Final Project Report:
GPS Celestial Object Locator

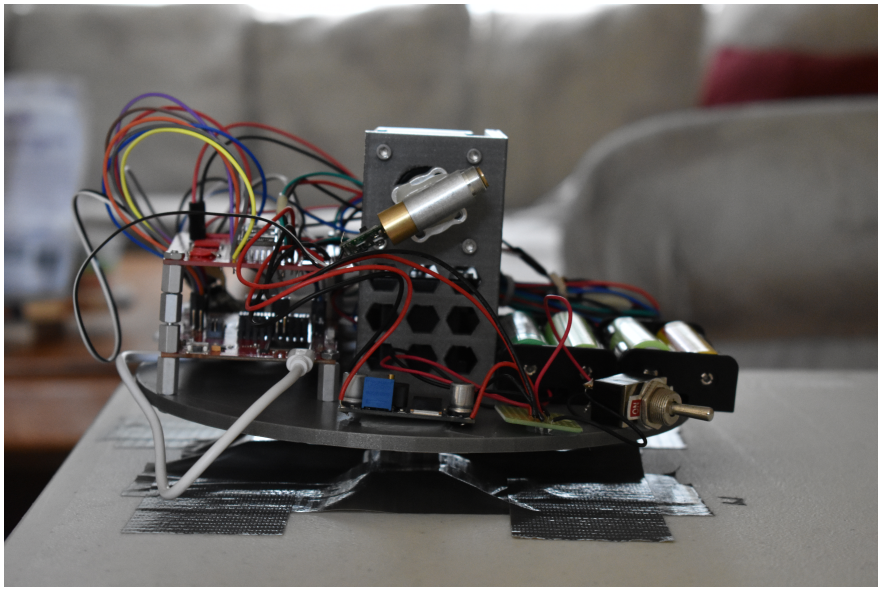
Clint Olsen

Joshua Biggio

Eric Daugherty

ECEN 2020: Applications of Embedded Systems

December 15, 2016



University of Colorado
Boulder

Introduction

The goal of this project was to construct a GPS based celestial body locator, centered around the MSP432P401R microcontroller. The design consists of 2 stepper motors that rotate based on the calculated altitude and azimuth of the desired body. One motor with an attached laser pointer rotates to the desired altitude angle and the other rotates the connected platform to the desired azimuth angle. The calculation of these parameters rely on the latitude, longitude, Local Hour Angle (LHA) of the object, Greenwich Hour Angle of the object and declination of the object. The latitude and longitude of the user's location is determined through the use of the GP-20U7 GPS receiver which communicates with the MSP via UART. The GHA and declination for the object vary based on the date and time, so these values are input by the user into a Bluetooth serial smartphone application that sends the data wirelessly to the device via UART. Finally, the LHA is calculated based on the received longitude and GHA. The software, hardware and mathematical implementation and design of the device is described in the following sections.

Firmware Development

The project relies on the use of 5 major software modules (C source files) to control the operation of each component: GPS, UART, logger, locator, and motors. Each module works in tandem with the others through sequential reference to previous data. The UART module is one of the most fundamental because it controls all of the input for the device. Both the GP-20U7 GPS receiver and HC-05 Bluetooth module work through UART, so one of the main functions of this module is the configuration of these devices. In order to discretize these input components, 2 UART modules were configured, one for the GPS (UCA0) and one for the Bluetooth module (UCA2). Luckily, the devices both require similar configuration such as the same baud rate, data frame size and structure, and receive interrupts, so the setup was simple. The UART software module also contain the functions that allow for UART transmission from the MSP. In the case of this project, the output was to the Bluetooth module to view data on a smartphone.

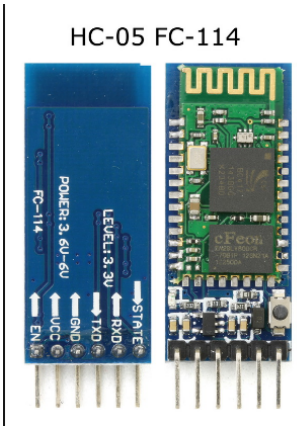
The GPS software is the first major step in the function of the device. Once the GPS receiver has found a lock, it begins receiving NMEA strings, which in the case of the GP-20U7, comes in 6 different protocols. Each protocol contains different sets of information, labeled with a message ID: GPRMC, GPGGA, GPGLL, GPGSA, GPGSV, and GPVTG. This project utilizes GPRMC, which is the minimum recommended GPS data, containing UTC time, latitude, longitude, and date. The latitude and longitude are needed directly for calculations, and the date and time are used for logging purposes. The first operation of the GPS software module is to identify when the GPRMC message ID is received via UART. This and all data collection is centered around the use of UCA0's receive interrupts. The collection starts when the $ character is received, because this character is the prefix to all IDs. Once the start

Figure 1: *GP-20U7 GPS Receiver*

of a ID has been identified, each message ID is stored as it comes in byte by byte until the standard length of the message ID is reached (5 bytes). From here, this is compared with a predefined string that holds GPRMC. Next, the data validity flag is checked within the NMEA string and if both this and the message ID are confirmed, the actual data collection begins. Originally, the data was processed one byte at a time for the entire duration of the string straight from UART, but the processing time of checking and storing the data caused issues with keeping up with the rate of the incoming data. To rectify this, the entire string was instead stored in an array of uint8_t's and from here, the data was passed through the processing code one byte at a time with no time constraint. The actual processing of the string was based around the consistent nature of the NMEA strings in terms of the size of each piece of data and the order it arrived. In the processing function, there are flags that indicate which piece of data is being collected. For example, if latitude is the piece of data that is being looked for, the corresponding flag is set. Based on which flag is set, the program enters the corresponding conditional block that handles that information uniquely, depending on what it is. This is done for time, latitude, latitude direction (N or S) longitude, longitude direction (W or E) and date. The storage of all of this information is in the form of a GPS info structure which is defined in the gps.h file for access from other modules. The final function of this module is an ASCII to double converter, which translates the array of uint8_t ASCII latitude and longitude to usable numbers for calculations later in the program. This was done by knowing that the first three characters corresponded to degrees and the last two corresponded to minutes. From here calculations were performed based on the number itself and using the power of 10 that correlated to that position in the number (hundreds, tens, etc.). The minutes also had to be converted to a fraction in degrees for the correct angle.

With this data established, the program is then held in an idle state while it waits receive interrupts from the other UART, UCA2, which is input from the Bluetooth module. The input from the Bluetooth is two values, Greenwich Hour Angle (GHA) and declination, which have to be looked up for the object based on the current time and date. The input format for this data was arbitrarily made to be 5 characters long, 3 characters for degrees and 2 for minutes, for simplicity of conversion to doubles later. This is where the locator module is introduced, because these input values are stored in another structure defined in locator.h, which holds GHA, declination, altitude, azimuth, and other relevant information to the object itself.

Figure 2: *HC-05 Bluetooth Module*

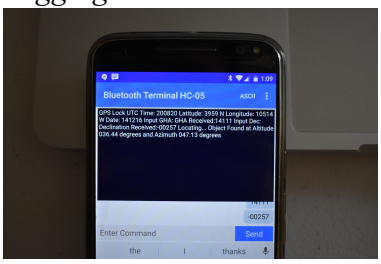


The locator module is dedicated to the calculation of LHA, altitude, and azimuth from the equations shown in the Mathematical Development and Modeling section of the report and converting these values to uint8_t arrays in order to be logged using UART. This module utilizes the math.h library due to the numerous trigonometric calculations that have to be done to calculate these values. The other important function is some preparation for the motor module. In order for the motors to know how far to rotate, the altitude and azimuth angles must be translated into discrete steps for the stepper motors to understand at their configured resolution. These calculations are performed in this module, but will be discussed more in depth in the motor module description. Finally, the location data structure that is defined in the locator.h file holds the double and uint8_t array versions of GHA, declination, altitude, azimuth, as well as values for LHA, altitude steps, and azimuth steps for use in other portions of the program.

Next, the motor module takes over and uses the location data to appropriately move the motors. The stepper motors that were used (described in more detail in Hardware Development) are frequency modulated and step on the rising edge of a signal. The motors were also configured to 1/32 microstepping, which provided an angle resolution of .05625 degrees per step. Using this resolution, the locator module calculates the number of steps based on the calculated angles for the motor module to use. In order to pulse the motors the appropriate amount, Timer A0 was configured to output a 1 kHz square wave from a GPIO pin. The configuration of the timer consisted of enabling capture compare interrupts, selecting the 3 MHz SMCLK as the clock source with no prescalar in up-mode, and setting the capture compare register to the value correlating to 1 kHz, 3000. The motor module also defined functions that enable or disable each motor by setting or clearing the direction and enable pins for each motor. In order to to have the timer count the correct number of pulses for a specific angle, the ISR for Timer A0 was set up with a global counter variable that counts to twice the calculated number of steps (due to the interrupts being triggered on falling edge as well and the motors only respond to rising edge). The ISR is also broken up into two conditional statements that are controlled by flags, indicating which motor pin to toggle (altitude motor or azimuth motor). When both motors have been stepped the desired amount, the ISR toggles the pin controlling the laser pointer, turning it on to point at the located object.

The logger module does not follow a chronology like the other modules, and is integrated throughout the program. This module defines the log messages to be output to the Bluetooth module by using predefined messages, and fetching desired data from the GPS and location info structures. This information is then assigned to the UCA0 TX buffer one byte at a time, outputting it to a smartphone via Bluetooth. The items that are logged throughout the program are the time, date, latitude, longitude, input GHA and declination, “GPS Lock” message, “Locating…” message when the motors are being stepped, and “Object found at X Altitude and Y Azimuth” when the motor movement is complete.

Figure 3: *Bluetooth Logging*



The final software piece to discuss is main.c, which is used to oversee the control of all of these modules, define the ISRs, and declare instances of the info structures. In main, all of the initial configuration is completed, and the program enters an infinite while loop. The loop contains several different conditional statements that per-

tain the stages of the program such as processing NMEA strings, gathering Bluetooth input, starting calculations, and starting the motors. This section acts as a state machine that is controlled by the status of interrupts, which also contain conditionals that control when the operation is completed, from the peripherals. When a certain termination condition is met within the ISR or a specified function is completed, the flag for that section of the infinite loop is cleared and the next one in sequence is set. This progresses the program to the next state and repeats. An example of this is the transfer from the initial data collection state to the "process string" state. Once the UCA0 ISR realizes that the full string has been stored by seeing the line feed character that terminates all NMEA strings, the process string flag is set, pushing the program into the next state of parsing the stored string and storing it in the GPS info structure. [†‡]

There is still a large opportunity for improvement in terms of software in this project. Some of the main ideas that we did not have time to add were more comprehensive error checking and response and a robust reset option. As it stands, the devices must be manually reset, but software implementation to make the device retrace its motor movements and idle for input again would be ideal. In terms of error checking, options to regather GPS data and re-input invalid information would be ideal to make the design more user friendly.

Hardware Development

Figure 4: *Nema 17 Stepper Motor*

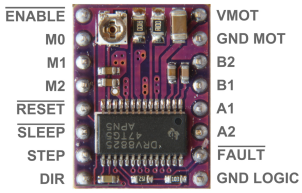
The physical device was also designed in parts and was based around stepper motors. The motors used in this project were 2 amp Nema 17 stepper motors that are normally used for home built 3D printers. These motors can be modeled as two inductors and are driven at a relatively high voltage. In this case, we chose to drive our motors with a 15 volt source that falls well within the 8 to 45 volt range that the motors are spec'd for. This value of 15 volts was decided upon based on our choice of power supply, which is described later. In order to drive these motors, DRV8825 stepstick motor driver boards were sourced from RepRapChampion, an online source for 3D printer parts. These driver boards are highly configurable with options like low power, sleep, and fault inputs and outputs. Both of our motor drivers were configured to always reset after a fault (overvoltage, overcurrent, or otherwise), never enter sleep mode and have 1/32 microstepping by soldering appropriate jumpers to high or low logic levels. The rest of the configuration is done through a GPIO

[†]The project in it entirety can be found in the following GitHub repository: https://github.com/cmolsen3/Motor_with_Bluetooth_Input.git

[‡]The software block diagram can be found in the Appendix

interface with the MSP as discussed on the code above in the firmware section. With this configuration, the stepstick drivers ended up working flawlessly, save one incident where the magical blue smoke was released during the breadboarding stage of development. This prompted a hurried re-order of extra motor driver boards along with new voltage regulator boards. As it turns out, the reason why the motor driver chip failed was due to the step pin being pulsed while the enable pin was held low (which is on). Thus, the protocol for moving the motors was changed to first setting the direction, then enabling the driver before any pulsing took place.

Figure 5: *DRV8825 Motor Driver*



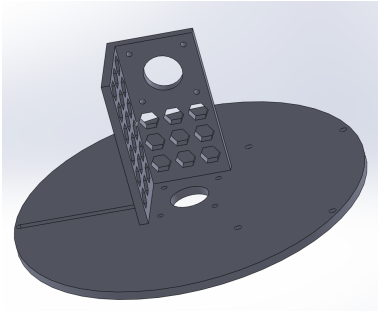
The power delivery system is pretty straight forward. 15 volts turned out to be a nice round value that could easily be maintained by 4 18650 lithium ion batteries. These batteries were ideal for this application because they pack a large capacity into a relatively small area and have a high enough voltage (3.7 volts plus) to merit the need for only a few of them. This was chosen so as to not need to pack a dozen batteries onto the device. The 15 volt raw source was wired directly into the motor driver voltage inputs on the motor drivers and stepped down to 3.3 volt logic levels using an LM2596 step down converter. It was originally planned that the MSP would be powered off this source, but this proved to be more difficult and time consuming than anticipated. Instead, the tried and true method of powering the MSP with a portable battery bank was used. Lastly, the laser pointer was powered off the 3.3 volt stepped down voltage and not the MSPs onboard power delivery system. This was done by connecting the laser pointer to a bc337 transistor whose base was connected to an output pin on the MSP.

Figure 6: *LM2596*

The mechanical device platform was based on the degrees of motion needed to point towards any star in the hemisphere above the device. A rough sketch that allowed for the motors to both be mounted above the central axis turned into a 3D part in Solidworks that included mounting holes for the battery packs, MSP and motors. A base for the turning table to rotate against was also designed in this manner. Both of these parts were printed using a FABTotum and took over 28 hours to print. With both the physical and electrical portions of the hardware designed, all that was left was to build the device and push code to it. Thanks to the careful planning of both the firmware and hardware, the prototype worked right out of the gate once built into its final state, and only minor tweaks were necessary to achieve the final result.[†]

[†]The hardware block diagram can be found in the Appendix

Figure 7: *Model of the platform in SolidWorks*



Mathematical Development and Modeling

This project relied heavily on the calculation of the altitude and the azimuth of a celestial body. Altitude corresponds to the angle the object is above the horizon and the azimuth is represented as an angle west of south. The equations for the altitude and azimuth are as follows:

$$Altitude = \sin^{-1}(\cos(LHA)\cos(Declination)\cos(Latitude)+\sin(Declination)\sin(Declination))$$

$$Azimuth = \tan^{-1}[\frac{-\sin(LHA)}{\tan(Declination)\cos(Latitude) - \sin(Latitude)\cos(LHA)}]$$

These equations will produce an angle, ±90 for the altitude and ±180 for the azimuth of the object. Positive values follow the convention stated above but a negative altitude means the object is below the horizon and a negative azimuth is an object to east of the observer. To simplify the functions of the motors we eliminated negative numbers by returning an error in the case of a negative altitude and by adding 360 to a negative azimuth. In the case of the azimuth this will not the change the angle our locator is positioned at, it only simplified the control of the motors since we did not need to configure the direction of the motors every time we calculated new values for altitude and azimuth. The equations above rely on values for latitude, declination, and LHA. The latitude comes from the GPS module and we gathered our declination from a table. We had to calculate a value for the LHA, which is the local hour angle. This relies on adjusting GHA, or Greenwich hour angle, for the observer's current longitude and can be done simply by using the equation

$$LHA = GHA + Longitude$$

The longitude was another value we gathered from our GPS while the GHA was found in the same table as the declination.

Originally we attempted to calculate the GHA using right ascension, the date, and time. This method required many more calculations to be done which would propagate small rounding errors through our system leading to altitudes and azimuths to be off by, at times, up to 5 degrees. This is

not a huge error; however, the distance these objects are from us means even small angles appear to be extremely far from the intended target. The highlight of this project was not to calculate everything using minimal inputs so accuracy was chosen over a more simple user experience.

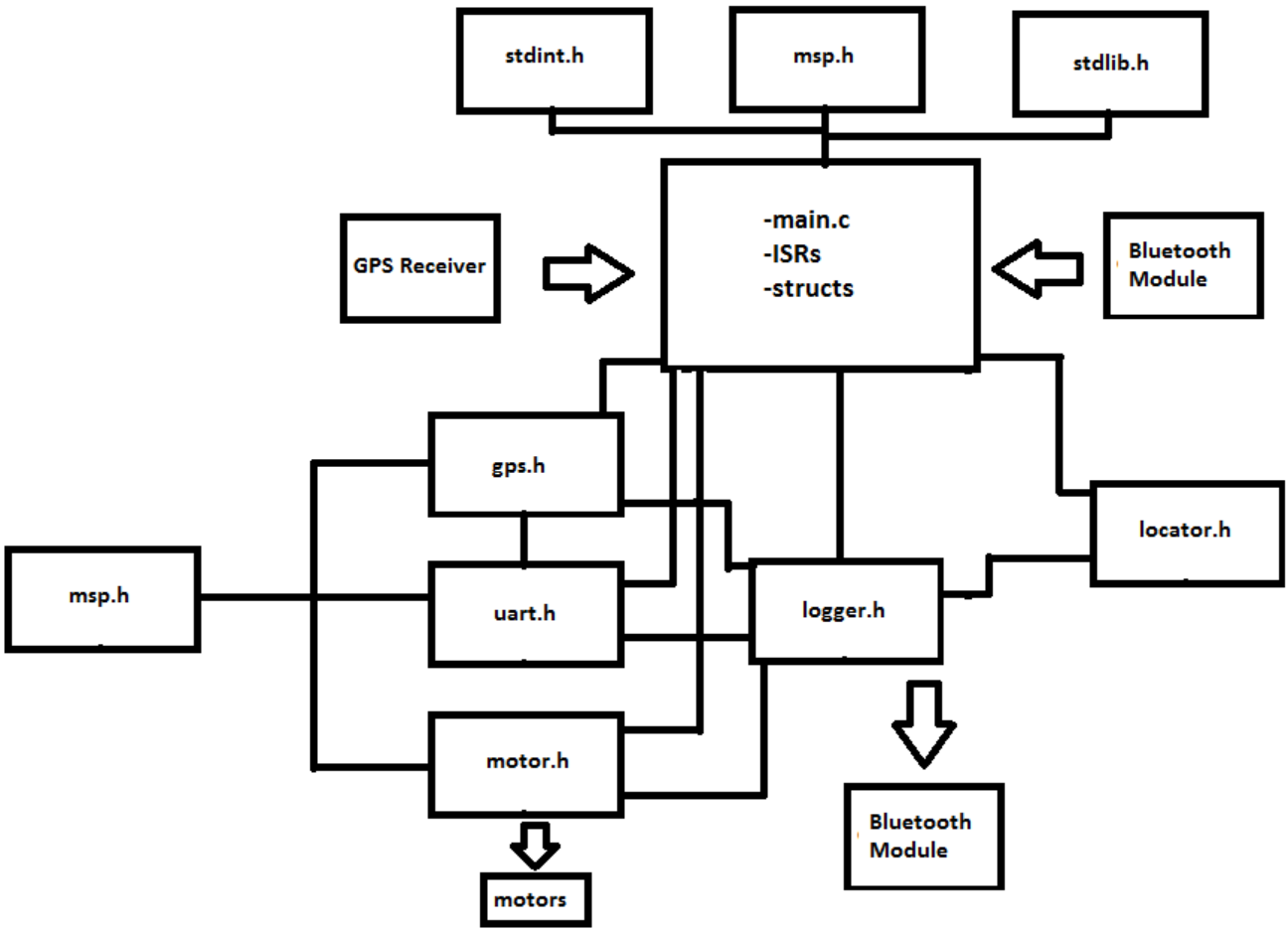
The locator source file utilizes the math.h library to perform all the trigonometric functions in our altitude and azimuth equations. These functions require the inputs to be in the form of radians. In the case of arcsin and arctan, the outputs are also in radians. The value of latitude, from the GPS, is in degrees and minutes. First we needed to calculate decimal degrees and this is done by dividing the minutes by 60 then adding it to the degrees. This had to be implemented in a different way in the code due to our ASCII to Double conversion but it used the same idea. The value in decimal degrees can then be multiplied by π and divided by 180 to convert it into radian. For LHA we needed to add GHA and longitude together. Longitude is in the form of degrees and minutes just like latitude so we needed to convert GHA into the same form. This required multiplying the hours of the GHA by 360 and dividing by 24, since there are exactly 24 hours in a circle. After LHA had been calculated it could then be converted into radians in the same was latitude was, first getting decimal degrees then converting. Declination was also in the form of degrees and minutes so its conversion was the same as latitude. All converted values could then be passed into our altitude and azimuth functions. After altitude and azimuth were calculated in radians we then converted both values back into degrees by multiplying by 180 and dividing by π. This extra step was for the purposes of logging since visualizing degrees is more intuitive than radians.

Conclusion

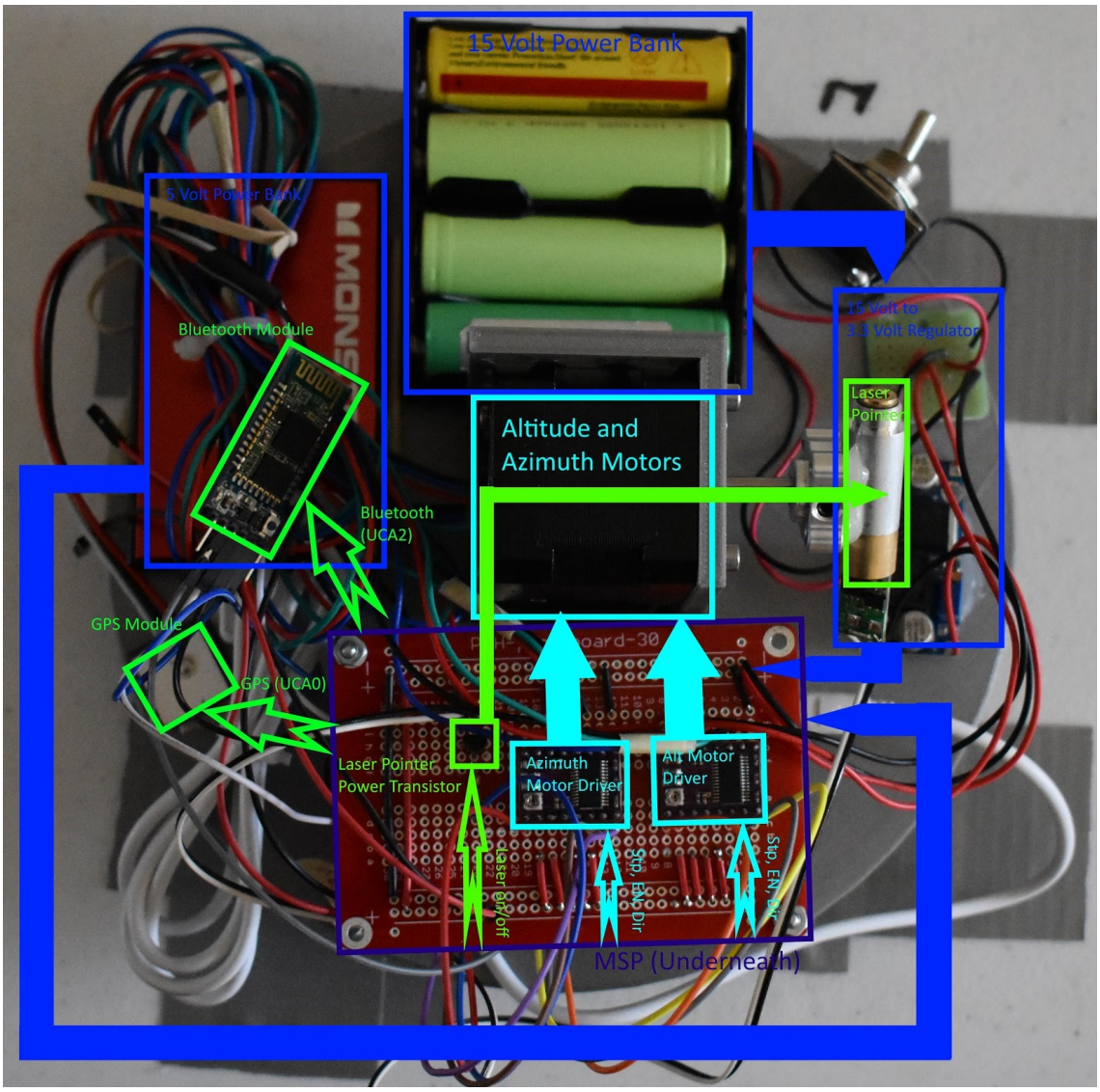
The process for completing this project involved breaking the work into different areas and subsections. The major parts were mathematics, hardware, and software. From here, these were broken up even further and defined from a qualitative level before implementation. This made the task for each function very clear and easy to integrate. We utilized skills from each of the team members to divide and conquer, working together to make this project possible by building on each other's strengths. For all three of us, this project will be something that we will look back on as something great we accomplished during college. We took pride in developing, prototyping, coding and constructing this project, and became more confident in the field of engineering along the way. We explored the field of embedded engineering in class, then made it our own. As a result, all three of us plan to continue working and studying in the field of embedded engineering.

Appendix

Software Block Diagram



Hardware Block Diagram



Bill of Materials

Bill of Materials					
Part	Part Number	Distributor	Unit Price	Qty. Used	Line Price
Nema 17 Stepper Motor (2A)	17HS16-2004S	Stepper Online	$7.51	2	$15.02
DRV8825 StepStick Driver		RepRapChampio	$3.49	2	$6.98
DC-DC LM2596 Step Down Converter		RepRapChampio	$0.99	1	$0.99
ProtoBoard - Square 1" Single Sided	PRT-08808	Sparkfun	$1.50	1	$1.50
SparkFun Solder-able Breadboard	PRT-12070	Sparkfun	$4.95	1	$4.95
Lithium Ion Battery - 18650 Cell (2600m	PRT-12895	Sparkfun	$5.95	2	$11.90
Battery Holder - 2x18650 (wire leads)	PRT-12900	Sparkfun	$1.50	2	$3.00
Set Screw Hub - 5mm Bore	ROB-12404	Sparkfun	$4.99	2	$9.98
Laser Module - Green	COM-09906	Sparkfun	$14.95	1	$14.95
Bluetooth Serial Pass-through Module	HC-05	DSD TECH	$9.00	1	$9.00
GPS Reciever	GP-20U7	Sparkfun	$19.95	1	$19.95
Toggle Switch SPST	GTS447N101HR	Digikey	$3.65	1	$3.65
					$101.87