Equatorial Mount Modification Enabling Automatic Setup and Motorized Star Tracking

A Senior Project

presented to

the Faculty of the Electrical Engineering Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

John William Burkett

March, 2010

© 2010 John William Burkett
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>4</td>
</tr>
<tr>
<td>I. Introduction</td>
<td>5</td>
</tr>
<tr>
<td>II. Background</td>
<td>6</td>
</tr>
<tr>
<td>III. Requirements</td>
<td>9</td>
</tr>
<tr>
<td>IV. Design</td>
<td>11</td>
</tr>
<tr>
<td>Microcontroller</td>
<td>11</td>
</tr>
<tr>
<td>Power Supply</td>
<td>13</td>
</tr>
<tr>
<td>Digital Keypad</td>
<td>15</td>
</tr>
<tr>
<td>Liquid Crystal Display</td>
<td>17</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>19</td>
</tr>
<tr>
<td>Compass</td>
<td>20</td>
</tr>
<tr>
<td>Global Positioning System</td>
<td>25</td>
</tr>
<tr>
<td>Servo Motor</td>
<td>26</td>
</tr>
<tr>
<td>Right Ascension Motor</td>
<td>28</td>
</tr>
<tr>
<td>I2C Communication</td>
<td>30</td>
</tr>
<tr>
<td>User Interface/Software Flow</td>
<td>31</td>
</tr>
<tr>
<td>V. Construction</td>
<td>36</td>
</tr>
<tr>
<td>VI. Testing</td>
<td>44</td>
</tr>
<tr>
<td>VII. Conclusions and Recommendations</td>
<td>48</td>
</tr>
<tr>
<td>VIII. Bibliography</td>
<td>50</td>
</tr>
<tr>
<td>IX. References</td>
<td>51</td>
</tr>
</tbody>
</table>

**Appendices**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Parts List</td>
<td>52</td>
</tr>
<tr>
<td>B. Schematic</td>
<td>53</td>
</tr>
<tr>
<td>C. ATmega328 code (Master)</td>
<td>60</td>
</tr>
<tr>
<td>D. ATmega328 code (Slave)</td>
<td>77</td>
</tr>
<tr>
<td>E. Schedule</td>
<td>85</td>
</tr>
<tr>
<td>F. Additional pictures</td>
<td>87</td>
</tr>
</tbody>
</table>
LIST OF TABLES AND FIGURES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. RMC format comma delimited data, from EM-406a datasheet</td>
<td>26</td>
</tr>
<tr>
<td>2. Stepper motor control matrix</td>
<td>28</td>
</tr>
<tr>
<td>3. Mount setup testing results</td>
<td>46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figures</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Atmel ATmega328 block diagram</td>
<td>12</td>
</tr>
<tr>
<td>2. Power Supply Circuit</td>
<td>15</td>
</tr>
<tr>
<td>3. Keypad Circuit</td>
<td>16</td>
</tr>
<tr>
<td>4. LCD Circuit</td>
<td>18</td>
</tr>
<tr>
<td>5. Lead voltage vs. Compass heading, from R1655 datasheet</td>
<td>21</td>
</tr>
<tr>
<td>6. General Lead readings vs. compass heading, from R1655 datasheet</td>
<td>23</td>
</tr>
<tr>
<td>8. Stepper motor circuit</td>
<td>29</td>
</tr>
<tr>
<td>9. LCD while the program starts up</td>
<td>31</td>
</tr>
<tr>
<td>10. LCD while leveling the mount</td>
<td>32</td>
</tr>
<tr>
<td>11. LCD while GPS signal is being acquired</td>
<td>32</td>
</tr>
<tr>
<td>12. LCD after GPS is acquired, date and time display</td>
<td>33</td>
</tr>
<tr>
<td>13. LCD after GPS is acquired, location display</td>
<td>33</td>
</tr>
<tr>
<td>14. LCD while finding north</td>
<td>34</td>
</tr>
<tr>
<td>15. LCD while in servo alignment mode</td>
<td>34</td>
</tr>
<tr>
<td>16. LCD while in tracking mode</td>
<td>35</td>
</tr>
<tr>
<td>17. LCD while in standby mode</td>
<td>35</td>
</tr>
<tr>
<td>18. Acceleration mounting configuration</td>
<td>39</td>
</tr>
<tr>
<td>19. Compass mounting configuration</td>
<td>40</td>
</tr>
<tr>
<td>20. Servo mount configuration</td>
<td>41</td>
</tr>
<tr>
<td>21. Stepper motor mount configuration</td>
<td>43</td>
</tr>
</tbody>
</table>
Acknowledgements

A special thanks to Professor DePiero for his advice and guidance on the project.
I. Introduction

Newcomers to astronomy can be overwhelmed by the amount of knowledge necessary to properly observe stars and other celestial bodies. In order to ease this steep learning curve, it was decided that a system to help with the initial setup would be created. This system is the focus of this project and is called “Equatorial Mount Modification Enabling Automatic Setup and Motorized Star Tracking” project. Through a series of sensors and motors a Celestron PowerSeeker 127EQ telescope and mount were modified to enable automatic setup of the mount to the celestial North Pole. After alignment, the system can automatically track almost any point on the celestial sphere for at least four hours. This project allows anyone the ability to easily observe the night sky.
II. Background

In order to understand some of the design decisions for this project, some basic understanding of astronomy is required. First and foremost is that the sky, for all intents and purposes, can be assumed to be a celestial sphere. This can be imagined as an infinitely large sphere that completely surrounds the earth [3]. It can also be assumed for this project, that the stars are static with respect to each other. While this is not true in reality, over the course of a night small changes such as the precession of the earth, rotation speed of the earth, and the changing distance between the stars have a smaller impact on their apparent motion than does atmospheric distortion, and thus can be ignored here [3].

For our purposes it can also be assumed that the earth is stationary in space. With this frame of reference the celestial sphere appears to be moving around us. If a line was drawn through the poles of the earth, this line would also intersect the celestial spheres apparent point of rotation. This point of rotation is known as the North or South celestial pole depending on if you are in the northern or southern hemisphere, respectively. Being able to find this point on the celestial sphere is essential to setting up an equatorial mount, as will be described later.

The location of the north celestial pole can be found with two pieces of information: the telescopes latitude and the direction of true north. Earth’s latitude is broken up into +/- 90 degree lines, with earth’s equator at 0 degrees. The celestial sphere is divided up in the same way, with the earth’s equator and the celestial
sphere’s equator on the same plane. This is called declination on the celestial sphere. This means that the point directly overhead of the observer, zenith, will have a declination equal to their latitude on the celestial sphere. For this to be true, the mount must be level, ensuring zenith is truly overhead of the mount. Doing this, the latitude angle will equal the angle of celestial pole above the horizon. Additionally, because the celestial poles lie on the same line as the earth’s north and south poles, the north celestial pole will always be due north from the telescopes position.

There are two main types of telescope mounts, equatorial mounts and altazimuth mounts. Altazimuth (altitude-azimuth) mounts, swivel 360 degrees at the base, azimuth, and up and down, altitude. This type of telescope can be easier for a beginner to understand how it moves, however it requires both azimuth and altitude to be changed over time in order to track stars. This introduces an extra variable to tracking not required by equatorial mounts. Equatorial mounts align the telescope so that its main axis is parallel to the celestial sphere’s axis. These mounts still have azimuth and altitude axes that can be moved, called right ascension (RA) and declination, respectively, however to track stars, only the RA axis needs to be changed, reducing the complexity in tracking compared to altazimuth mounts. This reduction in complexity is why an equatorial mount was selected for this project.

One area of this project that significantly reduces the longevity and accuracy of the telescope tracking is magnetic declination. Magnetic declination is the angle difference between magnetic north and true north. This is an ever changing variable that can be easily overlooked, but is essential to the accuracy of this project.
Depending on your location on earth, it can be off by as much as 180 degrees. In the continental United States, in areas of Washington and Maine, it can be off by as much as 20 degrees as of 2010 [6]. Additionally, declination changes over the course of the year. As of 2010 in San Luis Obispo, California, the declination is increasing by 0° 5’ W/year and is much worse in other parts of the country [6].

The last bit of background information is about the software development toolkit. For this project the free open source Arduino prototyping platform was used. The Arduino toolkit includes the software necessary to write, compile, and output to a microcontroller. It is C/C++ based code structure and includes extensive libraries and functions. This platform was selected based on its cost, my familiarity with C/C++, and its compatibility with the microcontroller used with this project.
III. Requirements

The idea behind this project is to enable a beginning astronomer, with little to no knowledge about constellations or the movement of the stars, to be able to view a point on the celestial sphere over the course of the night with minimal interaction. This basis introduced the requirements for this project.

The main requirement for the project was to be able to accurately track a fixed point on the celestial sphere over a period of 3 hours. This period would meet the demand of most users of this product; however a secondary goal with accuracy up to 12 hrs was also set. To meet the main requirement, the ability to accurately find true north to an accuracy of 0.5 degrees, measurement of the mount to ensure that it is level, the ability to automatically acquire latitude, and accurate operation of motors to enable proper setup and tracking are also requirements. This also requires interfacing between them all, and a way to accurately control the motors to the necessary accuracy.

This high accuracy is derived from the power of the Celestron PowerSeeker 127EQ telescope being used. This is a 127mm Newtonian Reflector telescope with a focal length of 1000mm. The power of this telescope is 329x and maximum magnification is 250x. This gives an angular field of view of 0.8 degrees, found by dividing the magnification by the telescopes power [2]. The field of view gives the amount of sky that the telescope can see at any given time. This is why a maximum accuracy less than 0.8 degrees was set for tracking and setup.
The next requirement was that there be as little user setup as possible. Ideally, the telescope could be placed on its stand, powered on, and the rest would be handled automatically. The main setup needed is for the equatorial mount to automatically position the main axis of the telescope in alignment with the axis of the celestial sphere. This was not always economically feasible in the final design, and thus some compromises to automation were made.

The next requirement was the power input source. The design was to assume that the user would be away from any power outlets, and that a 12V battery would be the power source. This is a typical power source for motorized telescopes, as the ideal telescope viewing conditions are in rural areas, away from light pollution, and most likely away from the electrical grid.

The final requirement was to use and modify an existing production telescope and mount. This requirement was set to reduce the large amount of time and money that would be spent on the construction of an equatorial mount to the required accuracy needed if built from scratch. This allows focus on areas related to electrical engineering and made completion of the final product more feasible in the allotted time. The telescope and mount used was the aforementioned Celestron PowerSeeker 127EQ. This telescope was chosen mainly due to its availability as it was in my possession before the start of the project. However, the same principles used for this project can be applied to almost any type of equatorial mount.
IV. Design

The design and construction of this product was done modularly. This allowed for an easier time if troubleshooting was required by reducing the potential number of error sources in testing. During this design phase, the requirements, cost, time and availability were all considered in the selection and use of each component.

**Microcontroller**

The main processing power behind the “Equatorial Mount Modification Enabling Automatic Setup and Motorized Star Tracking” project is the microcontroller. The microcontroller used is Atmel’s 8-bit AVR microcontroller, specifically the ATmega328p configuration. It is a low power, 28-pin PDIP, RISC based architecture IC. The operating voltage is between 1.8 and 5.5V and has a maximum clock speed of 20MHz. There are 13 digital input/output pins, six of which have pulse width modulation capability and six analog input pins which are attached to 10-bit analog/digital converters. The memory available on each chip is 32k bytes of flash memory, 1k bytes of EEPROM and 2k bytes of RAM, as taken from the ATmega datasheet. The block diagram of the IC can be seen in Figure 1.
Figure 1: Atmel ATmega328 block diagram

The ATmega328p was chosen due to my familiarity with the chipset in past use, its low power consumption, and its compatibility with the Arduino toolset. The main limitation of the IC in this project was the number of available digital input/output pins. It was decided early on that two microcontrollers would be
required, with a master/slave configuration between the two, in order to have the
required number of inputs and outputs. The use of a MUX was explored, however
was ultimately not utilized because on many occasions there is a need for the
simultaneous use of more than 14 digital pins at once. For example, the LCD and
keypad use 13 pins together, leaving one pin remaining. There was also a fear that
flash memory could become a limitation with the use of a MUX if all the code was on
one chip, but after completion of the project, this was deemed not an issue.

There were two parts external to the IC also that needed to be developed for
proper operation. The first was straight forward, the reset pin. Pin 1 is the reset pin,
and needs to be pulled up to Vcc with a resistor. A 10 k-ohm resistor was used here.

The other was the oscillator. The maximum clock speed listed in the
datasheet is 20MHz, however a 16MHz crystal oscillator was ultimately decided
upon. The 16MHz crystal was cheaper than the 20MHz and the lower frequency is
acceptable for this project, as frequent time sensitive calculations are not required.
The oscillator requires a capacitor attached between each side of it and the ground
plane. This is required to start and maintain oscillation. 18pF capacitors were used
here, as recommended in the datasheet for the oscillator.

**Power Supply**

In order to reduce complexity in the power supply as much as possible, it was
decided that everything else should be able to run off of a 5V supply rail. This
required careful selection of parts throughout, however things didn’t change much, as
five volts is a common voltage throughout industry. As stated in the requirements
section, a 12V DC input was the assumed source to the power supply, and it was
designed accordingly.

After review of the other required components, currents of up to 2.5 to 3A
needed to be provided. Most of the components require currents of less than 50mA,
but the stepper motor requires 1A per phase for maximum torque. This is the main
current draw. To achieve this high current, the LM350TFS adjustable voltage
regulator was used. It is similar to the more common LM317, however the LM350
can supply currents up to 3A, compared to the LM317’s 1.5A. To select the voltage
to be output by the regulator, the following equation from the datasheet was used:

$$V_{out} = 1.25V \times \left(1 + \frac{R_2}{R_1}\right) + I_{adj} \times R_2$$

$I_{adj}$ was assumed to be 100μA, the worst case, and $R_1$ and $R_2$ match with the resistor
labeling in Figure 2. For a $V_{out}$ of 5V and selecting $R_1$ to be 820 ohms, solving for
$R_2$ gives a value of 2309 ohms, or rounded down to a 2200 ohm resistor.

Testing of this configuration found discrepancies in the output voltage from
the design. This was due to $I_{adj}$ being approximated and the resistors measured values
not exactly equaling the labeled values. Measuring the resistors gave an $R_1$ of 809
ohms and $R_2$ of 2150 ohms, and gave an output voltage of 4.64V. To increase the
voltage to 5V, a 390ohm, 385ohm actual resistance, resistor, $R_3$, was added in series
with $R_2$ as seen in Figure 2. This increase in resistance raised the voltage to 5.05V,
an acceptable voltage for proper operation of the design.
Digital Keypad

It was determined early on that a user should be allowed to provide feedback to the microcontroller. To accomplish this, a digital keypad was designed and included in the final product. The keypad is a 12 button, three column, four row device. The buttons are 6mm single-pole/single-throw components. The keypad was connected to the main board with a 5ft long, black coiled cable with 10 wires inside. All ten wires were used, one for each row and 2 for each column. The circuit configuration in Figure 3 was used.
This keypad design came from the CPE329 reader. Each row is connected to an output pin on the microcontroller and each column is connected to an input pin on the microcontroller. Each row is also connected to Vcc through a 1.5 k-ohm pull-up resistor. The design works as follows: the microcontroller pulls all the rows but one high. The “low” row is the row being polled. If a button in that row is pressed, it will cause one of the columns and the corresponding input pins to also go low. Since the microcontroller knows which row is being polled and which column in low, the button that is being pressed can be found.

This polling is done through a function that systematically polls each row, checks the status of the input pins, and if one of them is low returns the value of the
depressed button, 1 through 12 following the numbering in Figure 3. With this configuration, depending on when the button function is called, different outcomes can be accomplished.

Liquid Crystal Display
Along with the keypad, a liquid crystal display was used for interaction between the user and the product. The most efficient way to do this is with the use of an LCD. The LCD used in this project is the RT204-1. This is a standard HD44780 LCD with four rows and 20 characters in the display. This display type was chosen because it fulfilled the 5V supply voltage requirement and, with the included backlight, was suitable for nighttime use. The one downside to this LCD was the use of white text on a blue backlight. This is not ideal for star watching, with red hues being the best for a user's night eyes. However, LCD's with red backlights are hard to find and can be quite expensive. As a compromise, a dimmer switch was added to the circuit which allowed the user to turn down the backlight when the LCD is not in use.

Figure 4 shows the pin outs for the LCD. There are a few areas of note. First, pin 5 is tied low to control the R/W pin to always be in write mode. Next, pins 11 through 14, part of the data line, are not connected. These pins are not required to be active for the LCD to function as needed. Lastly, pin 15 is the supply voltage to the backlight. Instead of connecting directly to 5V, it was connected to a 10k potentiometer in series with a 10k resistor. This allowed the voltage to the backlight to be varied between 2.5 and 5V, the voltages that would turn the backlight fully off and fully on, respectively.
Communication to the LCD is done with the liquid crystal library included with the Arduino platform. There are five main functions of that library that were used in this project. The first is the LiquidCrystal() function. It creates a variable, \textit{lcd}, of type LiquidCrystal and defines what pins on the microcontroller correspond to the RS, RW, EN, and data lines on the LCD. The rest of the functions use the \textit{lcd} variable to know which pins to use to properly interact with that LCD.

The begin() function defines the number of rows and columns in the \textit{lcd} variable, in this instance 4 and 20. The next function is clear(), which erases the characters currently displayed in the LCD. The setCursor() function places the cursor to where the next characters will be written. The column and row can be input to this function. The next two functions are write() and print(). The write function creates
one character at the cursor, while the print function can output a char, int, byte, long or string at the cursor. The final function that was used was the createChar() function. In this function you can create up to 8 custom characters at the pixel level. This function was used to create the degree (°) character on the LCD. There are many more available functions in the LCD library, however they are not used in this project.

**Accelerometer**

In order to ensure that the tripod is level, a device to measure the angle is required. This device was the Memsic MX2125 Dual-axis accelerometer. The MX2125 is capable of measuring tilt, acceleration and rotation up to +/- 3g. This device works by utilizing a heating element at the center, with a small chamber of gas above it. There are four temperature sensors placed at each edge of the gas bubble, 90 degrees apart. Using the fact that the heat will rise, tilting the device will cause some sensors to become hotter than others. This allows the tilt to be measured and output as a 100Hz PWM signal, one for each axis. The signal changes in duty cycle depending on the acceleration, with a 50% duty cycle considered 0g.

A few equations included in the MX2125 datasheet and support documents can be used to translate the pulse width into the tilt angle.

\[
g' s = \left(\frac{T1 \times 10}{10} - 500\right) \times 8
\]

\[
degrees = \arcsin \left(\frac{g' s}{1000}\right) \times \frac{180}{\pi}
\]
The first equation converts the pulse width, T1, into a signed number in milli-g’s. The second equation converts the milli-g’s into degrees. The measured g’s should always be between 0 and 1000 milli-g’s allowing the use of the arcsin() function with the 1000 as a divisor and the 180/\pi is used to convert the radian output of the arcsin() function into degrees. These functions can be used for both the x and y axes as they are both in the plane that has the force of gravity normal to it. This means that when the accelerometer is flat the pulse width T1 should be 5000 us, giving an angle of 0 degrees in both axes.

To measure the pulse width of the accelerometer the pulseIn() function of the Arduino toolset was used. This function returns the length, in microseconds, that a pulse width is HIGH or LOW on a particular pin. In this case while the pulse width is HIGH the length is measured. This pulse length is then translated into the tilt angle through the aforementioned equations.

**Compass**

In order to find the direction of north, a magnetic compass was used. The majority of compasses on the market are high cost and relatively low accuracy. In order to achieve the required accuracy and maintain a reasonable cost, the Robson R1655 analog hall-effect sensor was used. The sensor requires a 5V input and will measure the outside flux fields emitted by the earth’s magnetic field. It outputs two analog voltages between 1.8 and 3.2V, with some variation from sensor to sensor. These outputs are read by the 10-bit ADC built-in to the ATmega328 microcontroller. As seen in Figure 5, if these two voltages were plotted against the compass heading,
two sinusoidal curves would be seen, where one curve is shifted 90 degrees from the other.

![Figure 5: Lead voltage vs. Compass heading, from R1655 datasheet](image)

Initial testing of the compass gave similar voltage outputs from the data in Figure 5 taken from the datasheet, but different maxima, minima, and curve intersection points were measured. Testing found the maxima and minima of curve A to be 3.124V, 1.885V, and curve B was 3.140V and 1.875V. The intersection of the two occurred at 2.93V and 2.07V. Using this information, the voltages of A and B can be translated into a direction with sensitivity of up to 0.5 degrees/bit. Many of these equations were provided with the compass in the datasheet and the supplemental application notes by Ed Cannady [1].

\[
\frac{5V}{1024 \text{bits}} = 0.004883 \approx 0.005 \text{V/bit}
\]

\[
\frac{2.93V}{0.005 \text{V/bit}} = 600 \text{bits} \quad \text{and} \quad \frac{2.07V}{0.005 \text{V/bit}} = 424 \text{bits}
\]
This means that the crossing of the two curves happened at bit 424 and bit 600 in a 1024 bit A/D converter. From these numbers we can find the bit change per degree.

\[
\frac{90^\circ}{600 \text{ bits} - 424 \text{ bits}} = 0.511 \approx 0.5^\circ/\text{bit}
\]

This ensures that the required accuracy can be fulfilled by the 10bit ADC, however this also means that both curves need to be utilized in order to get this degree of accuracy.

By knowing the voltages of A and B with respect to each other and to their crossing points, the plot in Figure 6, taken from the datasheet, can be divided into four discrete sections, with the dividing lines placed where the curves traverse the upper and lower crossing lines. With these division set, the same general equations can be used for each section. For section 1, 0 to 90 degrees:

\[
\text{scale} = (((9000/(\text{upperLimit} - \text{lowerLimit})) \times 10)/4)
\]

\[
\text{Heading in degrees} = (((\text{upperLimit} - \text{curveA}) \times \text{scale})/10 \times 4)/10
\]

The scale variable can be used for all four sections. The multiplicative factors are there to reduce the size of the scale variable and to prevent the potential for an overflow or rollover condition. These are then added back into the next equation to get back to the correct decimal point. The following equations are used for the other three sections:

\[
\text{Heading in degrees} = (((\text{upperLimit} - \text{curveB}) \times \text{scale})/10 \times 4)/10
\]

\[
\text{Heading in degrees} = (((\text{curveA} - \text{lowerLimit}) \times \text{scale})/10 \times 4)/10
\]

\[
\text{Heading in degrees} = (((\text{curveB} - \text{lowerLimit}) \times \text{scale})/10 \times 4)/10
\]
These are the 90-180°, 180-270° and 270-360° sections, respectively. A scaling factor of 90, 180 and 270 must also be added to the heading for the proper reading.

![Graph showing Upper and Lower crossing lines vs. Heading](image)

Figure 6: General Lead readings vs. compass heading, from R1655 datasheet

Due to the variation in magnetic flux that can be measured from site to site, caused by changes in terrain, surrounding material, etc, it is possible that the crossing points that are in the code upon startup do not reflect the actual crossing points. Therefore code for automatic calibration was added. This code is invisible to the user but works by comparing both curves to each other whenever a reading is taken. If both outputs are the same, the upper or lower limit, depending on if the value is greater than or less than 512 bits, will be updated and the scale variable is recalculated. To ensure proper calibration of the compass it is best to slowly turn the mount 360° before turning the stand north, allowing scale to be updated if needed.

The last area of consideration required for the compass system is related to the magnetic declination. As described in the background section, this can cause the compass to be up to 20° off in the continental United States. To compensate for this, a
look-up table is utilized. The continental United States is divided up into squares whose boundaries are the latitude and longitude lines. The declination in each of these squares is stored in arrays. After the GPS has acquired the signal, the proper square can be found, and the associated magnetic declination, an example for 2004 is seen in Figure 7, can be added or subtracted from the compass reading. This provides the user with the direction of true north for proper alignment of the mount.

One downside to this is the fact that the magnetic declination is constantly moving and can change by several degrees over the course of a few years. This movement can’t be accurately predicted beyond a few years and requires significant processing power for a microcontroller to calculate. The recommended solution is a yearly or biyearly update to the code. At each update, the magnetic declination would be revised to the current declination. Only one ATmega would need to be updated in each project and in that IC, only the declination arrays would need to be changed.
This would be a relatively quick, infrequent update for the user while maintaining the required accuracy of the device.

**Global Position System**

Another important aspect of being able to calibrate the telescope is its current location on earth. The most common way to do this is with GPS. GPS is a free, U.S. government provided radionavigation system that provides time and positional data. This information is provided through 24 space-based satellites upon fixed orbits [5]. With these known orbits, a GPS module can determine the user’s location and provide GPS time.

The GPS module used in this project is the GlobalSat Technology Corporation’s EM-406a device. This module provides position accuracy up to 10 meters, a GPS time accuracy of 1 microsecond and has a high sensitivity of -159dBm. It includes the necessary hardware to receive the signals from the GPS network and the ability to translate that information into useable data. It provides this data through transmit and receive wires connected to the Arduino. This communication is transmitted using a TTL serial line at a 4800 bps baud rate.

The module sends the GPS data in several National Marine Electronics Association (NMEA) formats: GGA, GSA, GSV, and RMC. These formats arrange the GPS data into a standard, comma delimited format. The format used in this design was the RMC format, which sends the data in the order seen in Table 1, giving the necessary time, date, and position data.
Since the GPS module sends all of the different data formats, the others must be ignored. To do this, the microcontroller reads in each line and if the line does not start with “$GPRMC” then it is ignored. If the line does start with “$GPRMC”, then the data that follows can be formatted and placed into the proper variables. A function that reads the incoming data and looks for commas is used to know when to place the data into the next variable in the order seen in Table 1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message ID</td>
<td>$GPRMC</td>
<td></td>
<td>RMC protocol header</td>
</tr>
<tr>
<td>UTC Time</td>
<td>161229.487</td>
<td>hhmmss.sss</td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td>A</td>
<td>A=data valid or V=data not valid</td>
<td></td>
</tr>
<tr>
<td>Latitude</td>
<td>3723.2475</td>
<td>ddmm.mmmm</td>
<td></td>
</tr>
<tr>
<td>N/S Indicator</td>
<td>N</td>
<td>N=north or S=south</td>
<td></td>
</tr>
<tr>
<td>Longitude</td>
<td>12158.3416</td>
<td>dddmm.mmmm</td>
<td></td>
</tr>
<tr>
<td>E/W Indicator</td>
<td>W</td>
<td>E=east or W=west</td>
<td></td>
</tr>
<tr>
<td>Speed Over Ground</td>
<td>0.13</td>
<td>knots</td>
<td></td>
</tr>
<tr>
<td>Course Over Ground</td>
<td>309.62</td>
<td>degrees</td>
<td>True</td>
</tr>
<tr>
<td>Date</td>
<td>120598</td>
<td>ddmmyy</td>
<td></td>
</tr>
<tr>
<td>Magnetic Variation</td>
<td>2</td>
<td>degrees</td>
<td>E=east or W=west</td>
</tr>
<tr>
<td>Checksum</td>
<td>*10</td>
<td></td>
<td>End of message termination</td>
</tr>
</tbody>
</table>

Table 1: RMC format comma delimited data, from EM-406a datasheet

Servo Motor

A servo motor is a device that translates a PWM signal into a rotational angle. A servo is composed of a motor, gears, and control circuitry. The input pulse signal is typically between 1ms and 2ms, rotating the shaft between -90 and 90 degrees. Feedback internal to the servo keeps the shaft at the proper angle as long as power is supplied and the PWM signal is still being sent. Servos are typically high torque
devices that consume little power when not moving [4]. It was decided that because of these advantages a servo motor would be used to set the latitude angle on the equatorial mount.

Due to the heavy load that needs to be moved, a very high torque motor was required. Additionally, due to the required high accuracy, a high precision servo was also needed. Therefore it was decided that the Hitech HS-7950TH digital servo would be used. This servo operates at the required 5V and responds to pulse widths between 760 and 2250 microseconds over a 150 degree range. This meant that every 10 us increase in pulse width increased the servo by about 1 degree. The servo pulse width is calculated using the latitude read in by the GPS. In order to prevent any sudden movements in the telescope, preventing any unintended accidental shifting of the mount, the servo pulse width is slowly increased, at a rate of 2 °/sec. The pulse width to the servo is handled by a function in the Arduino toolkit called writeMicroseconds(). This function takes a value in microseconds and sends a pulse with that value to the servo. This will happen until another value is input to the function or the servo is removed.

In order to account for any potential inaccuracies that could occur, discrepancies in the servo, the pulse width, or anything else, the ability for the user to adjust the angle was included. By pressing 2 or 8, up or down respectively, the user is able to change the pulse width by 10 us, and thus up or down about a degree to hone in on the celestial pole if needed.
Right Ascension Motor

A high precision motor was required to change the right ascension of the equatorial mount. A full rotation of the screw on the mount translated to about 15 minutes of RA, with every minute of RA translating to the 15 minutes of rotation of the celestial sphere. Here, as with the servo, a high torque, high accuracy motor is required to turn the heavy telescope. In order to meet these requirements the 23HX18D10B stepper motor was selected. This motor has 200 steps, 1.8° step angle, per rotation and is capable of rotating a 20lb load. However, this motor also requires 1A/phase, consuming most of the current this design requires.

To rotate a stepper motor the sequence in Table 2 was used. Progressing down the table turns the motor one way, while progressing up rotates the motor the other way. In this project, the former is used. To control the current flow through the motor, an H-bridge is utilized. The H-bridge uses a series of switches that allow current to flow into the motor in one direction only. For this project the L293D H-bridge IC was used. This IC was selected due to its ability to handle the high currents required by the motor.

<table>
<thead>
<tr>
<th>Step</th>
<th>Wire 1</th>
<th>Wire 2</th>
<th>Wire 3</th>
<th>Wire 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>2</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>3</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>4</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2: Stepper motor control matrix

As seen in table 2, wires 1 and 2 are inverses of each other, as are 3 and 4. This allowed for a transistor array to be used to invert wire 1 and 3 and reduce the
number of pins connected to the ATmega328 from four to two, to control the motor. The circuit in Figure 8 was used with a CA3096CE BJT array to provide the transistors.

![Stepper motor circuit](image)

Figure 8: Stepper motor circuit

Calculating the time between steps was done as follows:

\[
\text{Time per step} = \frac{15 \text{ mins} \times 60 \text{ sec/min}}{200 \text{ steps}} = 4.5 \text{ sec/step}
\]

A delay of 4.5 seconds was set between step increments. Also, as with the servo, it was decided that the user should be able to increase or decrease the stepper motor. Pressing 4 or 6 on the keypad will increase or decrease the stepper motor as required by the user.
I2C communication

Due to the use of a master and slave microcontroller configuration, communication between the two is necessary. The Arduino toolset includes an I2C, or Inter-Integrated Circuit, library which the ATmega328 supports. This communication type requires the use of two wires, the serial data line (SDL) and a serial clock (SCL) line, both connected to pull-up resistors. These lines are connected to analog pins on the master and slave devices. These devices are given designations or addresses so that data is sent to the proper device if multiple slaves are used.

The included Arduino library features a number of functions for use with I2C communication. The first function is beginTransmission(). This function determines where the data that follows will be sent. This data is queued with the send() function. This function sends a byte or a string of characters. This property of the function causes some issues here. In order to send a number greater than 255, not as a character, some creative engineering was required. This either required calculations to be moved from the master to the slave, or vice versa, or the number sent needed to be reduced by some divisor, which would then need to be added back on after being received by the slave. After all the data to be send is queued, the endTransmission() function is used to let the devices know that no more data will be queued in this transmission and then it actually sends the bytes.

On the receiving end, the onReceive() function is an interrupt which calls the specified function. That function, called receiveEvent(), determines what to do with the data that is about to be received. This information is collected with the receive() function. This function collects one byte at a time. The total number of bytes to be
received is determined by the first byte. This allows only the necessary data to be sent when needed, reducing the number of clock cycles used to transmit data.

Through a series of if() statements the received data can be placed in the proper variables and the slave can be put in the proper display mode.

User Interface/Software Flow

In order to interact with the user and provide useful data for proper mount alignment, the LCD and keypad are used to step the user through the process. After the power is applied, the microcontrollers are initialized and the LCD displays what is shown in Figure 9 to let them know the device has turned on. The next step is to ensure the mount is level. The text format is that seen in Figure 10. The LCD is updated as the mount is moved and provides immediate feedback to the user. Once the display reads 0 degrees for both axes, the mount is level and the user hits “5” on the keypad to go to the next step.

Figure 9: LCD while the program starts up
Next up is the GPS section. This requires the user to wait until a GPS signal lock has been acquired. While waiting for signal acquisition, the LCD outputs the text in Figure 11. The screen will automatically update to the screen in Figure 12 when the GPS signal has been acquired. The time for this to happen can vary depending on several factors like weather and location, but according to the GPS datasheet, should take an average of 42 seconds under cold start conditions. As seen in Figure 12, the time and date are displayed until the user presses button 5.
After the button has been pressed, the LCD will display the current latitude and longitude as seen in Figure 13. As before, this will be displayed until the user presses button 5.

The next step is to point the mount to true north. This is done by displaying the current heading as seen in Figure 14 in real-time. The user turns the base until the display reads 0 degrees at which point the mount is pointed north. This is displayed until the user presses 5.
With these steps, the mount is properly aligned and the servo motor will move the mount to the angle for proper alignment. The LCD notifies the user that pressing 2 or 8 will move the mount for small adjustments if necessary as seen in Figure 15. This will be displayed until the user presses 5 on the keypad signaling the mount is aligned.

With the mount now properly aligned, the program goes into tracking mode. The LCD notifies the user of this and displays the screen in Figure 16. While in tracking mode, the stepper motor will move the RA axis, tracking an object on the
celestial sphere. This will happen until the user presses 5, at which time the program will exit tracking mode and enter standby mode. In standby mode the stepper motor will stop advancing and allows the user to point the telescope in a different location. The LCD will show the current date and time, as seen in Figure 17, and wait for the user to press 5. When the user presses 5, the program will go back into tracking mode and the process between tracking and standby will be repeated until the unit is powered off.

Figure 16: LCD while in tracking mode

Figure 17: LCD while in standby mode
V. Construction

Construction, as with the design, was done modularly. The first aspect to be built was the power supply. After some testing with the required output current, it was discovered that the voltage regulator chip could become quite hot, and as the temperature increased, the current output would decrease by as much as 500mA. In order to cool off the chip, an aluminum heat sink was attached to the back and a 12V cooling fan was added and aimed at the heat sink. This cooling method allowed the voltage regulator to properly cool off, allowing the necessary current to be provided.

The next section to be constructed was the LCD. This was done on the Arduino Duemilanove development board. This board provides the circuitry necessary to program the ATmega328 along with the other power and oscillator circuits required to operate it. This cut down the number of causes of any potential errors when troubleshooting. After connecting the LCD, a Hello Word program was run successfully on the LCD, showing it could be properly manipulated as needed. With the LCD working, the ATmega328 was moved off the development board and the Hello Word program was run again utilizing the new power supply and the designed oscillator circuitry. With this successfully running, the LCD could be used as an output to the user to give any error codes during construction and testing of the following modules. The development board was still used to flash the ATmega328 chips, but was not used for operation.
The next section to be built was the keypad. This would allow further user interaction and manipulation of the development code. Using the circuit in Figure 3, SPST buttons were soldered onto a 2in by 4in prototyping board. The proper connections were made with wire and then the 10 wire cord was connected between the keypad and the ATmega328. The LCD was used to ensure that the keypad and corresponding code was correctly working. The only limitation of the programming code is seen if multiple buttons are pressed. When more than one button is pressed, the button with the lowest value would be returned regardless of what button was pressed first. This could be considered an issue but was deemed an acceptable limitation since, under the current configuration, circumstances that require multiple simultaneous button presses, or the pressing of multiple buttons rapidly is not required.

After the keypad was constructed, the I2C wire communication code was developed. No large issues were seen here. The code was developed with the LCD on the slave device, and the keypad on the master. The test code was written so that when a button was pressed on the keypad, a number would be passed to the slave through I2C, were it would then be processed and displayed on the LCD.

Construction of the GPS module was next. This section required the most development and troubleshooting. The GPS module includes an LED onboard that turns on when power is applied and will blink when the GPS signal is acquired. This was a way to tell early on if the code was working correctly and helped with
troubleshooting. The LCD was used to ensure that the GPS data was being parsed correctly and that it was being placed into the correct variables.

The accelerometer was the next module to be built. Again the LCD was used during development and testing, outputting the angle. This angle was checked against a calibrated digital inclinometer and the angle was confirmed to an accuracy of 0.5 degrees. Placement of the accelerometer on the mount required two special considerations. First, the accelerometer needed to be attached to a place that would be indicative of the angle the mount was at. This was done by placing a 1ft by 1ft piece of ¼ in wood between the tripod and the mount. The flattest board available was selected and, to ensure that the accelerometer was on the same plane, it was placed as close to the mount as possible. This ensured that any small curvature in the wood would not introduce errors that would be more apparent at the edges of the board. The board was used to help with compass placement, explained later, and because the majority of the surfaces on the mount are curved. These curved surfaces significantly increased the difficulty for mounting.

The second consideration for the accelerometer was the actual attachment of the accelerometer to the board. To ensure that the attachment is perfectly flat, three flat topped screws, about 120 degrees apart from each other, were used to hold it down. The screws were adjusted until the sensor displayed that it was perfectly flat. The screws were then held in place with a small amount of epoxy. Wires were then connected between the accelerometer to the main board containing the other modules as seen in Figure 18.
The next module to be constructed was the compass. First the compass code was developed and then checked against a magnetic compass. This allowed for ease in troubleshooting by temporarily ignoring the declination variable. After that was working, the declination code was added in and connected to the GPS. The code selected the proper array value and compensated the magnetic reading to magnetic north. This was done outside using the North Star to find true north. After the compass was working correctly, careful consideration when attaching to the mount was required. First, it was important to move the sensor as far away from the large, metal mount as possible to prevent the metal in the mount from altering earth’s magnetic flux, throwing off the sensors readings. To do this, the compass was placed on one of the corners of the base board as far from the mount as possible. The sensor is six inches from any surrounding metal in this location and can be seen in Figure 19.
Testing showed that this distance was sufficient for the sensor to take measurements without introducing errors.

![Figure 19: Compass mounting configuration](image)

The other placement concern was ensuring that the compass module was orientated the correct way, so that north on the compass lines up with the desired direction of the mount. This was done by facing the mount to north and then rotating the sensor head so that the output read 0 degrees. This was then placed on the corner at the correct orientation. Had any errors resulted, it would have possible to introduce a software fix to the compass, altering the angle, similar to the way that the magnetic declination is compensated for. However this was ultimately not required. A perfectly flat compass is not required like the accelerometer because the datasheet states that it can properly operate at angles up to 12 degrees. Finally, wires were then connected between the compass and the main board containing the other modules.
The final two modules to be developed were the servo and right ascension motors. The servo required little troubleshooting and, with the Arduino toolset included functions, moved properly. The main construction here was the attachment of the servo to the mount as seen in Figure 20. The mount features a bolt that is at the center of the axis that rotates to align the mount to the latitude angle. This bolt was attached to the servo with high strength epoxy. The servo was then attached, through a series of brackets, to the base board. This prevents the servo from turning and the mount being stationary, as would tend to happen due to the lower weight of the servo.
The final module is the stepper motor, some aspects of which created the most issues. Circuit construction, following Figure 8, was straightforward and no issues were encountered. The main problems were encountered when connecting the motor to the mount. The motor needs to move with the latitude angle of the mount while staying continuously attached to the RA screw. To solve this, the motor was mounted to a 3in by 5in piece of wood. The other side of the board was attached to the mount with the same bolt attached to the servo. This allows the wood and stepper motor to rotate with the mount. This setup can be seen in more detail in Figure 21. The motor was then attached to the RA screw with high-strength epoxy. This connection failed more than once however under this method. The final attachment design utilized three thin metal rods to help secure the motor to the RA screw shaft. The rods were covered in epoxy and attached between the motor and the RA screw parallel with the shafts. The rods add surface area for the epoxy to attach to while helping with reinforcement, increasing the attachment strength. After drying, the whole connection was wrapped in electrical tape. This was to help hold everything together and, by letting some of the tape stick up, helped with testing of the motor by easing observation of shaft rotation.
The last bit of construction was putting the entire system together. This is when errors began to be introduced into the accelerometer data. Every few seconds, a false reading would be given. This was caused by a number of factors, namely the servo motor and the stepper motor modules. The fast changes in the digital signal of these motors seemed to cause errors in the pulse width of the accelerometer. To combat this, any pulse width less than 2500 microseconds was ignored. This seemed to remove the errors introduced by the motors from being output to the user.

No other major changes were made here, but a toggle switch was added at the voltage input. This allowed the system to be turned on and off easier than by plugging in and unplugging a power supply form a wall socket. This led to testing of the entire system.
VI. Testing

Full system testing caused the most issues during the development of this project. This was not necessarily due to design problems, but from the nature of testing required. First, testing could only be done after sunset, when the North Star was in the sky. This limited testing to about a 12 hour period each night. Secondly, weather was a huge obstacle. Over the two week testing period, six days prevented testing due to the rain and four days prevented testing due to cloudy skies. This reduced the testing period to 5 days; severely limiting the amount of time spent altering the code to ensure proper tracking.

Beyond the weather, the length of each test was a significant limiting factor. As listed in the requirements, a goal of accurate tracking for three hours was decided upon. This meant that any tracking test would take at least 3 hours to complete. To make the most out of each test, the most extreme conditions were tested, namely tracking of stars on the celestial equator. These stars move the most over the course of the night and any small errors in the tracking software would be exacerbated, and thus more noticeable when compared to tracking a star close to the celestial pole.

The final testing concern was with the GPS and compass modules. In order to fully test each module, they would theoretically need to be tested at every location in the continental United States. This extensive testing is not realistic in time or money. Instead, testing of the system occurred at two locations, separated by about 40 miles. These two locations, area codes 93436 and 93410, are in a different latitude and longitude grids square and thus have different values for magnetic declination and
location. While not testing the system for every possible condition, the different grids would help test more than one condition for the magnetic declination setup and GPS signal formatting. This was deemed an acceptable solution because the GPS signal format should not change since it is sent in the standard NEMA format. It is also a good assumption that if the magnetic declination is properly selected in more than one case than it should be able to do it in most, if not all cases. The only concern is testing of the boundary conditions; again however this was not economically feasible. Ideally testing would occur on opposite side of the United States, over a long winter night, but was not possible here.

Unless otherwise stated, testing occurred in the 93436 area code, where the majority of testing was conducted. The first round of tests was to refine the accuracy of the mount in pointing to Polaris. It was found that the magnetic sensor worked correctly, pointing to true north, however the servo wasn’t as successful. Due to the weight of the telescope and the initial position of the servo, the servo would move the mount to an angle 2 degrees below the celestial pole. To counter this inaccuracy, the starting position of the servo was increased by 2 degrees. This allowed for proper alignment. This test was repeated several more times, on different nights and locations, showing that this alteration fixed the issue and didn’t introduce any new errors. The test was deemed successful if Polaris was inside the view of the telescope after alignment. It was never perfectly in the center, but varied between 0.2 and 0.6 degrees off from center in each test after the servo fix, as seen in table 3. This small error did not seem to negatively affect the tracking ability in the next round of testing.
<table>
<thead>
<tr>
<th>Test #</th>
<th>Location</th>
<th>Error amount</th>
<th>Success?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93436</td>
<td>-2 degrees</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>93436</td>
<td>0.3</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>93436</td>
<td>0.2</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>93436</td>
<td>0.6</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>93436</td>
<td>0.6</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>93436</td>
<td>0.4</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>93410</td>
<td>0.6</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>93410</td>
<td>0.5</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>93410</td>
<td>0.6</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3: Mount setup testing results

After calibration of the mount was completed, the tracking capabilities were tested. This was the most lengthy part and required large blocks of time to accomplish. Fortunately, the three hour tracking capability of the design was ultimately successful in the 93436 and 93410 zip codes. To test this, a star upon the celestial equator was chosen and the telescope was positioned on it after the mount was calibrated. This star varied depending on the time of night, but always lied near the celestial equator. The project was then put into tracking mode. The only changes that needed to be made to the design to accomplish the required accuracy was the delay between steps. The original calculation was for a 4500 ms delay between steps, however this delay was too long. The actual time found for the required accuracy was 4300 ms. This meant that a full rotation of the RA screw was closer to 14 minutes than the originally estimated 15 minutes.

The 4300ms delay was sufficient to track the star for up to 4 hours before testing was halted. The ultimate goal of a tracking accuracy of up to 12 hours was not confirmed through testing for two reasons. First, since testing never occurred on the
longest night of the year, as the project was not completed until after that date, a full 12 hour tracking test couldn’t be completed. Second, due to the size of the telescope and the surrounding added motors and other equipment, movement of the telescope was restricted. This restriction would vary depending on the latitude angle on the mount, but in the worst case scenario, would be limited to just 8 hours of tracking. However the main goal of three hours was met and exceeded by at least 33%.
VII. Conclusions and Recommendations

Overall this project was a success. The number of variables that needed to be accounted for grew from start to finish, increasing the complexity, however these variables were successfully brought together to create a functional product. After refining the code, the mount was able to correctly find the celestial pole, properly aligning the equatorial mount. This alignment was done with minimal user interaction and input. The right ascension motor was then able to accurately track a point on the celestial sphere for at least four hours, exceeding the three hour requirement. This was all done by modifying an existing mount, and properly worked off a 12V input.

While the requirements were all met, there are a few areas that could be improved upon. The majority of these can be done through software, greatly increasing the functionality with minimal additional cost. First, the ability for the project to work outside of the continental United States, specifically adding functionality to align to the south celestial pole could be added when the device is in the southern hemisphere. Small changes would need to be made to the compass code to have it point to the South Pole and magnetic declination data would need to be added to the look-up table. Additionally, the RA motor would need to rotate in the opposite direction, and the servo code would need to be altered to be able to handle the negative latitude data, but this is all possible with a software update.
There were bumps and detours along the development path, but the final design was able to meet the initial requirements. Overall the modular testing and design ultimately turned out a successful approach and one that will be used by me again. This project was a great learning experience, and while this project is complete, I will continue to modify the mount in order to add functionality and to improve upon the design wherever possible. If there are any questions or concerns regarding this paper or project, please email me at jwburket@calpoly.edu.
VIII. Bibliography

*CPE329 Reader* by Dr. Liddicoat and Dr. Slivovsky
Digital keypad configuration

*How to Use a Computerized Telescope* by Michael A. Covington
Astronomy reference book

http://www.adafruit.com
GPS module procurement
GPS module code and datasheet reference

http://arduino.cc/
Arduino Library reference
Arduino Language reference

PowerSeeker 127EQ information

http://www.digikey.com
Parts for procurement
Part datasheets

http://www.ee.calpoly.edu
Report format and setup

http://www.ngdc.noaa.gov/geomagmodels/Declination.jsp
Magnetic declination calculations and information

Servo motor procurement and technical specs

*Practical Astronomy with your Calculator* by Peter Duffett-Smith
Astronomy reference and math reference book

*The Art and Science of C* by E. Roberts
C programming review and reference book
IX. References


### Appendix A: Parts List

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
<th>Quantity</th>
<th>Price/each ($)</th>
<th>Total Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM350TFS-ND</td>
<td>Adjustable voltage regulator, 3 Amp max output</td>
<td>1</td>
<td>0.74</td>
<td>0.74</td>
</tr>
<tr>
<td>23H118D10B</td>
<td>High torque, bi-polar stepper motor</td>
<td>1</td>
<td>102.41</td>
<td>102.41</td>
</tr>
<tr>
<td>B3F-1000</td>
<td>6mm SPST button</td>
<td>12</td>
<td>0.20</td>
<td>2.40</td>
</tr>
<tr>
<td>HC49US-16.000MABJ</td>
<td>16MHz crystal oscillator</td>
<td>2</td>
<td>0.63</td>
<td>1.26</td>
</tr>
<tr>
<td>L239D</td>
<td>16-DIP H-bridge IC</td>
<td>1</td>
<td>3.92</td>
<td>3.92</td>
</tr>
<tr>
<td>R1655</td>
<td>Robson flux field sensor (compass)</td>
<td>1</td>
<td>39.00</td>
<td>39.00</td>
</tr>
<tr>
<td>121-010-013</td>
<td>5ft black, 10 wire cable</td>
<td>1</td>
<td>4.74</td>
<td>4.74</td>
</tr>
<tr>
<td>CA3096CE</td>
<td>NPN/PNP BJT array</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>C315C220J2G5TA</td>
<td>22pF capacitor</td>
<td>4</td>
<td>0.21</td>
<td>0.84</td>
</tr>
<tr>
<td>EM-406a</td>
<td>GPS module</td>
<td>1</td>
<td>60.00</td>
<td>60.00</td>
</tr>
<tr>
<td>HS-7950TH</td>
<td>High torque servo motor</td>
<td>1</td>
<td>142.95</td>
<td>142.95</td>
</tr>
<tr>
<td>Atmega328</td>
<td>8bit AVR microcontroller</td>
<td>2</td>
<td>6.00</td>
<td>12.00</td>
</tr>
<tr>
<td>HD44780</td>
<td>20X4 LCD screen</td>
<td>1</td>
<td>18.00</td>
<td>18.00</td>
</tr>
<tr>
<td>Memsic 2125</td>
<td>dual axis accelerometer</td>
<td>1</td>
<td>32.99</td>
<td>32.99</td>
</tr>
<tr>
<td>10k-pot</td>
<td>0-10k potentiometer</td>
<td>1</td>
<td>1.49</td>
<td>1.49</td>
</tr>
<tr>
<td>273-240</td>
<td>12V D, 40mm brushless cooling fan</td>
<td>1</td>
<td>12.99</td>
<td>12.99</td>
</tr>
<tr>
<td>276-1368</td>
<td>TO-220 aluminum heat sink</td>
<td>1</td>
<td>1.99</td>
<td>1.99</td>
</tr>
<tr>
<td>Wood</td>
<td>1/4 in, 2X2 plywood</td>
<td>1</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>Loctite epoxy</td>
<td>Quick set epoxy</td>
<td>1</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Brackets</td>
<td>Metal brackets and wood screws for mounting (4 pack)</td>
<td>1</td>
<td>2.99</td>
<td>2.99</td>
</tr>
<tr>
<td>Resistors</td>
<td>Assorted resistor values</td>
<td>12</td>
<td>0.10</td>
<td>1.20</td>
</tr>
<tr>
<td>20uF capacitor</td>
<td>20uF polarized capacitor</td>
<td>1</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>100nF capacitor</td>
<td>100nF capacitor</td>
<td>1</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>Wire</td>
<td>Assorted wires and jumpers</td>
<td>1</td>
<td>2.00</td>
<td>2.00</td>
</tr>
</tbody>
</table>
Appendix B: Schematics

Overall system block diagram
Accelerometer block diagram, X and Y out go to pins 12 and 13 on the Master device

Compass Block diagram, Vout A and B go to pins 23 and 24 on the Master device
GPS block diagram, Tx, Rx and Vcc are connected to pins 9, 10 and 11 on the Master
Keypad block diagram, O1 – O4 and I1 – I3 are connected to pins 4 - 6 and 11 – 14 on the master device

![Keypad Block Diagram](image1)

LCD block diagram, pins RS, RW, E and D0 – D3 are connected to pins 13 -18 on the slave device

![LCD Block Diagram](image2)

Servo block diagram, the signal pin is connected to pin 12 on the slave device

![Servo Block Diagram](image3)
Stepper Motor block diagram, motor control pins 1 and 2 are connect to pins 6 and 11 on the slave device

Power Supply block diagram, 12V input for the “battery” and the 5V output goes to the other modules
## ATmega328p Master pin connections

<table>
<thead>
<tr>
<th>Pin #</th>
<th>In/Out</th>
<th>Function</th>
<th>Connected to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Reset</td>
<td>Pull up resistor to Vcc</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>Digital 0</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>Digital 1</td>
<td>NC</td>
</tr>
<tr>
<td>4</td>
<td>OUT</td>
<td>Digital 2</td>
<td>To keypad O1</td>
</tr>
<tr>
<td>5</td>
<td>OUT</td>
<td>Digital 3</td>
<td>To keypad O2</td>
</tr>
<tr>
<td>6</td>
<td>OUT</td>
<td>Digital 4</td>
<td>To keypad O3</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>Vcc</td>
<td>To Vcc</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>Gnd</td>
<td>To ground</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>Oscillator1</td>
<td>To crystal oscillator</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>Oscillator2</td>
<td>To crystal oscillator</td>
</tr>
<tr>
<td>11</td>
<td>OUT</td>
<td>Digital 5</td>
<td>To keypad O4</td>
</tr>
<tr>
<td>12</td>
<td>IN</td>
<td>Digital 6</td>
<td>To keypad I1</td>
</tr>
<tr>
<td>13</td>
<td>IN</td>
<td>Digital 7</td>
<td>To keypad I2</td>
</tr>
<tr>
<td>14</td>
<td>IN</td>
<td>Digital 8</td>
<td>To keypad I3</td>
</tr>
<tr>
<td>15</td>
<td>IN/OUT</td>
<td>Digital 9</td>
<td>GPS Serial connection</td>
</tr>
<tr>
<td>16</td>
<td>IN/OUT</td>
<td>Digital 10</td>
<td>GPS Serial connection</td>
</tr>
<tr>
<td>17</td>
<td>OUT</td>
<td>Digital 11</td>
<td>To Vcc on GPS</td>
</tr>
<tr>
<td>18</td>
<td>IN</td>
<td>Digital 12</td>
<td>From X-axis on Accelerometer</td>
</tr>
<tr>
<td>19</td>
<td>IN</td>
<td>Digital 13</td>
<td>From Y-axis on Accelerometer</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>AVcc</td>
<td>To Vcc</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>AVref</td>
<td>NC</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>AGnd</td>
<td>To ground</td>
</tr>
<tr>
<td>23</td>
<td>IN</td>
<td>Analog 0</td>
<td>From Compass, A curve</td>
</tr>
<tr>
<td>24</td>
<td>IN</td>
<td>Analog 1</td>
<td>From Compass, B curve</td>
</tr>
<tr>
<td>25</td>
<td>IN</td>
<td>Analog 2</td>
<td>NC</td>
</tr>
<tr>
<td>26</td>
<td>IN</td>
<td>Analog 3</td>
<td>NC</td>
</tr>
<tr>
<td>27</td>
<td>IN</td>
<td>Analog 4</td>
<td>SDL line to slave</td>
</tr>
<tr>
<td>28</td>
<td>IN</td>
<td>Analog 5</td>
<td>SCL line to slave</td>
</tr>
</tbody>
</table>
### ATmega328p Slave pin connections

<table>
<thead>
<tr>
<th>Pin #</th>
<th>In/Out</th>
<th>Function</th>
<th>Connected to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Reset</td>
<td>Pull up resistor to Vcc</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>Digital 0</td>
<td>NC</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>Digital 1</td>
<td>NC</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>Digital 2</td>
<td>NC</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>Digital 3</td>
<td>NC</td>
</tr>
<tr>
<td>6</td>
<td>OUT</td>
<td>Digital 4</td>
<td>To motor control pin 1</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>Vcc</td>
<td>To Vcc</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>Gnd</td>
<td>To Ground</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>Oscillator1</td>
<td>To crystal oscillator</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>Oscillator2</td>
<td>To crystal oscillator</td>
</tr>
<tr>
<td>11</td>
<td>OUT</td>
<td>Digital 5</td>
<td>To motor control pin 2</td>
</tr>
<tr>
<td>12</td>
<td>OUT</td>
<td>Digital 6</td>
<td>PWM to servo</td>
</tr>
<tr>
<td>13</td>
<td>IN/OUT</td>
<td>Digital 7</td>
<td>To LCD RS pin</td>
</tr>
<tr>
<td>14</td>
<td>IN/OUT</td>
<td>Digital 8</td>
<td>To LCD Enable pin</td>
</tr>
<tr>
<td>15</td>
<td>IN/OUT</td>
<td>Digital 9</td>
<td>To LCD, data 0 pin</td>
</tr>
<tr>
<td>16</td>
<td>IN/OUT</td>
<td>Digital 10</td>
<td>To LCD, data 1 pin</td>
</tr>
<tr>
<td>17</td>
<td>IN/OUT</td>
<td>Digital 11</td>
<td>To LCD, data 2 pin</td>
</tr>
<tr>
<td>18</td>
<td>IN/OUT</td>
<td>Digital 12</td>
<td>To LCD, data 3 pin</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>Digital 13</td>
<td>NC</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>AVcc</td>
<td>To Vcc</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>AVref</td>
<td>NC</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>AGnd</td>
<td>To ground</td>
</tr>
<tr>
<td>23</td>
<td>IN</td>
<td>Analog 0</td>
<td>NC</td>
</tr>
<tr>
<td>24</td>
<td>IN</td>
<td>Analog 1</td>
<td>NC</td>
</tr>
<tr>
<td>25</td>
<td>IN</td>
<td>Analog 2</td>
<td>NC</td>
</tr>
<tr>
<td>26</td>
<td>IN</td>
<td>Analog 3</td>
<td>NC</td>
</tr>
<tr>
<td>27</td>
<td>IN</td>
<td>Analog 4</td>
<td>SDL to Master</td>
</tr>
<tr>
<td>28</td>
<td>IN</td>
<td>Analog 5</td>
<td>SCL to Master</td>
</tr>
</tbody>
</table>
Appendix C: ATmega328 code (Master)
#include <Wire.h>
#include <NewSoftSerial.h>

// mode variables
boolean startup = true;
boolean tracking = true;
char section;

// keypad
int press = 0;
#define o0 2
#define o1 3
#define o2 4
#define o3 5
#define i0 6
#define i1 7
#define i2 8

// servo
int servoPos = 223;
int finalPos;

// GPS
NewSoftSerial mySerial = NewSoftSerial(9, 10);
#define powerPin 11
#define GPSRATE 4800
#define BUFSIZ 90
char buffer[BUFSIZ];
char *parseptr;
char buffidx;
byte hour, minute, second, year, month, date;
unsigned long latitude, longitude;
byte lat1, lat2, lat3, long1, long2, long3;
char latdir, longdir;
char stat;
int i;

// compass
long upperLimit, lowerLimit, curveA, curveB, scale;
long compass;
char compassSign;

// declination arrays as of Jan 1, 2010(lat 25 - 51 and long 70 - 120)
int lat25[] = {11, 11, 10, 10, 9, 9, 8, 8, 7, 7, 6, 5, 4, 3, 2, 1, 1, 0, -1, -1, -2, -2, -3, -3, -4, -4, -5, -5, -6, -6, -7, -7, -7, -8, -8, -9, -9, -10, -10, -10, -10, -11, -11, -11, -11, -11, -11, -12, -12, -12};
int lat26[] = {11, 11, 10, 10, 9, 9, 8, 8, 7, 7, 6, 5, 4, 3, 2, 1, 1, 0, -1, -1, -2, -2, -3, -3, -4, -4, -5, -5, -6, -6, -7, -7, -7, -8, -8, -9, -9, -10, -10, -10, -10, -11, -11, -11, -11, -11, -11, -11, -12, -12, -12};
int lat27[] = {12, 12, 11, 11, 10, 9, 9, 8, 8, 7, 7, 6, 5, 4, 3, 2, 2, 1, 0, -1, -1, -2, -2, -3, -3, -4, -4, -5, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -12, -12, -12, -12, -12};
int lat28[] = {12, 11, 11, 10, 9, 8, 7, 6, 5, 4, 4, 3, 2, 2, 1, 0, 0, -1, -2, -2, -3, -3, -4, -4, -5, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -13, -14, -14, -15, -16};
int lat29[] = {12, 11, 11, 10, 9, 8, 7, 6, 5, 4, 3, 3, 2, 1, 0, -1, -2, -3, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -14, -14, -15, -15, -16};
int lat30[] = {12, 11, 11, 10, 9, 8, 7, 6, 5, 5, 4, 3, 3, 2, 1, 1, 0, -1, -2, -3, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -14, -14, -15, -15, -16};
int lat31[] = {12, 11, 11, 10, 9, 8, 7, 6, 5, 5, 4, 3, 3, 2, 1, 1, 0, -1, -2, -3, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -14, -14, -15, -15, -16};
int lat32[] = {12, 11, 11, 10, 9, 8, 7, 6, 6, 5, 4, 3, 2, 1, 1, 0, -1, -2, -3, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat33[] = {13, 12, 12, 11, 11, 10, 9, 8, 7, 7, 6, 6, 5, 4, 3, 2, 1, 0, 0, -1, -2, -2, -3, -4, -4, -5, -5, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat34[] = {13, 12, 12, 11, 11, 10, 9, 9, 8, 7, 7, 6, 5, 5, 4, 3, 2, 2, 1, 0, 0, -1, -2, -2, -3, -4, -4, -5, -5, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat35[] = {14, 13, 13, 12, 12, 11, 10, 10, 9, 9, 8, 7, 6, 5, 4, 4, 3, 2, 1, 1, 0, -1, -2, -2, -3, -4, -4, -5, -5, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat36[] = {14, 13, 13, 12, 12, 11, 10, 10, 9, 9, 8, 7, 6, 5, 4, 4, 3, 2, 1, 1, 0, -1, -2, -2, -3, -4, -4, -5, -5, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat37[] = {14, 14, 13, 13, 12, 12, 11, 10, 10, 9, 8, 8, 7, 6, 5, 4, 3, 2, 2, 1, 0, -1, -1, -2, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat38[] = {14, 14, 13, 13, 12, 11, 10, 10, 9, 8, 8, 7, 6, 5, 5, 4, 3, 2, 2, 1, 0, -1, -1, -2, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat39[] = {15, 14, 13, 13, 12, 12, 11, 10, 10, 9, 9, 8, 7, 7, 6, 5, 4, 3, 3, 2, 1, 0, -1, -1, -2, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat40[] = {15, 14, 13, 13, 12, 12, 11, 10, 10, 9, 9, 8, 7, 7, 6, 5, 4, 3, 3, 2, 1, 0, -1, -1, -2, -3, -4, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat41[] = {15, 14, 14, 13, 13, 12, 11, 10, 10, 9, 9, 8, 7, 6, 5, 5, 4, 3, 2, 1, 1, 0, -1, -2, -3, -3, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat42[] = {15, 15, 14, 14, 13, 13, 12, 11, 10, 10, 9, 9, 8, 7, 6, 5, 5, 4, 3, 2, 1, 0, -1, -2, -3, -3, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16};
int lat43[] = {16, 15, 15, 14, 14, 13, 13, 12, 11, 10, 9, 8, 8, 7, 6, 5, 4, 3, 2, 2, 1, 0, -1, -2, -3, -3, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16, -16, -16, -16};
int lat44[] = {16, 15, 15, 14, 14, 13, 13, 12, 11, 10, 9, 8, 8, 7, 6, 5, 4, 3, 2, 2, 1, 0, -1, -2, -3, -3, -4, -5, -6, -6, -7, -7, -8, -8, -9, -9, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16, -16, -16, -16};
int lat45[] = {17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -15, -15, -15, -16, -16, -16, -17};
int lat46[] = {17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -14, -15, -15, -15, -16, -16, -17, -17, -17, -17};
int lat47[] = {17, 17, 16, 15, 14, 13, 12, 11, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -13, -14, -14, -15, -15, -16, -16, -17, -17, -17, -17};
int lat48[] = {17, 17, 16, 15, 14, 13, 12, 11, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -5, -6, -7, -8, -9, -10, -11, -12, -13, -13, -14, -14, -15, -15, -16, -16, -17, -17, -17, -17};
int lat49[] = {18, 18, 17, 17, 17, 16, 16, 15, 14, 14, 14, 13, 12, 11, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -5, -6, -7, -8, -9, -9, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16, -16, -17, -17, -17, -17};
int lat50[] = {18, 18, 17, 17, 17, 16, 16, 15, 14, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, -5, -6, -7, -8, -9, -9, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16, -16, -17, -17, -17, -17};
int lat51[] = {19, 19, 18, 18, 17, 17, 16, 15, 14, 13, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, -1, -2, -2,- 3, -4, -5, -6, -7, -8, -9, -9, -10, -11, -11, -12, -12, -13, -13, -14, -14, -15, -15, -16, -16, -17, -17, -17, -18, -18, -18, -18};
int arrayVal;

//accel
#define pi 3.1415926545898
int pin_X = 12;
int pin_Y = 13;
double degX, degY;
int degXdec, degYdec;
char signDegX, signDegY;

//motor
#define wait 4300 //4.3 seconds between 1.8deg steps
int tempWait = 0;
byte tempMotorPin1 = 0;
byte tempMotorPin2 = 0;

void setup(){
  //keypad
  pinMode(o0, OUTPUT);
  pinMode(o1, OUTPUT);
  pinMode(o2, OUTPUT);
  pinMode(o3, OUTPUT);
  pinMode(i0, INPUT);
  pinMode(i1, INPUT);
  pinMode(i2, INPUT);
  digitalWrite(o0, HIGH);
  digitalWrite(o1, HIGH);
  digitalWrite(o2, HIGH);
  digitalWrite(o3, HIGH);

  //compass

upperLimit = 600L;
lowerLimit = 424L;
scale = (((9000/(upperLimit-lowerLimit))*10)/4);

//accel
pinMode(pin_X, INPUT);
pinMode(pin_Y, INPUT);

//GPS
pinMode(powerPin, OUTPUT);
mySerial.begin(GPSRATE);
digitalWrite(powerPin, HIGH);

//Master-Slave communication
Wire.begin();

void loop(){
    press = button();
    //startup mode
    if(startup){
        start();
    }
    //star tracking mode
    if(tracking){
        track();
    }
    //standby mode, show time and date
    if(!tracking){
        noTrack();
    }
}

//function to determine what button, if any, if pushed
int button(void){
    digitalWrite(o0, LOW);
    if(!digitalRead(i0)){
        digitalWrite(o0, HIGH);
        return(1);
    }
    if(!digitalRead(i1)){
        digitalWrite(o0, HIGH);
        return(2);
    }
    if(!digitalRead(i2)){
        digitalWrite(o0, HIGH);
        return(3);
    }
    digitalWrite(o0, HIGH);
    digitalWrite(o1, LOW);
    if(!digitalRead(i0)){

digitalWrite(o1, HIGH);
return(4);
}
if(!digitalRead(i1)){
digitalWrite(o1, HIGH);
return(5);
}
if(!digitalRead(i2)){
digitalWrite(o1, HIGH);
return(6);
} digitalWrite(o1, HIGH);
digitalWrite(o2, LOW);

if(!digitalRead(i0)){
digitalWrite(o2, HIGH);
return(7);
}
if(!digitalRead(i1)){
digitalWrite(o2, HIGH);
return(8);
}
if(!digitalRead(i2)){
digitalWrite(o2, HIGH);
return(9);
} digitalWrite(o2, HIGH);
digitalWrite(o3, LOW);

if(!digitalRead(i0)){
digitalWrite(o3, HIGH);
return(10);
}
if(!digitalRead(i1)){
digitalWrite(o3, HIGH);
return(11);
}
if(!digitalRead(i2)){
digitalWrite(o3, HIGH);
return(12);
}

digitalWrite(o3, HIGH);
//nothing seen
return(0);
}

//
unsigned long parsedecimal(char *str) {
unsigned long d = 0;

while (str[0] != 0) {
    if ((str[0] > '9') || (str[0] < '0'))
return d;
d *= 10;
d += str[0] - '0';
str++;
}
return d;
}
//reading and storing GPS data
void readline(void) {
char c;

buffidx = 0; // start at beginning
while (1) {
c=mySerial.read();
if (c == -1)
    continue;
if (c == '\n')
    continue;
if ((buffidx == BUFFSIZ-1) || (c == '\r')) {
    buffer[buffidx] = 0;
    return;
}
buffer[buffidx++]= c;
}
}

long dir(void){
    curveA = analogRead(0);
    curveB = analogRead(1);

    //calibrate
    if(curveA == curveB){
        if(curveA > 512){
            upperLimit = curveA;
        }
        else{
            lowerLimit = curveA;
        }
        scale = (((9000/(upperLimit-lowerLimit))*10)/4);
    }
    // 0 - 90 degrees
    if(curveB >= upperLimit){
        compass = (((upperLimit - curveA) * scale)/10)*4)/10;
    }
    // 270 - 360 degrees
    if(curveA >= upperLimit){
        compass = (((curveB - lowerLimit) * scale)/10)*4)/10;
        compass = compass + 2700;
        if(compass >= 3600){
            compass = compass - 3600;
        }
    }
}
// 90 - 180 degrees
if(curveA <= lowerLimit){
    compass = (((upperLimit - curveB) * scale)/10)*4)/10;
    compass = compass + 900;
}
// 180 - 270 degrees
if(curveB <= lowerLimit){
    compass = (((curveA - lowerLimit) * scale)/10)*4)/10;
    compass = compass + 1800;
}
return (compass/10);
}

void angleX(void){
    long T1_X = pulseIn(pin_X, HIGH);
    if(T1_X < 2500)
        return;
    double accelX = ((T1_X / 10) - 500) * 8;
    degX = asin(accelX/1000) * 180 / pi;
    if(degX < 0)
        signDegX = '-';
    else if(degX > 0)
        signDegX = '+';
    degX = abs(degX);
    degXdec = degX*100;
    degXdec = degXdec%100;
}

void angleY(void){
    long T1_Y = pulseIn(pin_Y, HIGH);
    if(T1_Y < 2500)
        return;
    double accelY = ((T1_Y / 10) - 500) * 8;
    degY = asin(accelY/1000) * 180 / pi;
    if(degY < 0)
        signDegY = '-';
    else if(degY > 0)
        signDegY = '+';
    degY = abs(degY);
    degYdec = degY*100;
    degYdec = degYdec%100;
}

void start(void){
    // level stand
    section = '2';
    while(press != 5){
        angleX();
    }
angleY();

Wire.beginTransmission(4); // transmit to slave
Wire.send(section);
Wire.send(signDegX);
Wire.send(byte(degX));
Wire.send(byte(degXdec));
Wire.send(signDegY);
Wire.send(byte(degY));
Wire.send(byte(degYdec));
Wire.endTransmission();
delay(50);
press = button();
}
delay(1000);
press = button();

//GPS
section = '8';
Wire.beginTransmission(4); // transmit to slave
Wire.send(section);
Wire.endTransmission();
gps();
while(stat != 'A'){
gps();
}

//time
section = '4';
while(press != 5){
    Wire.beginTransmission(4); // transmit to slave
    Wire.send(section);
    Wire.send(date);
    Wire.send(month);
    Wire.send(year);
    Wire.send(hour);
    Wire.send(minute);
    Wire.send(second);
    Wire.endTransmission();
gps();
delay(50);
press = button();
}
delay(1000);
press = button();

//lat and long
section = '5';

while(press != 5){
    Wire.beginTransmission(4); // transmit to slave
    Wire.send(section);
    Wire.send(latdir);
Wire.send(lat1);
Wire.send(lat2);
Wire.send(lat3);
Wire.send(longdir);
Wire.send(long1);
Wire.send(long2);
Wire.send(long3);
Wire.endTransmission();
delay(50);
press = button();
}
delay(1000);
press = button();
startup = false;

section = '3';
press = button();

//point north
while(press != 5){
    compass = dir();
    //add declination
    arrayVal = long1 - 70;
    if(lat1 == 25)
        compass = compass + lat25[arrayVal];
    else if(lat1 == 26)
        compass = compass + lat26[arrayVal];
    else if(lat1 == 27)
        compass = compass + lat27[arrayVal];
    else if(lat1 == 28)
        compass = compass + lat28[arrayVal];
    else if(lat1 == 29)
        compass = compass + lat29[arrayVal];
    else if(lat1 == 30)
        compass = compass + lat30[arrayVal];
    else if(lat1 == 31)
        compass = compass + lat31[arrayVal];
    else if(lat1 == 32)
        compass = compass + lat32[arrayVal];
    else if(lat1 == 33)
        compass = compass + lat33[arrayVal];
    else if(lat1 == 34)
        compass = compass + lat34[arrayVal];
    else if(lat1 == 35)
        compass = compass + lat35[arrayVal];
    else if(lat1 == 36)
        compass = compass + lat36[arrayVal];
    else if(lat1 == 37)
        compass = compass + lat37[arrayVal];
    else if(lat1 == 38)
        compass = compass + lat38[arrayVal];
    else if(lat1 == 39)
        compass = compass + lat39[arrayVal];
}
else if(lat1 == 40)
    compass = compass + lat40[arrayVal];
else if(lat1 == 41)
    compass = compass + lat41[arrayVal];
else if(lat1 == 42)
    compass = compass + lat42[arrayVal];
else if(lat1 == 43)
    compass = compass + lat43[arrayVal];
else if(lat1 == 44)
    compass = compass + lat44[arrayVal];
else if(lat1 == 45)
    compass = compass + lat45[arrayVal];
else if(lat1 == 46)
    compass = compass + lat46[arrayVal];
else if(lat1 == 47)
    compass = compass + lat47[arrayVal];
else if(lat1 == 48)
    compass = compass + lat48[arrayVal];
else if(lat1 == 49)
    compass = compass + lat49[arrayVal];
else if(lat1 == 50)
    compass = compass + lat50[arrayVal];
else if(lat1 == 51)
    compass = compass + lat51[arrayVal];
// wrap around if over 360 or under 0 degrees
if(compass > 360) {
    compass = compass - 360;
}
else if(compass < 0) {
    compass = compass + 360;
}

// change heading for byte transfer
if(compass > 180) {
    compass = 360 - compass;
    compassSign = '-';
}
else {
    compassSign = '+';
}

Wire.beginTransmission(4); // transmit to slave
Wire.send(section);
Wire.send(compassSign);
Wire.send(byte(compass));
Wire.endTransmission();
delay(50);
press = button();
}
delay(1000);
press = button();

// servo
section = '1';
finalPos = 223 - lat1;
while(servoPos > finalPos){
    if(press == 1)
        break;
    Wire.beginTransmission(4);
    Wire.send(section);
    Wire.send(servoPos);
    Wire.endTransmission();
    servoPos = servoPos - 1;
    delay(200);
    press = button();
}
press = button();
while(press != 5){
    if(press == 2){
        if(servoPos > 71)
            servoPos = servoPos - 1;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(servoPos);
        Wire.endTransmission();
        delay(500);
        press = button();
    }
    if(press == 8){
        if(servoPos < 223)
            servoPos = servoPos + 1;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(servoPos);
        Wire.endTransmission();
        delay(500);
        press = button();
    }
    press = button();
}
delay(1000);
servoPos = 90;

//tracking mode, stepper motor control
void track(void){
    section = '6';
    tempWait = 0;
    Wire.beginTransmission(4);
    Wire.send(section);
    Wire.send(tempMotorPin1);
    Wire.send(tempMotorPin2);
    Wire.endTransmission();
    delay (50);
    press = button();
}
if(tempMotorPin1 == 0 && tempMotorPin2 == 0)
    while(tempWait < wait && press != 5)
    
    if(press == 4)
    {
        tempMotorPin1 = 1;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(tempMotorPin1);
        Wire.send(tempMotorPin2);
        Wire.endTransmission();
        delay(200);
        goto end;
    }
    
    if(press == 6)
    {
        tempMotorPin2 = 1;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(tempMotorPin1);
        Wire.send(tempMotorPin2);
        Wire.endTransmission();
        delay(200);
        goto end;
    }
    delay(10);
    tempWait = tempWait + 10;
    press = button();
}

if(press != 5)
    tempMotorPin2 = 1;
    Wire.beginTransmission(4);
    Wire.send(section);
    Wire.send(tempMotorPin1);
    Wire.send(tempMotorPin2);
    Wire.endTransmission();
}

if(press == 5)
    tracking = false;
}

if(tempMotorPin1 == 0 && tempMotorPin2 == 1)
    while(tempWait < wait && press != 5)
    
    if(press == 4)
    {
        tempMotorPin2 = 0;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(tempMotorPin1);
        Wire.send(tempMotorPin2);
        Wire.endTransmission();
        delay(200);
        goto end;
    }
    
    if(press == 6)
tempMotorPin1 = 1;
Wire.beginTransmission(4);
Wire.send(section);
Wire.send(tempMotorPin1);
Wire.send(tempMotorPin2);
Wire.endTransmission();
delay(200);
goto end;
}
delay(10);
tempWait = tempWait + 10;
press = button();
}
if(press != 5){
    tempMotorPin1 = 1;
    Wire.beginTransmission(4);
    Wire.send(section);
    Wire.send(tempMotorPin1);
    Wire.send(tempMotorPin2);
    Wire.endTransmission();
}
if(press == 5)
    tracking = false;
}
if(tempMotorPin1 == 1 && tempMotorPin2 == 1){
    while(tempWait < wait && press != 5){
        if(press == 4){
            tempMotorPin1 = 0;
            Wire.beginTransmission(4);
            Wire.send(section);
            Wire.send(tempMotorPin1);
            Wire.send(tempMotorPin2);
            Wire.endTransmission();
            delay(200);
            goto end;
        }
        if(press == 6){
            tempMotorPin2 = 0;
            Wire.beginTransmission(4);
            Wire.send(section);
            Wire.send(tempMotorPin1);
            Wire.send(tempMotorPin2);
            Wire.endTransmission();
            delay(200);
            goto end;
        }
        delay(10);
        tempWait = tempWait + 10;
        press = button();
    }
}
}
if(press != 5){
    tempMotorPin2 = 0;
    Wire.beginTransmission(4);
    Wire.send(section);
    Wire.send(tempMotorPin1);
    Wire.send(tempMotorPin2);
    Wire.endTransmission();
    
    if(press == 5)
        tracking = false;
}

if(tempMotorPin1 == 1 && tempMotorPin2 == 0){
    while(tempWait < wait && press != 5){
        if(press == 4){
            tempMotorPin2 = 1;
            Wire.beginTransmission(4);
            Wire.send(section);
            Wire.send(tempMotorPin1);
            Wire.send(tempMotorPin2);
            Wire.endTransmission();
            delay(200);
            goto end;
        }
        if(press == 6){
            tempMotorPin1 = 0;
            Wire.beginTransmission(4);
            Wire.send(section);
            Wire.send(tempMotorPin1);
            Wire.send(tempMotorPin2);
            Wire.endTransmission();
            delay(200);
            goto end;
        }
        delay(10);
        tempWait = tempWait + 10;
        press = button();
    }
    if(press != 5){
        tempMotorPin1 = 0;
        Wire.beginTransmission(4);
        Wire.send(section);
        Wire.send(tempMotorPin1);
        Wire.send(tempMotorPin2);
        Wire.endTransmission();
    }
    if(press == 5)
        tracking = false;
}

delay(1000);
end:

tempWait = 0;
press = button();
}

void gps(void){
    unsigned long tmp;
    readline();

    // check if line starts with $GPRMC
    if (strncmp(buffer, "$GPRMC", 6) == 0) {

        // time data
        parseptr = buffer+7;
        tmp = parsedecimal(parseptr);
        hour = tmp / 10000;
        minute = (tmp / 100) % 100;
        second = tmp % 100;

        parseptr = strchr(parseptr, ',') + 1;
        stat = parseptr[0];
        parseptr += 2;

        // latitude and longitude data
        // latitude
        latitude = parsedecimal(parseptr);
        if (latitude != 0) {
            latitude *= 10000;
            parseptr = strchr(parseptr, '.')+1;
            latitude += parsedecimal(parseptr);
        }
        parseptr = strchr(parseptr, ',') + 1;
        // read latitude N/S data
        if (parseptr[0] != ',') {
            latdir = parseptr[0];
        }

        // format latitude data
        if (latdir == 'N')
            latdir = '+';
        else if (latdir == 'S')
            latdir = '-';
        lat1 = latitude/1000000;
        lat2 = (latitude/10000)%100;
        lat3 = (latitude%10000)*6/1000;

        // longitude
        parseptr = strchr(parseptr, ',')+1;
        longitude = parsedecimal(parseptr);
        if (longitude != 0) {
            longitude *= 10000;
            parseptr = strchr(parseptr, '.')+1;
            longitude += parsedecimal(parseptr);
        }
        parseptr = strchr(parseptr, ',')+1;
// read longitude E/W data
if (parseptr[0] != ',') {
    longdir = parseptr[0];
}
// format longitude data
if (longdir == 'E')
    longdir = '+';
else if (longdir == 'W')
    longdir = '-';
long1 = longitude/1000000;
long2 = (longitude/10000)%100;
long3 = (longitude%10000)*6/10000;

// groundspeed, not used
parseptr = strchr(parseptr, ',')+1;
// track angle, not used
parseptr = strchr(parseptr, ',')+1;

// date
parseptr = strchr(parseptr, ',')+1;
tmp = parsedecimal(parseptr);
date = tmp / 10000;
month = (tmp / 100) % 100;
year = tmp % 100;
}

void noTrack(void){
    section = '7';
gps();
    press = button();
    while(press != 5)
    {
        Wire.beginTransmission(4); // transmit to slave
        Wire.send(section);
        Wire.send(date);
        Wire.send(month);
        Wire.send(year);
        Wire.send(hour);
        Wire.send(minute);
        Wire.send(second);
        Wire.endTransmission();
gps();
delay(10);
    press = button();
}
tracking = true;
delay(1000);
Appendix D: ATmega328 code (Slave)
#include <Wire.h>
#include <LiquidCrystal.h>
#include <Servo.h>

LiquidCrystal lcd(7, 8, 9, 10, 11, 12); //LCD setup
Servo latServo; //servo object
int servoPos = 223; //servo time

//accel
byte degX, degY, degXdec, degYdec;
char signDegX, signDegY;

//compass
char compassSign;
int compass;

//stepper motor
#define motorPin1 4
#define motorPin2 5
byte tempMotorPin1, tempMotorPin2;

//GPS variables
int hour, minute, second, year, month, date;
int lat1, lat2, lat3, long1, long2, long3;
char latdir, longdir;

//degree character
byte deg[8] = {
    B01100,
    B10010,
    B10010,
    B01100,
    B00000,
    B00000,
    B00000,
    B00000,
};

void setup() {
//initialize the LCD
  lcd.begin(20,4);
  lcd.createChar(0,deg);
  lcd.clear();
  lcd.print("Welcome to");
  lcd.setCursor(0,2);
  lcd.print("  STAR TRAKKER");
  delay(3000);

//initialize stepper motor
  pinMode(motorPin1, OUTPUT);
  pinMode(motorPin2, OUTPUT);
  digitalWrite(motorPin1,LOW);
  digitalWrite(motorPin2,LOW);
}
//initialize the servo motor
latServo.attach(6);
latServo.writeMicroseconds(servoPos);

//initialize the I2C wire communication
Wire.begin(4);
Wire.onReceive(receiveEvent);
}

void loop() {
//continuous loop
  delay(100);
}

//I2C wire receive interupt
void receiveEvent(int howMany) {
  char x = Wire.receive();

  //intro section
  if(x == '0') {
    lcd.setCursor(0,0);
  }

  //servo section
  if(x == '1') {
    servoPos = Wire.receive();
    servoPos = servoPos*10;
    latServo.writeMicroseconds(servoPos);
    lcd.clear();
    lcd.print("Latitude angle:");
    lcd.setCursor(0,1);
    lcd.print("  Press -2- to INC");
    lcd.setCursor(0,2);
    lcd.print("  Press -2- to DEC");
    lcd.setCursor(0,3);
    lcd.print("Press -5- for next");
  }

  //Accel section
  else if(x == '2') {
    signDegX = Wire.receive();
    degX = Wire.receive();
    degXdec = Wire.receive();
    signDegY = Wire.receive();
    degY = Wire.receive();
    degYdec = Wire.receive();

    lcd.clear();
    lcd.print("Make stand level ");
    lcd.setCursor(0,1);
    lcd.print("  X-axis: ");

lcd.print(signDegX);
if(degX < 10)
    lcd.print("0");
lcd.print(degX, DEC);
lcd.print(",."");
if(degXdec < 10)
    lcd.print("0");
lcd.print(degXdec, DEC);
lcd.write(0);
lcd.setCursor(0,2);
lcd.print("  Y-axis: ");
if(degY < 10)
    lcd.print("0");
lcd.print(degY, DEC);
lcd.print(",."");
if(degYdec < 10)
    lcd.print("0");
lcd.print(degYdec, DEC);
lcd.write(0);
lcd.setCursor(0,3);
lcd.print("Press -5- for next");
}

//compass section
else if(x == '3'){
    compassSign = Wire.receive();
    compass = Wire.receive();
lcd.clear();
lcd.print("Turn the telescope");
lcd.setCursor(0,1);
lcd.print(" to 0 degrees");
lcd.setCursor(0,2);
lcd.print("Direction: ");
lcd.print(compassSign);
if(compass < 10)
    lcd.print("00");
else if(compass < 100)
    lcd.print("0");
lcd.print(compass);
lcd.write(0);
lcd.setCursor(0,3);
lcd.print("Press -5- for next");
}

//time and date after GPS acquired
else if(x == '4'){
    date = Wire.receive();
    month = Wire.receive();
    year = Wire.receive();
    hour = Wire.receive();
    minute = Wire.receive();
second = Wire.receive();
lcd.clear();
lcd.print("Date: ");
if(month < 10){
    lcd.print("0");
}
lcd.print(month);
lcd.print("");
if(date<10){
    lcd.print("0");
}
lcd.print(date);
lcd.print("/20");
if(year<10){
    lcd.print(" ");
}
lcd.print(year);

lcd.setCursor(0,1);
lcd.print("Time: ");
if(hour<10){
    lcd.print("0");
}
lcd.print(hour);
lcd.print("");
if(minute<10){
    lcd.print("0");
}
lcd.print(minute);
lcd.print(":");
if(second<10){
    lcd.print("0");
}
lcd.print(second);
lcd.print(" UTC");
lcd.setCursor(0,3);
lcd.print("Press -5- for next");
}

//location after GPS acquired
else if(x == '5'){
    latdir = Wire.receive();
    lat1 = Wire.receive();
    lat2 = Wire.receive();
    lat3 = Wire.receive();
    longdir = Wire.receive();
    long1 = Wire.receive();
    long2 = Wire.receive();
    long3 = Wire.receive();

    lcd.clear();
lcd.print(" Lat: ");
lcd.print(latdir);
if(lat1 < 10) {
    lcd.print("00");
} else if(lat1 < 100) {
    lcd.print("0");
} lcd.print(lat1);
lcd.write(0);
lcd.print(" ");
if(lat2 < 10) {
    lcd.print("0");
} lcd.print(lat2);
lcd.print(" ");
if(lat3 < 10) {
    lcd.print("0");
} lcd.print(lat3);
lcd.print(" ");

lcd.setCursor(0,1);
lcd.print("Long: ");
lcd.print(longdir);
if(long1 < 10) {
    lcd.print("00");
} else if(long1 < 100) {
    lcd.print("0");
} lcd.print(long1);
lcd.write(0);
lcd.print(" ");
if(long2 < 10) {
    lcd.print("0");
} lcd.print(long2);
lcd.print(" ");
if(long3 < 10) {
    lcd.print("0");
} lcd.print(long3);
lcd.print(" ");
lcd.setCursor(0,3);
lcd.print("Press -5- for next");
}

//tracking mode
else if(x == '6') {
    tempMotorPin1 = Wire.receive();
    tempMotorPin2 = Wire.receive();
    if(tempMotorPin1 == 0)
        digitalWrite(motorPin1, LOW);
else
    digitalWrite(motorPin1, HIGH);

if(tempMotorPin2 == 0)
    digitalWrite(motorPin2, LOW);
else
    digitalWrite(motorPin2, HIGH);

lcd.clear();
lcd.print("Tracking...");
lcd.setCursor(0, 1);
lcd.print("Press -4- to REVERSE");
lcd.setCursor(0, 2);
lcd.print("Press -6- to ADVANCE");
lcd.setCursor(0, 3);
lcd.print("Press -5- to exit");
}

// Standby mode
else if(x == '7'){
    date = Wire.receive();
    month = Wire.receive();
    year = Wire.receive();
    hour = Wire.receive();
    minute = Wire.receive();
    second = Wire.receive();
    lcd.clear();
lcd.print("Date: ");
    if(month<10){
        lcd.print("0");
    }
    lcd.print(month);
    lcd.print("/");
    if(date<10){
        lcd.print("0");
    }
    lcd.print(date);
    lcd.print("/20");
    if(year<10){
        lcd.print("0");
    }
    lcd.print(year);
    lcd.setCursor(0, 1);
    lcd.print("Time: ");
    if(hour<10){
        lcd.print("0");
    }
    lcd.print(hour);
    lcd.print(":");
    if(minute<10){
        lcd.print("0");
    }
    lcd.print(minute);
    lcd.setCursor(0, 2);
}
}  
lcd.print(minute);  
lcd.print(":");  
if(second<10){  
lcd.print("0");  }  
lcd.print(second);  
lcd.print(" UTC");  
lcd.setCursor(0,3);  
lcd.print("Press -5- to track");  
}  

//while acquiring GPS signal  
else if(x == '8'){  
lcd.clear();  
lcd.setCursor(0,0);  
lcd.print("Aquiring GPS signal");  
lcd.setCursor(0,1);  
lcd.print(" Please wait...");  
}  
}  
}
Appendix E: Schedule

Week 1
• Research for component design and procurement
• C programming review for microcontroller development

Week 2
• Research for component design and procurement
• C programming review for microcontroller development
• Initial modular product design

Week 3
• Research for component design and procurement
• Continue modular product design

Week 4
• Research for component design and procurement
• Continue modular product design

Week 5
• Continue modular product design
• Initial product procurement

Week 6
• Continue modular product design
• Finish product procurement

Week 7
• Continue modular product design
• Begin modular construction and testing

Week 8
• Finish modular product design
• Continue modular construction and testing

Week 9
• Continue modular construction and testing

Week 10
• Finish modular construction and testing
• Begin system level construction

Week 11
• Continue system level construction
Week 12
  • Finish system level construction
  • Begin system level testing

Week 13
  • Continue system level testing

Week 14
  • Work on report
  • Finish complete system level testing

Week 15
  • Finish report
Appendix F: Additional Pictures

Final configuration, side A
Final configuration, side B
Final configuration, side C
Final configuration, side D
Final configuration, side E
Final keypad configuration
Final configuration, close up on the main board