `analogRead()` to see if it exceeds these values, and only when it does will an `analogWrite()` be sent out move the motor. During programming the frequencies, we had the option to make the input signal linearly scale with the frequency set to be outputted. This worked with the max value of 0 or 5 volts as the max frequency we wanted to give, however there was a bit of an incentive to make a joystick cap where 0.5 volts to 0 volts output the same frequencies into the motors. These were just possible safety precautions that might not need to be taken. Along with the frequency and discrete voltage scaling, the final values we used were determined by testing.

The final requirement for the joystick programming was the case of how to determine when the microcontroller hands over control of the motors from the automated input of the computer. For this there were three options possible to choose from, but they weren't too different in their outcomes to really warrant any specific choice. The first option was to use the fact that the Arduino can listen to and perform other tasks due to the motor optimization described above. A simple if statement can be made to check if the user pressed a button. Once that button is pressed, we could change a global variable to now say that the user is in manual control and will execute the controller code as written above.

Another option was to have whatever input the user puts into the analog port override and move the motors as intended. This was the least favorable choice as accidents could happen with the joystick and programming the Arduino to pick up after the manual movement is finished could become extremely messy. The last option was the use of interrupts to control the joystick activation. Interrupts are immediate and can be activated at any time allowing the signal from the user to never be missed. The interrupt will simply change a variable that will then be checked by the main loop with an if statement. No hard programming is ever done inside of the interrupt because the interrupt should be the fastest it can possibly be as to not cause any errors. The advantage of this method over the one that just has the check in the main loop is that this one has almost no chance of being missed by the program doing something else. Our final implementation however did not use an interrupt since they were all being allocated already, so we just had the main loop of our program constantly check if the button was pressed, and using debouncing, we could determine if the user intended to take the joystick out of tracking mode.

6.3.4 Parsing the Directions from the Computer Science Team

The core information that allowed us to locate and track a celestial body was coming to our Arduino via USB and the Computer Science team's program. The USB uses a serial type connection where all the bits come in sequentially into Rx line of the board. The Arduino itself handles the data intake and transformation automatically for us, and outputs bytes. As decided from meeting with the Computer Science team, their software would output a string containing the direction and distance that the motors need to travel, along with maybe some other elements. It is our job to then parse the given string and extract the actual data, while turning it into usable information. The string that was finally determined looked like this: "M-120.0-120.0". The 'M' would tell the program that it was a command to move the motors, and the following numbers were degrees in which to move both motors to.

The actual programming of this string parser came down to what the Computer Science team actually ended up submitting to our Arduino, but general coding practices were able to be made before hand. `Serial.read()` is a built-in function that reads in byte by byte the input coming from the Rx pin of the USB. Translating to code it means that a while loop will be used to iterate through each byte given to us until a specified stopping byte is reached. This byte can be determined by the CS team as some random character, or more usually a null terminator character. Since these bytes are of ASCII format, we had to use a conditional statement to look for the terminating character by associating the numerical value with it. If the default null terminator is used, our code will be looking for a 0 character to appear and then use that to exit the loop.

The loop itself would just be inputting each byte into a locally stored string and separating the bytes into different strings based on particular flags the CS team will provide to us. These flags tell us when the byte data will change from one data piece to another, allowing them to transmit and us to receive multiple types of data. How that data is being stored can be a bit of a complication. Normally in C, the use of the `malloc()` or `calloc()` function would be utilized, but those two functions rely upon an OS to operate, as stated before. Our current method will just be filling in given arrays allocated by the Arduino microcontroller. Once inside of a local string, the `atoi()` function will be of great use transforming a string byte into a numerical byte, with which we can do calculations with.

6.3.5 Feedback Control using Sensors and Interrupts

The feedback control was a very simple coding system to design. It was just a set of interrupts looking for the event to happen. Once the event under scrutiny happened, we would then see the ISR trigger and change a couple of variables which would then be checked by the main loop. Under no circumstances did we want to have delays or heavy processing work under the ISR as that could have caused some extreme and undefined behavior. If some of the feedback needs to be sent to the Computer Science team, a `Serial.write()` can instead be used to send any incident to the Computer Science team.

6.3.6 Meridian Flip and Motor Control

The Meridian Flip was anticipated to require some delicate coding and team work between the Computer Science team and our team. Though we ended up not needing to implement this flip, the plan that we initially devised is described in this section. From an informational standpoint, the Computer Science team should be able to predict when the motors reach their maximum and need a meridian flip. Another note is that the meridian flip and motors reaching their maximum aren't completely mutually exclusive. Meridian flip is more of just a special case for a need of our motors to adjust themselves. However, regarding the Computer Science's ability to deal with the event, there can be a better way of controlling the logic for our system. Our main movement would have essentially consisted of two actions. One will be the movement to a specified location, while the other will be tracking the location in the sky. Since all these movements are technically originating from the CS team, they should be able to also know when to perform a Meridian Flip. This is the more complicated and prone to accident compared to what is planned.

The plan was to use the two optical sensors to send a signal back to the computer and to the Arduino. The signal being sent contains two pieces of information disguised as one. It tells the motors that they have reached their limit in turning radius for the telescope base, and it also that a meridian flip is going to be required. This information doesn't come in two different packets but more is represented as two different things regarding hardware side and software side.

The signal must go through our Arduino which will be processed as a flag interrupt. We want this to be a flag interrupt because on a hardware side we want our motor to stop the rotation as fast as possible. If we have any bit of delay on the motors turning, then that can lead to damage done to the structure it's attached to. This is because there is a large possibility that the time it takes for the motors to react to the sensor trip will be too slow for a large, heavy moving telescope. We could have

had the telescope move at a slower speed but then it might risk not being able to properly find its location or track something because it's moving too slowly. There's even another possible problem that can arise with the centering of the telescope. If the telescope travels too far off from where the trigger was flagged, the system will not be synced up with the offset telescope position.

The proposed solution to this was to actually have the telescope flag the interrupt twice or more. The concept is to oscillate the telescope to be exactly in the optical sensor by changing the direction and speed of the respective motor, constantly slowing down and changing direction. The more this occurs the more accurate the telescope will be pointed as it will essentially be lined up with the optical sensor, raising the flag continuously. This is a positive because now we know exactly where the telescope will be after an abrupt stop. A small problem comes up but is easily fixed by Arduinos given interrupt commands. Having the telescope directly on the sensor means we know exactly where it's aimed and can get the same results after a meridian flip every single time.

To actually program this, the method that planned to use was to have the interrupt add a value to a variable. Once this variable is greater than zero, a function will run, changing the direction and speed of the corresponding motor. It then continues this until the trigger is tripped again, adding to the variable. This new number will again change the motor, causing it to trip again. This constant back and forth will happen very quickly causing the telescope to have an oscillating decay into the optical sensor. Once the given variable surpasses a particular number, we can assume that it is essentially close to being in the exact center of the optical sensor. Realistically this could only take a single oscillation to bring it to the center. The amount of oscillations just heavily varies on how fast and far the telescope flies out.

Resuming with the code, the variable that was being used will be set to zero again and the interrupt will be disabled for the period of time the meridian flip will begin to occur. Otherwise, the flag will be raised indefinitely causing an endless loop and holding up the processor. There are two immediately known ways to do this, one is with the `detachInterrupt()` function. The function would just stop allowing any interrupts to be triggered through the desired pin. The pin however would still keep sending a signal from the port of the interrupt, it's just that nothing will happen now.

The other function is `noInterrupts()`, which leads to an inferior method of controlling the telescope. `NoInterrupts()` is a system wide function that stops all interrupts from occurring. Unsurprisingly this is a bad idea to use because then our program will not be able to detect when other interrupts occur such as the other motor reaching its limit. Therefore, this function will most likely not be used in this particular area. It does however bring up one particular aspect of the code. There is a strong chance that both motors can hit their limits and set off their respective interrupts at the same time. Having them precisely connect at the same time is highly unlikely, the more plausible event that would occur is that one motor is performing its

oscillation while the other motor then trips is respective trigger. Although this seems like it could be a big problem, it logically won't really cause much of a disturbance. Since both ISRs are just changing a certain variable and then the loop() function of the Arduino is checking that variable, it means that both processes can technically be deployed at the same time. The oscillation on one motor will commence, and then immediately after the oscillation on the second motor will start as well. Since these two things rely on real-time to be completed, the Arduino should have no problem handling the changing controls between two motors and centering them in the optical sensor.

The second part of the signal will be sent through a USB serial command saying that the telescope is in a position needing a meridian flip. The meridian flip itself would take time to perform and during that time the Computer Science teams' program will be waiting for an ok signal to be sent. The ok signal will just a simple completed flag to notify the program that tracking can resume and countermeasures can be taken to make up for the lost time. Our program itself should have been able to handle a meridian flip, but there could have been the case where the CS teams' program sends us a serial signal telling it to meridian flip. If this is what happens, then our program will need to have an updated parser that is able to understand the meridian flip command. When it comes down to the actual flip, then our program will take over and adjust the motors itself.

An actual Meridian Flip, if implemented, would have just consisted of the program rotating both motors by 180 degrees. In practice this might have resulted in something a little bit extra then just simply switching the direction of the motors for a given amount of time and then continuing along. The time lost during the Meridian Flip would need to be taken into account as well as the possibility of one of the motors not needing to rotate at all.

6.3.7 Work Load Distribution

The organization and collection of logic in programming as well as placement of features had a huge impact on how project would perform. It shouldn't need to be said that how the code is made will heavily affect the time spent and frustration levels of the developers. Time for debugging and finding errors can be drastically reduced down with well-maintained code and can also even be easier to write if care is used when writing it. The focal points that will dictate how we write the code will be revolving around the main functionalities of the ATmega itself.

The setup function is one of the default given items that we used to organize and program our project. As per the name most of the initialized configuration code was placed here to start the board off as its intended. This also meant that if the reset pin is triggered from the board, then the setup function is the first one that get execute again. In our project we had mainly our pin setups and configuration to start getting the hardware to respond to the real-time and internal actions that

are about to happen. The serial commands to read and write data were also set up in the given setup function. An interesting note is that all these things didn't actually have to be set in the setup function and could have been called later on, but it was still a significantly more organized way to see the logic of the code. Not much else was placed in the setup function as much of the brains of the project came later on in the loop function and its supporting functions.

Global variables are something that is generally looked down upon in a computer science community because they tend to make code less readable, more complicated to fix, and have worse portability. Despite all that, we used global variables because of the advantage they provide. The global variables were declared outside of the setup, ISR, helper, and loop functions. Great use could come from this as we can use these variables as status variables. They'll monitor the system and be updated by anything that happens elsewhere. Since they are global variables they can be changed at any time or anywhere without us having to maintain the passage of data flow by the return values of functions. Under no circumstances would any logic or work be done in the global variable area. Doing so could cause a host of problems and bugs while also muddying the logic flow up and preventing certain work from being done.

The main loop of the function was where all the core decision actions were made, but not where the actual actions themselves were executed. This was to avoid clutter and unnecessary collection of ideas. Conditional statements are the brains of what will happen in the microcontroller, which will often be checking the global variables for status listed above. This includes the functionality of an ISR as the ISR's are generally just going to be limited to changing a global variable if required. This main loop function would also be home to the parsing logic in reading of the input data.

Another big part of the main loop was to respond to the analog input stick and configure the Arduino system to not respond to certain things. One of the big things with this design is being able to sort out how the ISR's can interrupt with minimal 'interrupting' to the actual flow of the program. The more we can exclude as many possible ways an interrupt can interfere with our program, the better. The worst case is having an interrupt disrupt a crucial task and causing the outcome of the event to mess up and desynchronize the telescope. Even though we left all the extra work to be done in the helper functions, the helper functions are just more ways of uncoupling the work to other areas. When it comes down to it, executing a function is still technically operating in the area of the main loop.

The ISR functions are going to be kept as simple as possible. To reiterate, this is because the ISR's can interrupt or cause un-warranted behavior in the system. There are several general key points to abide by when writing an ISR, which then dictate the distribution of work. The first is just a general idea of keeping it as short as possible. The second is to never put delays inside of the ISR function itself. It turns out that the delay function in Arduino monitors an internal timer to count for

a delay. However inside of a delay, the timer cannot return an updated time amount to the ISR, causing it to create undefined behavior.

Another problem with a delay inside of an ISR is that interrupts are turned off while the program is executing inside of an ISR. So, if interrupts would happen as demonstrated in the Meridian flip section, our program would not be able to detect it. The third limitation is to make variables that are only used by ISR's volatile. This more has to do with the compiler optimizations that can be made at time of building. The compiler will often look for unused variables and wasted resources to be cleaned up if it can. This directly affects an ISR as compilers will look for all the calls to a function. If it can't find a function call to function outside of the main code, then it will effectively remove that allocated resource as it sees fit. Interrupts aren't called from the main code, which is their defining characteristic, so when the ISR is called, it crashes the program and possibly causes segmentation faults because the variable needed isn't there. Assigning the keyword volatile to a variable causes it to be set specifically aside in the compiler, making sure it still exists in the final machine code.

The final note for information flow will be about the helper functions. The helper functions are just simple extra functions created to organize and space out the core logic from the mechanics of the program. It eases the time to debug, distribute workload among team members and make the code readable. An advanced use of functions can come in the form of implementing a recurrence relation, however there will probably be very little need for it. It is mainly used to help solve complicated logic or math problems, instead of the simple commands we need to create. Figure 64 which is pictured below demonstrates a realized explanation of the program. The figure shows the major blocks of information flow that go into the main loop of the program as well as the operations that will be occurring within each of the main blocks.

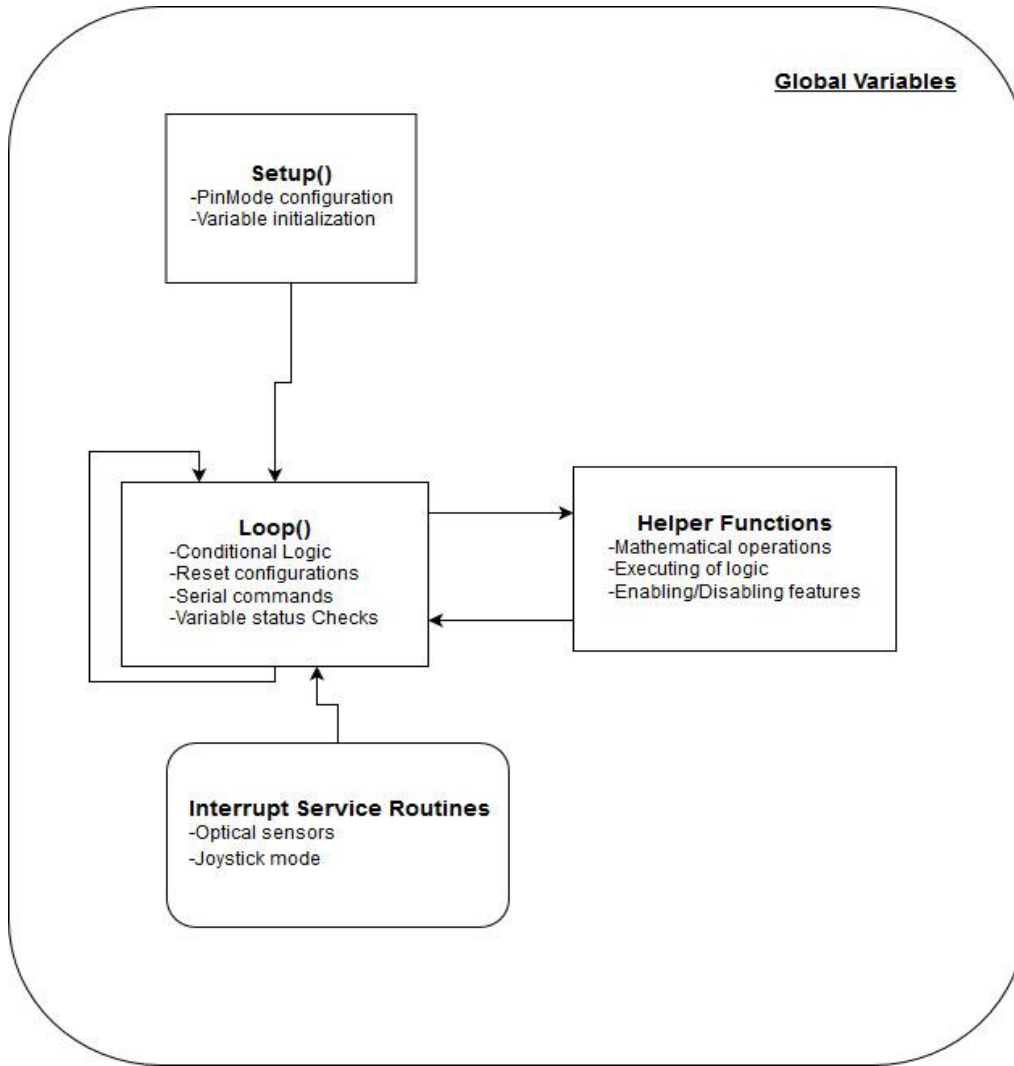


Figure 64: Final Code Logic Diagram

7. Project Prototype Testing Plan

In general, this plan covers four main areas. First, we examine the environment that we completed our testing in. That is, the senior design lab and its available equipment. Next, we discuss testing for each component and an overall plan used to test the integrated prototype. We then provide a consideration of our software and its associated capabilities, and finally conclude with a summary of our software-specific testing plan.

7.1 Hardware Test Environment

The bulk of the testing and integration for this project was performed in the University of Central Florida (UCF) Senior Design lab. Safety is a concern when designing any piece of electronic equipment, and this project is no exception. The UCF Senior Design lab is monitored 24/7 by Closed Circuit Television (CCTV) cameras and has an additional requirement of at least two personnel from a design team on site during any testing. These stipulations serve to mitigate some of the safety concerns related to electronics design work.

Much of the equipment required for the testing of this design was present in the Senior Design lab. Other equipment, to include connectors, breadboards, leads, discrete components (e.g. resistors, capacitors, etc.) could be acquired via checkout through the UCF Electrical Engineering lab manager, David Douglass. Major components that were used in the design and testing of this project are detailed below.

Tektronix AFG3022B Arbitrary/Function Generator

Although this is a discontinued model, the functionality of the AFG3022B is very robust. In addition to sinusoidal and other periodic waveforms of up to 25MHz, this device also supports 14-bit arbitrary waveforms at up to 2 GS/s. Supported periodic waveforms include: sine, square, pulse, ramp, triangle, sine, exponential rise and decay, Gaussian, Lorentz, Haversine, DC and noise. The function generator was a key element in our hardware test environment as the operation of the motor encoders was dependent on the frequency of a pulse.

Tektronix MSO4034B Mixed Signal Oscilloscope

The MSO4034B is also a discontinued model but supports features that are more than sufficient for the needs of this test environment. The oscilloscope provides four analog channels with up to 350MHz of analog bandwidth. This device will record 20M points and additionally features a sample rate of 2.5 GS/s. Although it was not needed in the scope of our project, this oscilloscope also supports FFT analysis for in-depth analysis of the frequency domain. In general, our project

functioned adequately with the standard passive voltage probes, but this model does also support active, differential and current probes if the need arises.

Keithley 2230-30-1 Triple Channel DC Power Supply

The Keithley 2230 is a triple channel DC power supply that offers two channels that vary from 0 – 30V at a maximum of 1.5A and a third channel that is able to output 6V at 5A, generally recommended for powering digital circuits. This allows for a maximum power output of 45W on the 30V channels and 30W on the 6V channel. When strictly considering the power requirements of our motors, we believed these power supplies would meet the necessary specifications. However, since the max current draw of each motor individually was 2A, powering both in addition to other components would not be possible with the 1.5A max output.

However, the lab power supply is a regulated power supply. Per the manual for our STM17R motors, it is possible that a regulated power supply will encounter a problem with regeneration. When a load is rapidly slowed from a high speed, much of the kinetic energy will be transferred back to the power supply. This, in turn, can trip overvoltage protection, if present, on the power supply.

For this reason, we moved forward with securing an additional, long-term solution with respect to the power supply. The manufacturer recommends their own model, the PS150A24. If another power supply were to be sourced, the manufacturer further recommends the installation of their RC-050 regeneration clamp. We however, purchased the PS150A24 recommended model.

Tektronix DMM4050 Digital Multimeter

The Tektronix DMM4050 bench multimeter offers significant accuracy, with 6.5-digit resolution and VDC accuracy of up to 0.0024% (rated at one year). This device is capable of detecting voltage from 100mV to 100V with up to 100nV resolution, and 100 μ A to 10A of current, with up to 100pA of resolution. In addition, resistances between 10 Ω and 1G Ω can be measured, with up to 10 $\mu\Omega$ of resolution. Additionally, temperature, continuity and diode tests are available on this model. This model does not detect capacitance, so if this functionality were to be required, an additional solution would have been needed.

Dell OptiPlex 990

The bench PC available in our test environment is the Dell OptiPlex 990. This is a reasonably recent PC and should be sufficient to support the needs of our test environment. This PC will be supplemented with personal laptops as needed to run software that is not natively available on the bench PC. The bench PC is worth noting because a number of the measurement tools available (multimeter, function

generator, etc.) support capabilities that allow them to be tied directly into a PC for more detailed analytical work. We did not need to use these advanced functions in our testing, but had we, the OptiPlex 990 would have been the first choice for these tie-ins.

Global Specialties PB-60 Externally Powered Breadboard

The bulk of early prototyping will be completed using this breadboard, and it is included here to memorialize its specifications. This board supports 1,680 tie-points and is rated for 36V and 1.5A. In all cases, these specifications were able to meet our needs. Two of these devices were secured for ease of testing.

7.2 Hardware Specific Testing

Sensors

One of the primary concerns with the sensors was whether they would output a enough voltage for the Arduino to recognize a high signal when they are tripped. Relevant entries from the datasheet are memorialized in Table 14 below. Of note is the fact that the high-level output voltage (V_{OH}) only confirms a minimum value, not a typical.

It was preferable to power the optical sensors via the Arduino's built-in 5V source, as this eliminated the need for additional DC-DC conversion circuitry and minimized the footprint of the Shield-style PCB that is being developed.

Table 14: OPB980T51Z Electrical Characteristics

Symbol	Parameter	Min	Max	Units
V_F	Forward Voltage		1.70	V
V_{CC}	Operating D.C. Supply Voltage	4.5	16	V
I_{CC}	Supply Current		12	mA
V_{OL}	Low Level Output Voltage		0.4	V
V_{OH}	High Level Output Voltage	$V_{CC} - 2.1$		V
I_{OH}	High Level Output Current		100	μ A

The implication here is that the V_{CC} constraint of a minimum of 4.5V was to be satisfied by the Arduino's 5V supply. Moreover, the minimum High-Level Output Voltage should be guaranteed at or above 2.9V ($5V - 2.1V$). The relevant portion of the ATmega2560 datasheet is summarized in Table 15 below. The Low-Level Output Voltage is capped at 0.4V maximum.

Table 15: OPB980T51Z High and Low Output Voltages

Symbol	Parameter	Condition	Min	Max	Units
V_{IL}	Input Voltage	Low $V_{CC} = 2.4V - 5.5V$	-0.5	$0.3V_{CC}$	V
V_{IH}	Input Voltage	High $V_{CC} = 2.4V - 5.5V$	$0.6V_{CC}$	$V_{CC} + 0.5$	V

Assuming a maximum V_{CC} of 5.5 (our expectation is to operate at 5V), $V_{IL(max)}$ can be calculated as 1.65V and $V_{IH(min)}$ can be calculated at 3.3V. V_{IL} is certainly satisfied with this condition but there is a question of whether V_{IH} would be sufficient. At the expected operating point of 5V V_{CC} , the requirement for V_{IH} drops to 3V minimum, although this is still slightly above the guaranteed minimum output voltage for the OPB980T51Z optical switch.

Therefore, the testing for this component is comprised of two phases. First, we connected the optical switch to the 5V Arduino power pin and took measurements of V_{OL} and V_{OH} . The second phase connects the output of the OPB980T51Z to a digital input of the Arduino and simple code is executed to determine whether or not the high and low outputs of the switch register at the correct logic level on the microcontroller. In the end, the logic level required was met and we could use the sensors as expected.

Motors

The Applied Motion STM17R-3NE – NEMA 17 motors that had been selected for this project consist of two primary systems. First is the motor and the associated drive, and the second is the integrated encoder. Our testing process will address these components individually and in concert with each other.

Before we began any testing, it is important to note that the STM17R has an internal fuse connected to its power supply. This fuse is not made to be user replicable. Therefore, any testing or operation of these motors occurred using a fast acting 2A external fuse in series with the positive power supply lead. A second consideration here is related to regeneration. When the drive is rapidly decelerated, kinetic energy is transferred to the power supply and can trip overvoltage protection; thus, shutting down the supply. We did not have any issues with the motor decelerating rapidly enough to fulfill this condition.

We first consider the operation of the motor. The STM17R supports a number of different configurations through user-selectable dip-switches to control parameters such as current, idle current, load inertia and step size. Many of the specific requirements for these variables were determined by the needs of the mechanical engineering team, but several default values were selected for initial testing. These are summarized in Table 16, included below.

Table 16: Proposed Testing Configuration for Motors

Parameter	Value
Current	100%
Idle Current	90%
Load Inertia	0-4X
Step Size	400

In brief, these parameters can be explained as follows. Current determines torque, with maximum torque available at 100%. Reduced current will concurrently reduce available torque and heat produced by the motor. Idle current can further reduce heat and impacts holding torque. Generally, 50% is sufficient, although the higher value (90%) may be required in some applications, such as supporting a vertical load. Load inertia is strictly determined by a calculation of load inertia divided by the STM17R rotor inertia (82 g-cm²), which would be determined by the mechanical engineering team.

Step size is the parameter of most interest to our electrical engineering team. The native step size of this device is 200 counts per revolution (CPR), but the motor is capable of more granular resolution through a dip-switch setting. Resolutions of between 200 and 25,600 CPR are supported. The motor is commanded through the frequency of the pulse that drives it, using the following formula:

$$Frequency (Hz) = RPS * Step Count$$

This means that, as an extreme example, a speed of 50 revolutions per second with the 25,000 setting would require a pulse frequency of 1.25MHz, which is well outside the native operating parameters of our microcontroller. Therefore, for this initial testing, we will limit the step size to 400 and test at frequencies of 5kHz and below. When specifics on gear ratio and requirements were obtained from the mechanical engineering team, we realized we had a problem when they indicated the RA drive gear ratio was 38,000:1. This would mean for slewing, our motors would have to run at 635rps which was impossible to do. They next suggested 50rps would be acceptable speed to turn the base at 28 degrees per minute, but we weren't able to support frequencies that high.

The solution was for them to replace the planetary gear system and reduce the gear ratio to 3,800:1 which is what was used during showcase. Shown below in is the RA motor with the original 38,000:1 gear ratio designed by the mechanical engineering team. The motor is in the bottom of the photo, with the planetary 100:1 drive hidden below the large gear. The white piece in the center was the sensor interrupter used to determine when the telescope had reached its home position. Important to note, was that the declination gear drive was designed to be a gear ratio of 100:1 which our microcontroller was able to support natively.

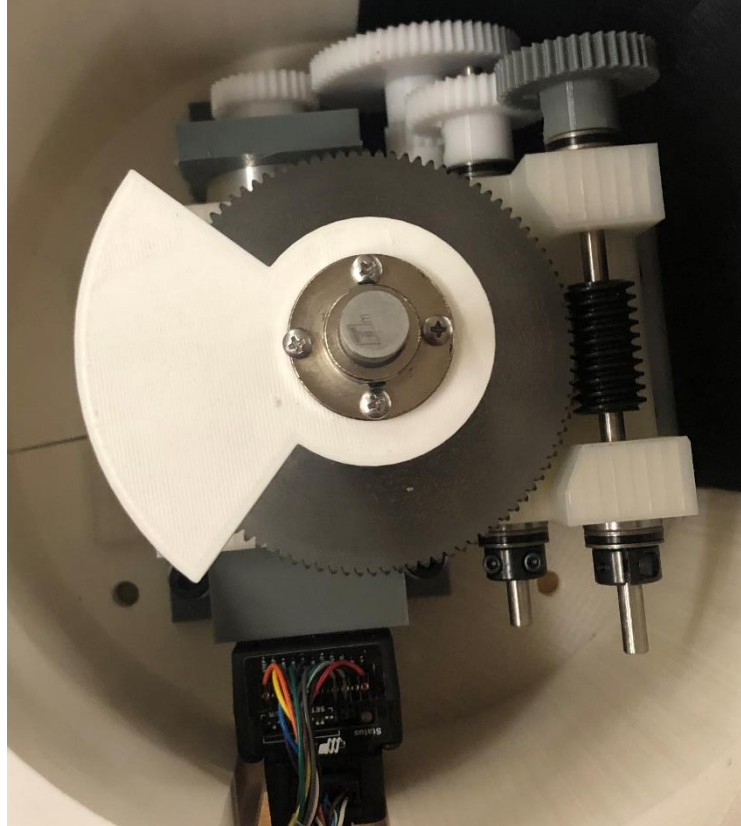


Figure 65: RA Gear Drive and Sensor Interrupter

Microcontroller

The hardware specific testing of the microcontroller will be focused on two primary areas: signals that the microcontroller is able to output and the response to signals that the microcontroller receives.

The first revision of our design involved using the ATmega2560 to drive the motors directly, as well as to communicate with the encoder. Additionally, the microcontroller was expected to drive the LEDs on the board. Inputs primarily consisted of: signals from the PC (delivered over a USB and translated into serial communication by a USB-to-serial converter), digital inputs/outputs from/to the encoder and the sensors and analog inputs from the joystick. The test plan would necessitate confirming the ability to respond to and produce each of these signals.

Digital outputs (in the form of a pulse) form one of the challenges of this project. The speed of the motors is controlled by the frequency of the pulse. As mentioned above, in an extreme case, with 25,000 steps and 500 RPS, this would necessitate a pulse frequency of 1.25MHz. Research indicated that a library called PWM.h is capable of supporting frequencies per Table 17 below [24]. Initial testing in this area verified the accuracy of this library and its ability to produce the frequencies necessary to drive the motor across a wide breadth of speeds. The Arduino Mega 2560 makes pins 2 to 13 and 44 to 46 available for PWM output.

Table 17: PWM.h Library Frequency Ranges

Timer	Frequency
Timer0	31Hz to 2MHz
Timer1 – Timer5	1Hz to 2MHz

Note: it bears mentioning here that internal Arduino time keeping functions are dictated by Timer0, so the preferred method of using this library is to call the *InitTimersSafe()* function and preserve the existing functionality of Timer0.

In addition to the outputs of the motor/encoder, it was necessary to test the inputs being received from the encoder. This testing was completed in two phases. First, we connected the output of the encoder directly to an oscilloscope, which were used to measure amplitude and frequency. In the second stage, we connected the outputs to the Arduino and used a rudimentary software package to send the amplitude and frequency received directly to the Arduino serial monitor for verification.

Ultimately, the Arduino would receive signals generated by a software package developed by the computer science team, delivered over the USB. Exact communication protocols were established with the CS team to be through USB to serial, though initial testing consisted of sending command strings through the serial monitor built into the Arduino IDE. The first revision of the proposed command string included three parts: a designation for which motor was to be controlled, total angular displacement (degrees, counts or revolutions) and a direction of rotation. However, the final string provided was instead one which provided angular displacement in degrees to both motors in one string. Direction was not needed because it was always determined with reference to the home position and thus always in the same direction.

Much of the analog testing of the board will be covered under the joystick section, but preliminary tests of the *analogRead()* functionality were performed first. The Mega2560 supports 15 analog input pins (although only 2 were use for this project) that include built-in analog to digital converters (ADCs). The ADC maps an input voltage between 0 and 5V (operating voltage of the Arduino) to a 10-bit resolution, meaning that the output ranges from 0 – 1023. Initial testing consisted of feeding precise fixed values of DC voltage into the analog input pins and confirming the output of the *analogRead()* function through the use of the Arduino serial monitor. Actual testing results are seen in Table 18 below.

Note: it is possible to change the upper end of the analog reference range by utilizing the AREF pin in conjunction with the *analogReference()* function. However, that functionality was not required for this project.

Table 18: AnalogRead() Test Results

Input (V)	analogRead()
0	0
1	204
2.5	511
4	818
5	1023

Joystick

The joystick selected for this project is a 2-axis analog joystick with select button. The select shorts to ground (i.e. go low) when it is depressed, and the X and Y axis function as 10K Ω potentiometers. The X and Y outputs are both analog out. The joystick accepts an input of any voltage up to 5V, which is where it will be operating at for our project. The X and Y axis act as potentiometers, therefore the analog output varies from 0 to 5V in a linear fashion as the potentiometer is adjusted.

Testing determined how this voltage varied with position. When the joystick is at extended to one extreme of the axis it read 0V, when it was zeroed (at resting position) it read 2.5V and when at the other extreme of the axis it read 5V. Testing this verified the voltage distribution across the working range of the joystick. In addition, voltage on the select output was measured to be 5V out under normal conditions and 0V out when depressed. Expectations made before testing are detailed here in Table 19.

Table 19: Projected Voltage Outputs for Joystick

	Lower Extreme	Center	Upper Extreme	Off	On
X/Y Axis	0V	2.5V	5V		
Select				5V	0V

The second piece of the testing was the interaction of the joystick with the analog pins on the Arduino. As referenced above, the analog voltage on these pins is indexed evenly to a value between 0 and 1023. The rest position for both axes remained between 511 to 522, thus making us incorporate a dead zone of 30 to prevent unwanted motor movement.

LEDs

The LED operation was fairly straightforward and intuitive but was still tested to confirm assumptions made were correct. Relevant electrical characteristics from the datasheet are included here as Table 20.

Table 20: LED Electrical Characteristics

Characteristics	Symbol	Condition	Unit	Typical	Max
Forward Current (max)	I_F		mA		50 ¹
Peak Forward Current²	I_{FP}		mA		200
Forward Voltage	V_F	$I_F = 20\text{mA}$	V	2.1	2.6
Reverse Current	I_R	$V_R = 5\text{V}$	μA		100

1. For long term performance the drive currents between 10mA and 30mA are recommended.
2. Pulse width ≤ 0.1 msec, duty $\leq 1/10$.

The major item for test here was to confirm that all purchased LEDs function at the expected turn-on voltage of 2.1V and not somewhere higher in their possible range. Current into the LED were be limited with a resistor in series to meet the specifications, and it was important to verify that enough voltage was provided after the voltage division to power the LED.

Integrated Testing

Although the individual components were tested and verified according to the plan set forth above, it was also necessary to test the prototype with all systems integrated as a functioning whole. The true test of this functionality would only occur once computer science team has made their completed code base available to us, which occurred near the middle of the semester, near the end of the prototype life-cycle. Similarly, the scale-model developed by the mechanical engineering team was available until well after this testing needs to have been completed. However, it was important that we be able to test the integration of all parts of the system well before that, so a brief outline of this test procedure is set forth here.

The prototype testing discussed here utilized our in-house software package (i.e. the basic version developed by our team, versus the production version developed by the computer science team). We relied on a terminal window to issue commands and receive feedback from our microcontroller. Although much of the feedback received from the system was able to be identified in real-time, the best practice here was to dump the serial monitor data into an Excel file for later analysis. Although the Arduino serial monitor is capable of some work-arounds that enable dumping data to a .CSV file, other applications, such as puTTY, have this functionality built-in. Therefore, this was the preferred terminal application for this portion of the testing.

Simple control strings were issued in the format agreed upon with the computer science team. This included an 'M' character to determine the command was sent to move the motor, and two numbers to indicate the degree of rotation in degrees for each motor. We had to make assumptions about the gear ratios being

developed by the mechanical engineering team to complete this testing, although those values were modified when the full-scale model was implemented the weekend before showcase.

The feedback requested by the computer science team included confirmation of success or failure in terms of movement of the motors to the desired position, and an indicator of position if the desired movement is not completed. Therefore, these are the signals that we will be sending back to the terminal window and capturing into an Excel file.

Without the use of the mechanical engineering team's scale-model, it was necessary to have an alternative means to test the rotation of the motors and the functionality of our optical switches (e.g. as limit switches and home position sensors). To test the rotation, the best practice here was to issue commands that necessitate a minimum degree of rotation from the motors. The simplest test with the least room for measurement error was to command the motor to complete one full rotation. In addition, quarter and half-rotations could be easily measured. This was accomplished by affixing a position indicator (i.e. a wheel with degree tick marks around it) to the shaft of the motor and establishing a point of reference from which to measure degrees moved. The correspondence of the position indicator to the markings will broadly confirm whether the motor is being commanded as intended. Figure 66 below shows the mentioned wheel with tick marks in the increments of five degrees and the optical sensor at the top as the point of reference.



Figure 66: Testing of Pointing Accuracy

Similarly, this system could also confirm positional tracking (i.e. compensation for the earth's rotation). Since the earth's rotation is fixed, a simple calculation can

determine the time it should take to rotate a quarter, half or full-turn. Since this was also be dependent on the mechanical engineering team's gear ratio, it was necessary to make an assumption here. The simplest assumption was to assume a gear ratio of less than or equal to one (although this was certainly not the gear ratio of the final product) to allow the motor to move a minimal distance for each command sequence.

The integrated testing of the motors, sensors, encoders, power supply and final PCB revision can be seen in Figure 67 below. The declination motor (pictured to right) can be mounted on top of the right ascension motor (pictured beneath the white 3D printed stand) but in this testing, they were detached to prevent unwanted wire entanglement.

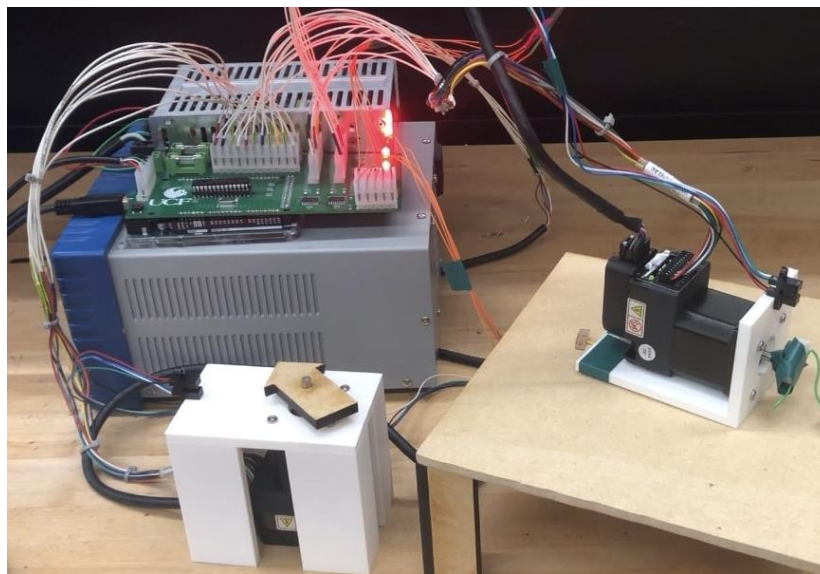


Figure 67: Integrated Testing

7.3 Software Test Environment

As the core design of this project involved the use of a Shield daughterboard designed to interface directly with the Arduino MEGA 2650, the Arduino environment naturally lent itself to our software testing. There are two environments specific to the Arduino, and both will be addressed in this section. These environments are the Arduino IDE (Integrated Development Environment) and the Arduino Web Editor. Both of these systems are available directly from the Arduino website.

Arduino IDE

The Arduino IDE is a traditional desktop IDE and it can be downloaded directly from Arduino. The main application of the desktop IDE is for offline work. This

platform was used to write short, quick test routines and other informal scripts for verification of hardware functionality. The desktop Arduino IDE is limited in some ways as compared to the Arduino Web Editor, and these differences will be elaborated below.

Arduino Web Editor

The Arduino Web Editor is part of the Arduino Create software suite. This suite provides functionality to create code, review online tutorials, perform board configuration and share project amongst collaborators. In addition, the Arduino Web Editor is an entirely online platform; therefore, the latest features are immediately available. Moreover, over 700 Arduino libraries are natively supported.

The Web Editor is able to automatically recognize any official Arduino/Genuino board, and code is backed up and saved to the cloud. The platform is available across Windows, Mac and Linux, which allows ease of adoption throughout our interdisciplinary team; however, Google Chrome is the recommended browser.

The Arduino Web Editor organizes projects as Sketches. These Sketches can include the code uploaded to the board, documentation and schematics for hardware layout. Our primary schematic design tool is Autodesk's Eagle PCB design tool; however, the ability to include informal schematics within the Sketch is a useful tool for associating breadboard testing configurations with the related code.

Arduino Serial Monitor

Both the Arduino Web Editor and IDE include built-in functionality to send and receive data to a console via the Serial Monitor. The Serial Monitor functions through the native USB port on the Arduino MEGA 2560 and can both send and receive data to and from the microcontroller. As with much of the Arduino language, this capability is abstracted to a high level. Core functionality is similar to traditional C statements for outputting to the monitor (e.g. *serial.print()*). Before initiating communication, the *serial.begin()* function had to be passed with a baud rate that agrees with the communication terminal built-in to the Arduino IDE [25].

GitHub

The Florida Space Institute (FSI) retains an enterprise GitHub repository that was used across the interdisciplinary teams to maintain communication, documentation and code. GitHub allows for version control of software via the Git platform. Although the Arduino Web Editor allows for sharing of projects, any code that would communicate across the interdisciplinary teams was housed on GitHub,

as the associated distributed version control could serve to mitigate any problems that may have otherwise arisen due to the size of the team working this project.

7.4 Software Specific Testing

In general, we had a number of specific software subsystems that would require testing, alongside an integrated test of all functionality. The integrated test was dependent upon a selection of gear ratio by the mechanical engineering team as well as the full implementation of the PC-side code by the computer science team. As that full integrated testing was not implemented until near the conclusion of this project, our team had to make initial assumptions for completeness of testing.

7.4.1 USB Input/Output

We planned to receive commands from, and transfer positioning data to, the software running on the PC. This was accomplished over the built-in Arduino USB interface. The Arduino USB interface has a built-in serial converter that translates differential USB signals into serial data, as well as perform the outbound translation. Much of our testing used the built-in Arduino Serial Monitor, but a stand-alone test of USB functionality was preferred in advance of full integration, as the serial monitor would not be a part of the integrated package. A rudimentary USB functionality check could easily be accomplished through a simple scripting language, such as Python – although our intent was to compile a more robust test platform using C.

7.4.2 Pulse Frequency Output

We selected a library that allowed the Arduino to vary the frequency of pulses on its digital I/O pins. This library is referred to as PWM.h and has associated sub-libraries. A simple functionality test would include installing these libraries onto the Arduino and sending commands to initialize the timers and vary the pulse frequency on the I/O pins. An oscilloscope was used to capture the output, determine the error and/or range of the frequency outputs and verify this functionality.

7.4.3 Tracking

One of the significant challenges for this embedded design was to include tracking functionality – that is, our motors needed to compensate for the earth's rotation, in addition to accepting positioning commands from the PC software package. As with much of our other positioning software, the final variables needed for this code will vary, depending on the gear ratio chosen by the mechanical engineering team.

However, for a simple proof-of-concept test, we chose to implement the tracking software using a single rotation. The tracking rotation should provide one full revolution each 24-hours, in accordance with the earth's rotation. However, to avoid running the motors for so long, we could interpolate this data to confirm results by using a shorter time window.

7.4.4 Encoder

The encoder that we selected is a quadrature incremental encoder. Therefore, it was essential that we included a software element that was capable of differentiating the two signals so as to determine which is leading and which is lagging. This provided data on the direction of motor rotation. In addition, the 1000-line encoder would increment a counter by +1 for each clockwise rotation and decrement the counter by -1 for each counterclockwise rotation. In conjunction with the gear ratio, this allowed us to determine the absolute position of the telescope mount with reference to its home position. This software package needed to be implemented on the Arduino and tested for all possible cases.

8. Project Operation

In this section, we discuss how to operate the resulting system with step-by-step instructions. The needed components are the modified Stellarium software package, right ascension and declination motors, optical sensors, joystick, 150W power supply, 120VAC outlet, PCB board, and interrupters to trigger the optical sensors. Once it is verified that the power supply is properly connected to the terminal block on the PCB, the steps below can be followed.

1. Plug in power supply cord to 120VAC outlet
2. Verify status LED is blinking green on both motors (next to the dipswitches)
3. Verify most recent version of code is loaded to the microcontroller
4. Power the Arduino 2560 via either USB cable or barrel jack power cable
5. Let both motors spin until both optical sensors are triggered (if interrupters are not mounted, then slide a piece of paper between sensors to manually trigger sensors)
6. Verify Home LED on PCB is lit, indicating the mount reached home position
7. Open Stellarium and select celestial body to track or enter it's coordinates manually
8. Once selected, let motors slew to the target and then verify only the right ascension motor is moving at standard tracking rate
9. If desired, to take the mount out of tracking, push down on the joystick
10. When not in tracking, move mount to desired pointing location manually via the joystick. The X-axis is for right ascension movement, and the Y-axis is for declination movement. If desired to move both motors at same time, optimal motor performance occurs when the motors are moved at the same speed.
11. Once locked on to desired target, push the joystick button to begin tracking

Figure 68 below shows the final integrated software logic flow of all components of the system. As described above and shown in the flow diagram, in the setup routine, the telescope goes to home position and then waits for input from the PC Stellarium interface, otherwise the telescope is able to be controlled with the joystick. The encoders are used to determine direction and feedback for when the pointing position is reached, and tracking has begun.

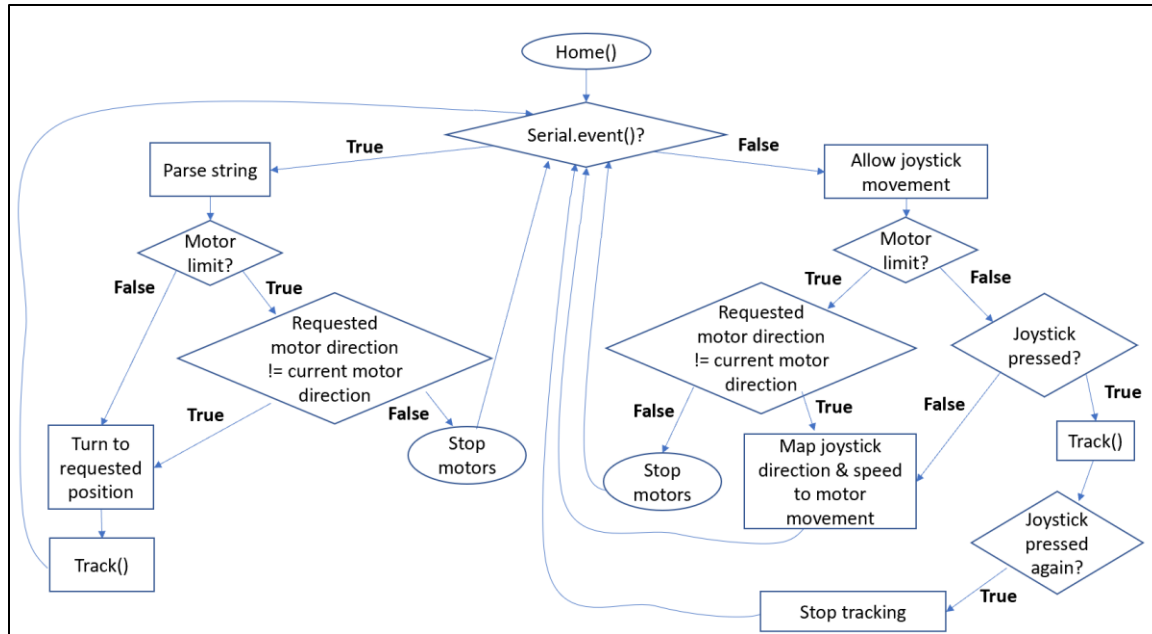


Figure 68: Final Software Logic Flow Diagram

9. Administrative Content

Being on an intradisciplinary team meant a lot more coordination and organization was involved as opposed to working with only the ECE members of a team. Due to this, Gantt charts and work distributions were key to communicating expectations both within our ECE team of four electrical engineering students as well as within the broader scope of our intradisciplinary team. This section outlines important milestones that were made aware to the other mechanical and computer science teams, as well as the project schedules and work distributions for our ECE team.

Our project director, Mike Conroy suggested to create a project schedule chart (referred to here as a Gantt Chart) and make it available to all sub-teams and customers. By alerting the other teams of when parts need to be ordered, this made the mechanical team aware that they might have to make some design choices earlier than normally expected of them. Making this schedule available to the customer enabled them to see how much progress has been made with ease.

8.1 Milestone Discussion

Our project milestones in Table 21 and Table 22 below appear to span throughout 47 weeks, however most of the members on our team were completing full-time summer internships during the summer term. The summer term was during weeks 19 through 34. It was our goal to hold bi-weekly meetings online to address the status of our project and make as much progress possible towards our Senior Design II deliverables.

Table 21: Senior Design I Project Milestones Table

Tasks	Week
Divide and assign duties	3
Divide and Conquer Document	4
Research	4-6
Divide and Conquer 2.0	7
Design	7-10
Begin writing Documentation	10
60-page draft due	12
Order Parts	13
100-page submission due	14
Finalize documentation	15
Final documentation due	16

Some of the things that our team accomplished over the summer included the completion and fine tuning our code for communication with the PC, the joystick, the sensors, the LEDs and the ATmega328. Our team also hoped to work on the encoder and motor code which we allotted five weeks to do. The remaining portion of the summer our team hoped to work on the integrated code which we had allotted over six weeks towards its completion.

In parallel to the code integration and completion, our team was able to complete testing and verification of all major components both separately and together as an integrated project. We were able to finish PCB schematic and board layout and had our first iteration of the board printed and assembled by the beginning of the fall semester. Lastly, our team hoped to work on a makeshift mount for which we could test our system while we waited on the mechanical team to deliver to us a working design. Though the makeshift mount wasn't worked on until the fall semester, we were still in a really good place going in to the fall term to allow for a second revision of the PCB to be made to account for any issues integrating our system with the code developed by the Computer Science team and the mount developed by the Mechanical Engineering team.

Table 22: Senior Design II Project Milestones Table

Tasks	Week
Build Prototype	35-39
Hardware/Software Check	40
Address Prototype Issues	41
Assemble Final Project	42-46
Test and Fine Tune	47-49
Presentation	50

For a more visual representation of our teams' milestones, specifically looking at the first semester, see Figure 69 below.

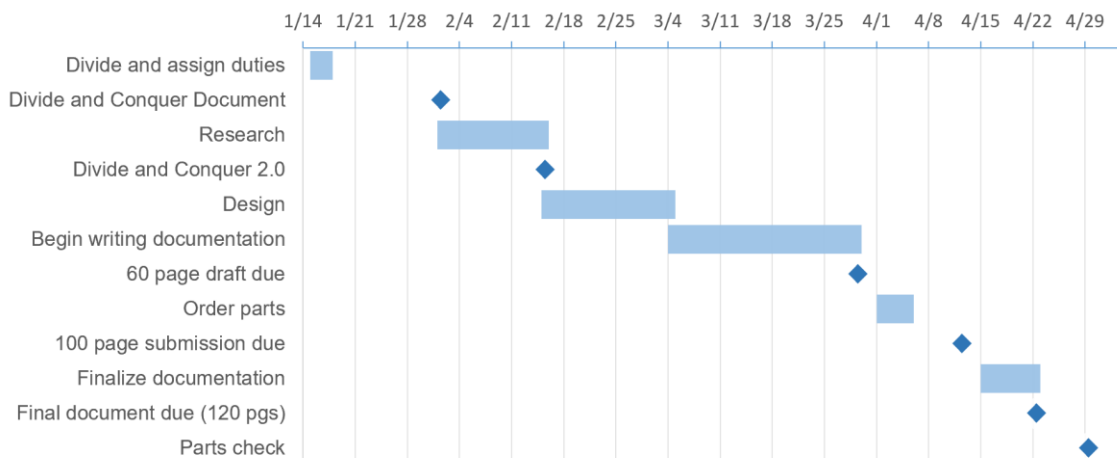


Figure 69: Project Gantt Chart, Senior Design I

It was also important to consider how each of the other teams involved in the project impacted decisions made on either side. Some decisions such as ordering parts was a collaborative effort to make sure that there was a usable interface and communications handover between each of the sub-teams comprised of mechanical engineering students and computer science students. The Figure 70 below illustrates the overall milestones for the three sub-teams over the course of Senior Design I.

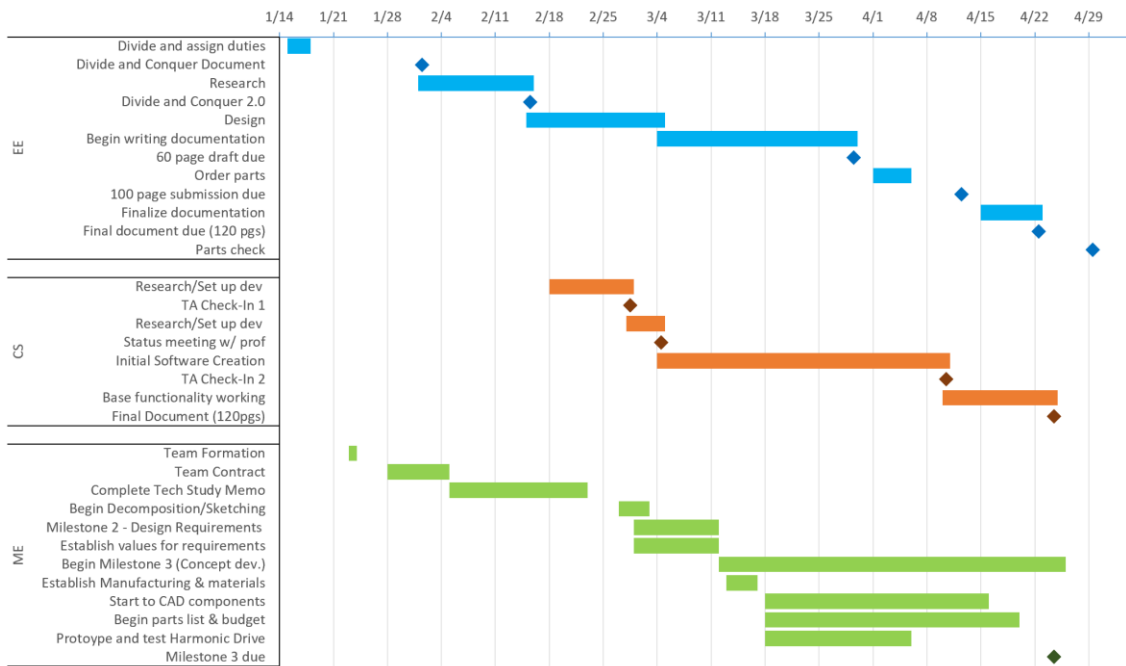


Figure 70: Integrated Team Gantt Chart

Lastly, in Figure 71 below, the schedule of testing and important milestones are shown for the fall semester. Integration with the Computer Science team began in the middle of the semester, and integration with the Mechanical Team began much later. In the end, final integration was fully completed by the morning of showcase.

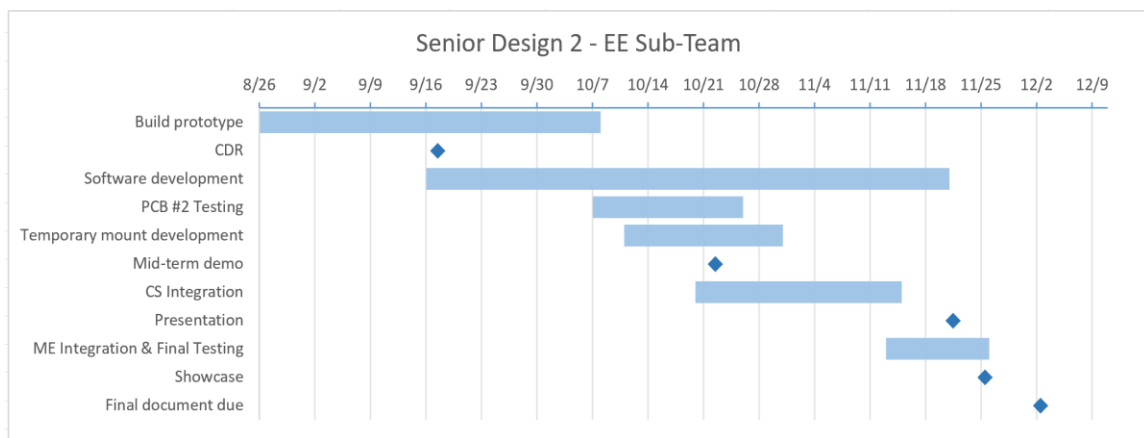


Figure 71: Project Gantt Chart, Senior Design II

8.2 Budget and Finance Discussion

The ultimate source of funding for this project was through the Florida Space Grant Consortium (FSGC); however, process of obtaining the grant was facilitated by the Florida Space Institute (FSI) and UCF's Office of Research (ORC).

The preliminary estimate for the grant was \$750 per team. However, it was later discovered that there was a maximum amount of \$1,000 available to be split between the three teams. Thankfully, the Computer Science team didn't require any funding, so the rest was split between the mechanical and electrical teams.

Table 23 lists the total cost for all components of the designed system. The actual amount spent by our team was higher since we had to purchase spare components and a second revision of PCB, as well as covering shipping costs, but to someone wishing to recreate the system without any spares needed, the total amount of \$785.85 is accurate. Though this amount was over the provided \$500 from our grant, during the design process decisions on parts were made so that the overage over the total grant allocation was minimized. Our team was willing however to split the excess cost of parts over the grant allotment, so this was not an issue.

Table 23: Project Budget

Description	Quantity	Unit Price	Extended Price
Integrated Motors with Encoders	2	\$204.00	\$408.00
Arduino Mega 2560	1	\$38.50	\$38.50
Power Supply	1	\$172.00	\$172.00
Optical Sensors	2	\$5.04	\$10.08
Joystick	1	\$5.95	\$5.95
PCB manufacturing and soldering	1	\$16.41	\$16.41
PCB components	N/A	\$105.21	\$105.21
Misc. cables, connectors, LED's, etc.	N/A	\$29.70	\$29.70
Total			\$785.85

8.3 Work Distributions

The following table (Table 24) show how the major design work was distributed among each of the team members.

Table 24: Subsystem Design Work Distribution

Subsystem Name	Primary	Secondary
PCB Development	Anthony Eubanks	Melinda Ramos
Status LEDs and Sensor	Brian Glass	Anthony Eubanks
Motors	Brian Glass	Anthony Eubanks
ATMega2560	Thomas Vilan	Melinda Ramos
Software Development	Thomas Vilan	Anthony Eubanks
Joystick	Melinda Ramos	Brian Glass
ATMega328	Anthony Eubanks	Thomas Vilan

In addition to the major subsystem design work distributions, there were also other tasks that required ownership by a selected team member. Table 25 shows which team member was majorly responsible for various administrative and overall project coordination type tasks that came with being a part of an intradisciplinary team.

Table 25: General Tasks Work Distribution

Task	Primary	Secondary
Task Delegation and Communications Lead	Anthony Eubanks	Brian Glass
FSGC Grant Paperwork Lead and Purchasing Forms Lead	Melinda Ramos	Thomas Vilan
Temporary Mount Design and Manufacturing	Melinda Ramos	Anthony Eubanks
Final Report Integration and Quality Lead	Melinda Ramos	Anthony Eubanks

8.4 Personnel

Anthony Eubanks

Anthony is a senior electrical engineering student at the University of Central Florida. He has received a previous degree in Electrical Engineering Technology from Western Piedmont in Morgantown, NC. He is currently a member of the Science Mathematics and Research for Transformation program through the Department of Defense. Upon graduation, he will be working at the U.S. Army Space and Missile Defense Command in Redstone Arsenal, developing radar and high-energy laser systems.

Brian Glass

Brian is a senior Electrical Engineering student at the University of Central Florida. Prior to attending UCF, Brian spent 10 years working for Guardian Protection

Services in Pennsylvania. He has spent two summer internships at Northrop Grumman, both in Baltimore and Orlando. Upon graduation, he has committed to working at Lockheed Martin Missiles and Fire Controls in Orlando, Florida.

Melinda Ramos

Melinda will be graduating with a bachelor's degree in Electrical Engineering and a minor in Intelligent Robotic Systems from the University of Central Florida. She is currently working for Lockheed Martin Missiles and Fire Control as an Advanced Manufacturing Technologies Intern. After graduation, she is transitioning to a full-time position in the Test Engineering department and hopes to join the Engineering Leadership Development Program Class of 2023 at Lockheed.

Thomas Vilan

Thomas is a senior Electrical Engineering student at the University of Central Florida. He is presently working at Bogen Communications. He will receive a minor in Computer Science, and his major interests lie in computer communications and MEMS fabrication.

10. Project Summary and Conclusions

The scope of this project evolved significantly since the initial proposal, based on feedback from the customer and the proprietary nature of some of the hardware and software existing in the observatory. In short, this project shifted from an immediate effort to rehabilitate the UCF observatory into an intermediate step, where a working scale-model of the equipment was implemented so that future teams are able to follow behind and continue the effort without fear of damaging the observatory's equipment or causing extended downtime. In addition, one of the aims of this project was for astronomy enthusiasts to be able to reproduce a similar set-up using this open-source design.

Through the research, design and construction of this scale-model, we have learned a great amount on stepper motor control systems. Our design essentially consisted of a short list of major components such as the stepper motors, sensors, joystick, power supply, and two different microcontrollers. After our initial prototyping efforts, we came up with a sufficient design that we believe can be easily implemented in a larger scale which would benefit the UCF observatory. Our largest challenge was to create the foundation program which would then interact with the software created by our computer science team members and could send feedback to ensure our scale-telescope meets the most important specification of 3.5 degrees in accuracy. Our implementation of the designed system was able to exceed this requirement while serving as a modifiable, open-ended solution to the observatory.

Appendices

This section consists of any references and sources used throughout the paper as well as permission emails to use copyrighted materials.

Appendix A – References

- [1] "Different Types of Telescope Mounts," Astronomy WA Partners, [Online]. Available: <http://www.astronomywa.net.au/different-types-of-telescope-mounts.html>.
- [2] "Types of Telescopes and Mounts," Optics Central, [Online]. Available: <https://www.opticscentral.com.au/types-of-telescopes-and-mounts.html>.
- [3] "What Is the Difference between an AC Motor and a DC Motor?," Ohio Electric Motors, [Online]. Available: <http://www.ohioelectricmotors.com/2015/07/what-is-the-difference-between-an-ac-motor-and-a-dc-motor>.
- [4] "What's the Difference between AC and DC," MIT School of Engineering, [Online]. Available: <https://engineering.mit.edu/engage/ask-an-engineer/whats-the-difference-between-ac-and-dc>.
- [5] "Different Types of Motors and Their Use," Design Spark, [Online]. Available: <https://www.rs-online.com/designspark/different-types-of-motors-and-their-use>.
- [6] "3 Phase Power vs Single Phase Power," OEM Panels, [Online]. Available: <http://www.oempanels.com/what-does-single-and-three-phase-power-mean>.
- [7] C. Woodford, "Hall-Effect Sensors," Explain That Stuff, [Online]. Available: <https://www.explainthatstuff.com/hall-effect-sensors.html>.
- [8] "How Do Servo Motors Work?," Jameco Electronics, [Online]. Available: <https://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html>.
- [9] "Motor Encoder Overview," Dynapar, [Online]. Available: https://www.dynapar.com/technology/encoder_basics/motor_encoders.
- [10] S. Mraz, "What's the Difference between a Motor and a Drive?," Machine Design, [Online]. Available: <https://www.machinedesign.com/motorsdrives/what-s-difference-between-motor-and-drive>.

- [11] "Classification of Power Supply and Its Different Types," EIProCus, [Online]. Available: <https://www.elprocus.com/classification-power-supply-different-types>.
- [12] "What Are Standards?," Queen's University Library, [Online]. Available: <https://guides.library.queensu.ca/c.php?g=501793&p=3436599>.
- [13] "ASME," Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/ASME>.
- [14] "Mechanical Engineering: Standards," The University of Canterbury, [Online]. Available: <https://canterbury.libguides.com/enme/standards>.
- [15] "ANSI C," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/ANSI_C.
- [16] "JTAG," Wikipedia, [Online]. Available: <https://en.wikipedia.org/wiki/JTAG>.
- [17] "NEMA ICS 16," NEMA, [Online]. Available: <https://www.nema.org/standards/securedocuments/ics16.pdf>.
- [18] "RS232 Data Interface: A Tutorial on Data Interface and Cables," ARC Electronics, [Online]. Available: <https://arcelect.com/rs232.htm>.
- [19] "AN-214 Transmission Line Drivers and Receivers for TIA/EIA Standards RS-422 and RS-423," Texas Instruments, [Online]. Available: <http://www.ti.com/lit/an/snla137a/snla137a.pdf>.
- [20] "TIA/EIA-422-B Overview," National Semiconductor, [Online]. Available: <http://rbsfm.org/Downloads/APPNOTE/RS232-485/TIA-EIA422B%20Overview.pdf>.
- [21] "RoHS Compliance," RoHS Guide, [Online]. Available: <https://www.rohsguide.com/>.
- [22] R. T. Fienberg, "Sky & Telescope," 24 October 2015. [Online]. Available: <https://www.skyandtelescope.com/observing/some-pointers-on-the-use-of-laser-pointers/>.
- [23] "Lead Soldering Safety Guidelines," Carnegie Mellon University, [Online]. Available: <https://www.cmu.edu/ehs/Laboratory-Safety/chemical-safety/documents/Lead%20Soldering%20Safety%20Guidelines.pdf>.
- [24] runnerup, "PWM Frequency Library," Arduino Forum, [Online]. Available: <http://forum.arduino.cc/index.php/topic,117425.0.html>.
- [25] "Arduino Software," Arduino - Introduction, [Online]. Available: <https://www.arduino.cc/en/Main/Software>.
- [26] "What is a Position Sensor?," AZoSensors, [Online]. Available: <https://www.azosensors.com/article.aspx?ArticleID=308>.

Appendix B – Copyright Permissions

Dear Anthony,

Thank you for your interest in Texas Instruments. We grant the permission you request in your email below.

On each copy, please provide the following credit:

Courtesy Texas Instruments
Regards,

Larry Bassuk
Senior Patent Counsel &
Copyright Counsel
Texas Instruments Incorporated
214-479-1152

From: Anthony Eubanks [mailto:eubanksaj@knights.ucf.edu]
Sent: Tuesday, April 16, 2019 7:53 PM
To: copyrightcounsel@list.ti.com - Copyright and trademark web requests (May contain non-TIers)
Subject: [Requests & questions from ti.com] [EXTERNAL] Permission Request

Hello,

I am a student studying electrical engineering at the University of Central Florida (UCF). I am working with a team of engineering students in a senior design class. In this class, we are tasked to design and build a project of our choice. For our design, we are planning to use an optical sensor with a part number of OPB980T51Z in our design. In addition to our design, we must produce documentation explaining the design and all that comes with it. When explaining our design, the operation of the optical sensor is discussed. We would like to include an image in the datasheet that shows the internal circuitry of the device. We are only intending on making one copy that will be submitted to our professor instructing this course. Please let me know if you have any further questions.

Copyright Permission Request for Figure 24 and Figure 38



Hello,


I am an electrical engineering student at the university of central florida, and I am emailing to ask that I could use the attached image for my senior design paper that will not be published.

v/r,




Anthony Eubanks

Copyright Permission Request for Figure 52

Re: Permission request to use STM17R photos for school report

 Dennis Joyce <djoyce@applied-motion.com>
To: Melinda Ramos

 You replied to this message on 11/17/2019 3:23 PM.

 Reply  Reply All  Forward

Sun 11/17/2019 3

Hi Melinda,

Yes, we grant you permission to use information, photos and any info you may require to complete your report.

All the best!
Dennis

Sent from my Verizon, Samsung Galaxy smartphone

----- Original message -----

From: Melinda Ramos <melindaramos@Knights.ucf.edu>
Date: 11/17/19 3:18 PM (GMT-05:00)
To: Dennis Joyce <djoyce@applied-motion.com>
Subject: Permission request to use STM17R photos for school report

Hello,

I am an electrical engineering student at UCF, and was wondering if I could have written permission to use some pictures from the STM17R hardware manual found here (https://www.applied-motion.com/sites/default/files/hardware-manuals/STM17R_Hardware_Manual%20920-00548-opt-.pdf) for a report which will not be published.

Copyright Permission Request for Applied Motion figures