Spring 2018

ECE 103 Engineering Programming

Final Report

Abdulrahman Alamar: aalamar@pdx.edu

Chuck Faber: cfaber@pdx.edu

Ebtehal AlEnezi: ebte@pdx.edu

Portland State University

Introduction

In this project we attempted to develop a tracking beacon for a high altitude balloon. The whole project had to be coded in C, and developed mostly by ourselves with limited use of pre-developed libraries. Despite not having a completely functional product at the end ready for flight, we have a lot of the key pieces necessary completed to make flight possible, and we have a much much deeper understanding of the concepts and thinking that goes into developing something like a tracking beacon that seems simple on its surface yet is anything but.

Hardware

Description

Our beacon's heart is the Arduino Mega 2560 which was chosen because it has multiple serial ports, and greater SRAM than the Uno. The other two main pieces of hardware to collect location data and send it are the Radiometrix HX1 radio transceiver which operates at the APRS frequency which is 144.390 MHz, and the Ublox Neo-6M GPS module which retrieves GPS data from the Global Navigation Satellite System (GNSS). Various sensors such as humidity, temperature, pressure, and acceleration sensors were added to collect telemetry data for possible future analysis of the upper atmosphere environment.

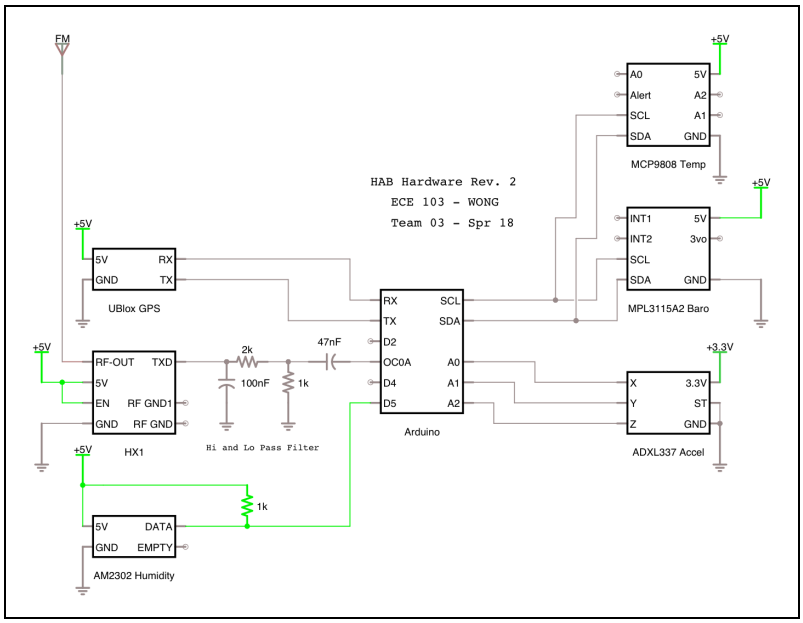


Figure 1: Hardware Schematic

Some of the sensors communicate using the I^2C communication protocol and thus require use of the clock and data pins on the Arduino. The GPS module requires the use of a serial port (one of the TX/RX lines). And the radio requires the use of a PWM pin which generates a sine wave. The radio module also requires a low pass filter (resistor with a cap to ground) to blend the square waves produced into a sine wave. The high pass filter is to make the amplitudes of the two frequencies we are generating more even.

Bill of Materials

Arduino Mega 2650: $ 14.99

Radiometrix HX1: $ 38.46

Ublox Neo-6M GPS module and antenna: $ 15.66

ADXL337 Acclerometer: $ 11.95

MCP9808 Temperature Sensor: $ 11.49

MPL3115A2 Pressure/Altimeter Sensor: $ 11.99

AM2302 Humidity/Temperature Sensor: $ 8.99

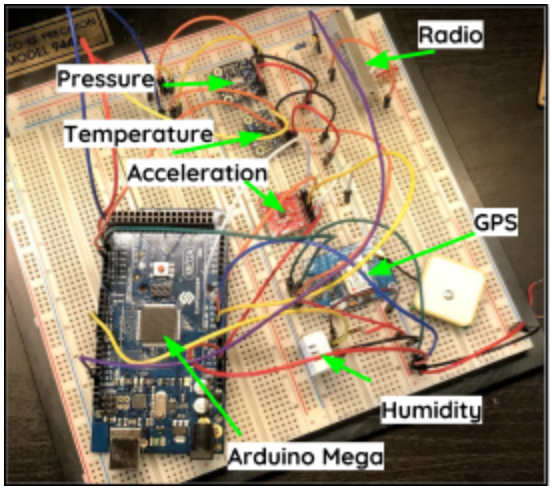


Figure 2: Constructed Hardware

Software

Our software was divided up into several distinct modules: GPS, sensors, radio, and the front end. As such we will discuss the problem descriptions and design approaches separately.

GPS Problem Description

The GPS segment of the software is required to make a connection to receive GPS telemetry to the orbiting satellites in the GNSS. It is then required to parse NMEA sentences (NMEA: National Marine Electronics Association -- a GPS data protocol) for information regarding the beacon's latitude, longitude, heading, altitude, and time, as well as the number of satellites it currently has a fix on. This data must be saved in the global telemetry data structure.

GPS Design Approach and Assumptions

The Neo-6M produces NMEA formatted strings which it reads to the serial port. For our purposes we decided to only parse one or two NMEA sentences (the module produces many). Data from the GPS is sent byte by byte to the serial port and thus our code had to operate byte-wise adding characters to a buffer, and then parsing that buffer to see if it had the sentence we were looking for. Finally the code parses that sentence for data and only saves data to our global telemetry struct if the fix is valid so that we don't inadvertently waste a beacon transmission on bad data. We needed to ensure that the code did all of the processing before gathering another sentence into the buffer thus, the functions call other functions within themselves to enact a chain that occurs sequentially completing all the parsing before looking for another sentence.

Sensors Problem Description

The sensors need to accurately represent temperature, altitude, pressure, and humidity data and record these values into our telemetry data structure.

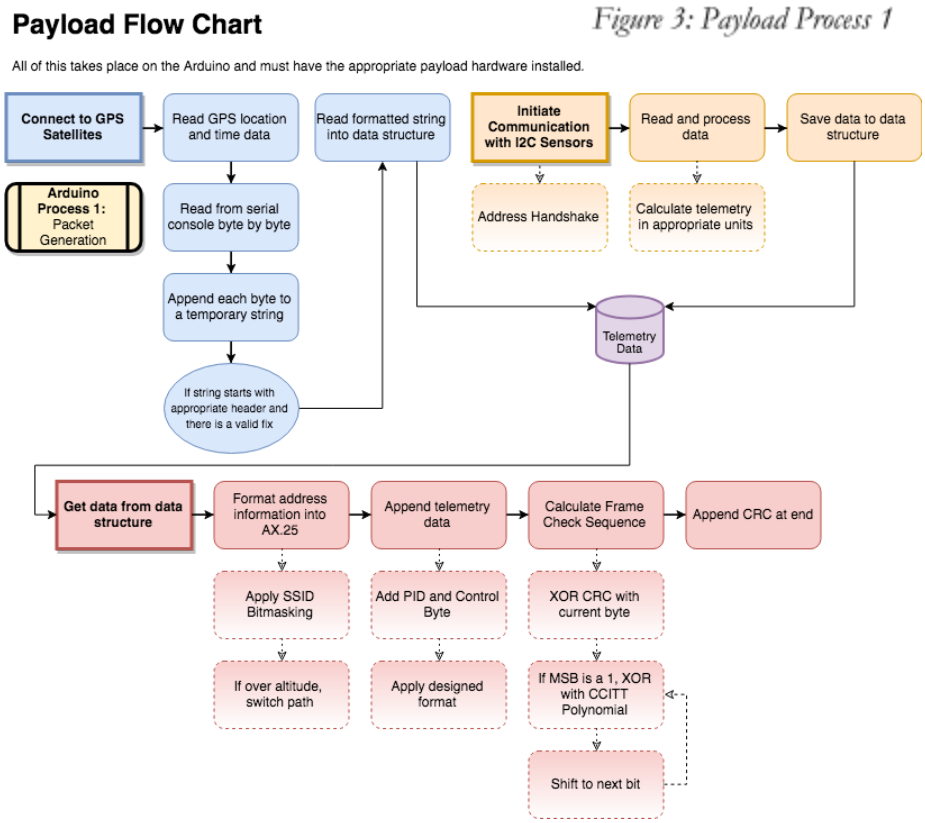


Figure 3: Payload Process 1

Sensors Design Approach and Assumptions

Currently we have two sensors functioning which are the temperature sensor, and the altitude/pressure sensor. Both of these sensors communicate using the I^2C protocol. This protocol requires connecting all sensors to the same clock and data pins on the microcontroller and communicating with each one using each device's individual address. In this case the Arduino acts as the "master" and each sensor acts as the "slave". In order to facilitate the implementation of sensors in this project we use the Wire.h library which is the default I^2C communication library for the Arduino.

Radio Problem Description

The radio component of this project can be further broken into two parts: AX.25 protocol packet generation (this is the encoding protocol for using APRS servers), and frequency modulation for transmission. The AX.25 packet protocol includes a specially formatted address component which must include the call sign of the user (ours is K5TRL) and the SSID of the device (we chose 11). It must also include the destination (APRS in our case), and the 'vias' which is how many hops the packet should travel between digipeaters. After the address segment, it must transmit a control byte and PID byte, followed by the actual data which will not be altered by the APRS server. The packet must end in a frame check sequence (aka a cyclic redundancy check) which is calculated bit by bit throughout the entire packet. If it isn't calculated properly the server will reject the message as it will assume the message is fragmented. The packet must start and end with a flag byte of 0x7E (01111110) to signal that the packet is beginning or ending.

Transmission occurs through frequency modulation. The script must produce a sine wave at 1200Hz and 2200Hz. By switching or not switching we produce a '0' or '1' bit. It must also keep track of the number of

ones, and "stuff" a 0 bit into the message if the count reaches five '1' bits so it doesn't inadvertently create an ending flag byte too early in the message.

Radio Design Approach and Assumptions

Addresses in the address section of the AX.25 packet must be 7 characters long including the SSID byte. If the callsign is less than 6 characters, spaces are added. The SSID was constructed by shifting the ID by one to the left and OR masking it with 0x60 or 0x61 to put it into the 011 XXXX 0/1 format. The "via" is "WIDE1-1, WIDE2-1" if the payload is below 1500m of altitude, and just "WIDE2-1" if it is above this altitude (because there are less obstructions). This was implemented with an if statement reading the altitude data from the GPS.

The data was constructed by accessing the telemetry struct to print the data in to the format we designed. The whole string is produced including the address, control and PID bytes, and data string. The CRC function takes in a pointer to the string to get one character, and gets its MSB. If it is a high bit, then it shifts the current CRC value over by 1, and XORs it with the pre-defined "polynomial" value 0x1021 (for MSB CRC calc). If it is a low bit, it still shifts it over by one and gets the next MSB (see "Computation of Cyclic Redundancy Checks -- Wikipedia"). The 16 bit CRC is then divided into a high byte and low byte and appended to the message.

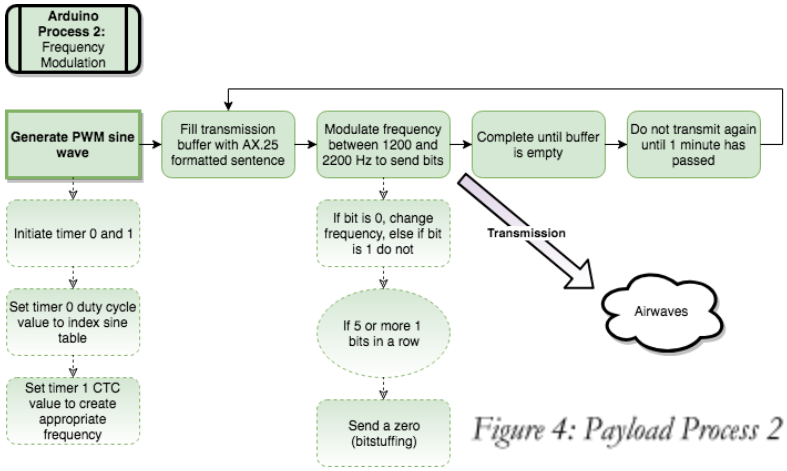


Figure 4: Payload Process 2

To generate a sine wave, Timer 0 on the Arduino is set to Fast PWM mode, which generates a voltage between 0 and its max value by varying the duty cycle. The duty cycle is altered by changing the output compare register (OCR) value. We create the appropriate sine form by having the OCR value iteratively change its index on a look-up table which is pre-generated and stored in the flash memory. We chose 64 samples. The frequency is controlled through timer 1, which is set to Clear Timer on Compare (CTC) mode. The sampling rate (rate at which timer 0 is called to run through the sine table) must be calculated using the following equation: sample_rate = (freq desired) * (# of samples). Then the OCR for timer 1 is calculated using the following equation: OCR = (CPU Clock speed) / (sample_rate). This will produce two OCR values for timer 1, one at which the sample rate produces a 1200Hz wave, and the other produces a 2200Hz wave. The timers have to be initialized by setting particular bits in registers associated with each timer to particular configurations which are outlined in "How to Modify the PWM Frequency on the Arduino" (see references).

The AX.25 packet is then added into a buffer and the head and tail of the buffer is defined. A third timer (timer 2) is initialized to trigger at 1200Hz. This encoding trigger and will send out bits 1200 times a second. The sinewave generation and transmission are done in two interrupt sequences. The encoding interrupt sequence checks if the transmission buffer is empty or not. If it is it does nothing. If it isn't, it sends an AX.25 flag (0x7E) to signal a new packet. Then it iterates through each character in the buffer, and goes bit by bit.

The encoding interrupt sequence includes a table with the two OCR values for timer 1 to switch frequencies. If the next bit is a zero it does a negation operation (switching 0 to 1 or 1 to 0) on the table index, which sets the global frequency to the new frequency value. If the bit is a 1 and it isn't currently sending the AX.25 flag, then it doesn't change the frequency and increments the 'ones' counter by 1. If there are 5 or more counts of '1' bits in a row, then it switches the frequency to stuff in a '0' bit and resets the counter back to zero. Also any '0' bit resets the 'ones' counter back to zero. The buffer is declared as empty when the head index of the buffer is the same as the tail index of the buffer.

Front End Problem Description

The front end runs on any internet connected computer and must connect to the APRS server, gather the raw data packets associated with our call sign, and then save them to a file. The program must then parse this file to get up to date data from the latest packet and display it on the console, ideally with a map.

Front End Design Approach and Assumptions

The front end uses the socket.h library to open a TCP connection to an APRS server. To handshake with the server, a proper login command must be given to it in the following form:

user K5TRL **pass** 8325 **vers** ece103group3 1.0 **filter** b/K5TRL*

Bold terms are required keywords. The "user" must be a valid callsign, and the passcode is generated based on this callsign. "vers" describes the program and version making the connection. Finally "filter" can be used with pre-defined server filter commands to filter packets based on desired settings. Our program opens a socket connection and converts the file descriptor (an integer) into a file stream (FILE * pointer), so we can use the File I/O commands we learned in class. The hostname of the server and port information is saved in a structure, which is called using the connect() function to open a connection. From there, knowing the general form of the output allows us to read the data packets. The first line generally produced is information about the server. The second line produced after writing our login string to the socket is generally a status on the success of our login. From there each line afterwards is a raw data packet or a server ping. Server pings are filtered from being written to our file by omitting strings beginning with '#'. All other strings are written to RawPackets.txt by appending them. The socket pointer must be rewound after each read operation to read the next string or else it is left at the end of the stream.

After the data is saved to the RawPackets.txt file, it is read by a separate script which parses and displays this data. It currently uses the strtok() function with commas as delimiters to save the data from the raw packet into a structure, which it then displays using the colors in the conio.h library. We may in the future use formatted file I/O commands to gather the data and remove extra delimiters that are present in the data.

Front End Logic Flow

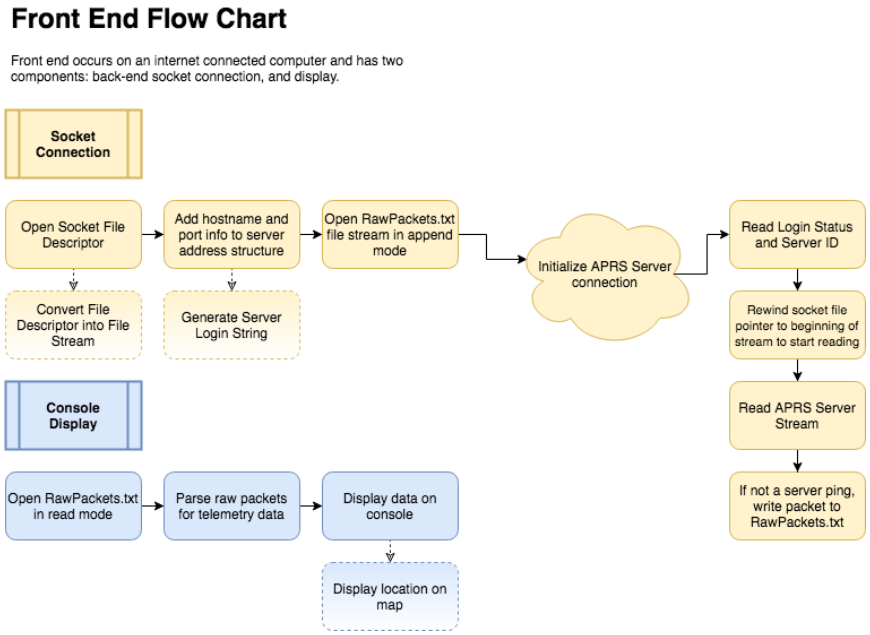


Figure 5: Front End Logic Flow

Problems and Solutions

GPS Problems

One problem with working with this GPS module with an Arduino Uno (which was what was used for debugging and testing) was that the GPS took up the only serial port on the device, and so we needed to use the SoftwareSerial library to create a virtual serial port that we could use to see the data that the GPS was producing. This is not needed on the Mega as it has multiple serial ports. We started with attempting to parse the GPRMC sentence. Eventually we changed to parsing the GPGGA sentences as they included the data that we required all in a single sentence. We initially had many issues trying to parse the sentences. We believe the issue was that the GPS data was changing too quickly and editing the data buffer as it was being read.

Sensors Problems

It was a struggle to figure out how the the Wire.h library worked and how the I^2C protocol worked. It was also a challenge to find the I^2C device addresses in the sensor data sheets. Figuring out how to read serial data from the sensor was also difficult.

Radio Problems

For the radio code the most common issue was finding good documentation for all of the things that it needed to do, including the AX.25 data link layer protocol, the timer configuration settings, methods on how to produce sine waves using the square wave that the Arduino produces, and the calculation of the CRC value. Many times the documentation from different sources conflicted with each other, so we had to eventually just choose one, until we could get a chance to test it out. There were many complex pieces in what

seemed to be a simple task we wanted it to do, and each of these bits required careful research and deeper understanding so that we could fit it into the larger whole.

General Payload Problems

One major issue we are still facing is taking the code we wrote and being able to port it to Arduino. In some cases, the code which works fine in a regular C IDE does not function properly with Arduino hardware because of memory limitations, or built in functions. For example, sprintf() is an available function on the Arduino, but it doesn't work with a float format specifier requiring us to use dtostrf() which is not a common C function and produces errors on our C IDE.

Front End Problems

Understanding how to use the sockets.h library was challenging. It was particularly difficult to figure out how to apply File I/O to a filestream that both expects input and produces output, and understand the login protocols associated with that particular socket connection. Producing colors on our front end was also difficult as it seems that the library we used could only produce these colors in Windows, and 2/3rds of our group uses Macs. Creating a map that accurately displays the location is a future challenge for us.

Discussion

Our project met most of the specs that we defined as separate pieces, however it still requires a lot of work to compile everything into a format that works well on the Arduino and works well together. For example, we still need to test the modulation and transmission of the data, however this may require close assistance from an amateur radio expert with access to equipment that can read and decode packets at close range, to eliminate the possibility that our beacon isn't working simply due to power/distance factors. We also need to write the actual code to display our location on the map. Currently it is hardcoded in to produce an asterisk at our approximate location, but producing map which shows the Pacific Northwest in greater detail and using our latitude and longitude calculates the approximate position of the beacon.

Overall, though this project was not fully completed by our deadline, we will not abandon it. We will fly this beacon! Certain members have made plans to meet with radio experts to troubleshoot the transmission, and to get the code fully formatted and functional on the Mega. Future goals are to test the packet transmission, develop a PCB for the radio, further develop the front end, and better document our process for helping out those who might be interested in doing something similar in the future.

Contributions

AJ's Contributions

- Front End
- Sensors
- Trello Meeting Manager

Chuck's Contributions

- Radio
- Packet Encoding
- Socket Connection
- Materials/BoM

Ebtehal's Contributions

- GPS
- Group Photography
- Gannt Chart

References

Ada, Lady. "I2C Addresses!" *Power Usage | Adafruit Motor Shield | Adafruit Learning System*, 29 July 2017, learn.adafruit.com/i2c-addresses/overview.

Admin. "How to Modify the PWM Frequency on the Arduino-part1(Fast PWM and Timer 0)." *Eprojectszone*, 7 Aug. 2016, www.eprojectszone.com/how-to-modify-the-pwm-frequency-on-the-arduino-part1/.

"Arduino - MultiSerialMega." *Arduino Reference*, 2018, www.arduino.cc/en/Tutorial/MultiSerialMega.

"Arduino - SoftwareSerial." *Arduino Reference*, 2018, www.arduino.cc/en/Reference/softwareSerial.

Ayoma. "GY-GPS6MV2 – NEO6MV2 (Neo 6M) GPS Module with Arduino / USB TTL." *Website and Blog of Ayoma Wijethunga*, 28 Nov. 2015, www.ayomaonline.com/iot/gy-gps6mv2-neo6mv2-neo-6m-gps-module-with-arduino-usb-ttl/.

"Computation of Cyclic Redundancy Checks." *Wikipedia*, Wikimedia Foundation, 5 Apr. 2018, en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks.

Hansen, John. *PIC-Et Radio: How to Send AX.25 UI Frames Using Inexpensive PIC Microprocessors*. pp. 29–37, *PIC-Et Radio: How to Send AX.25 UI Frames Using Inexpensive PIC Microprocessors*, www.tapr.org/pdf/DCC1998-PICet-W2FS.pdf.

Industries, Adafruit. "MCP9808 High Accuracy I2C Temperature Sensor Breakout Board." *Adafruit Industries Blog RSS*, www.adafruit.com/product/1782.

Industries, Adafruit. "MPL3115A2 - I2C Barometric Pressure/Altitude/Temperature Sensor." *Adafruit Industries Blog RSS*, www.adafruit.com/product/1893.

Ingalls, Robert. "Sockets Tutorial." *Deadlock*, www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html.

"PWM Sine Wave Generation." *https://Web.csulb.edu/~Hill/ee470/Lab 2d - Sine_Wave_Generator.Pdf*.

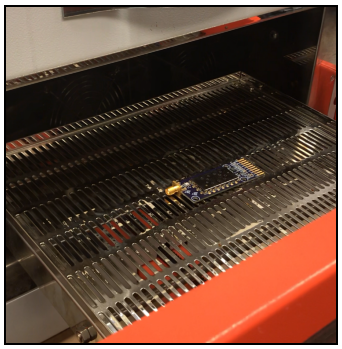
Tcort. "Tcort/va2epr-Tnc: Terminal Node Controller and APRS Tracker Project." *GitHub*, 2018, github.com/tcort/va2epr-tnc.

Appendix

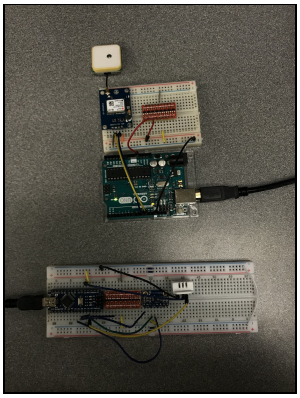
Photos

Soldering Headers on Sensors

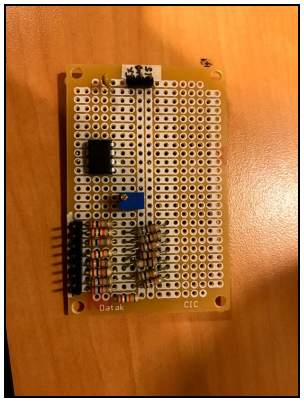
Getting the Radio Board out of the Reflow Oven



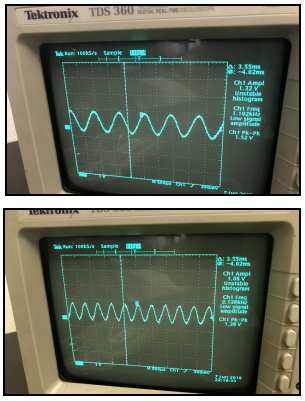
Complete Radio Board (not used)



GPS and Sensor Testing

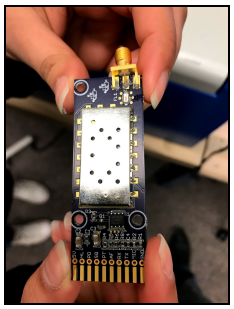


R-2R Ladder Created but not used (didn't work right)

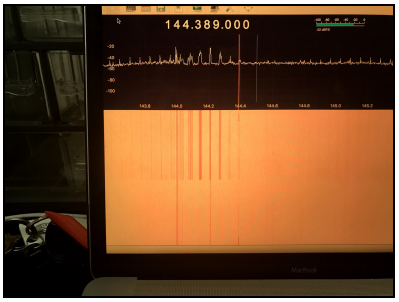


Generation of Sine Waves! (Eureka!)

Hard at work!



Adventures in Solder Paste (module not used)



Testing out the transmitter with our SDR