Table of Contents

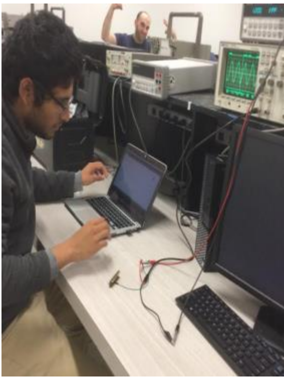
I. Abstract 3
II. Introduction 3
III. Component Selecting 3
IV. PCB Design and Assembly 5
V. Testing 7
VI. On Board Signal Processing 10
VII. Conclusions 13
VIII. References 13
IX. Acknowledgements 13

EEC 134B – High Frequency System Design

Radar Development Process

Team: DiodeHard3 (Final Report)

Vinay Vidyasagar | Mohammad Biswas | Tim Nordqvist | Yharo Torres



I. ABSTRACT

A 2.4 GHz FMCW Radar was built in order to measure distances of up to 50 meters on a single target. The radar was separated into a baseband section as well as an RF section and used digital signal processing on an external computer to showcase the results. The knowledge gained from previous quarter experience and careful consideration of components allowed a reading of up to 43 meters with a deviation of 3-4 meters in results.

II. INTRODUCTION

Frequency modulated continuous wave radar transmits continuous electromagnetic wave which is different from pulsed or any other radar. Generally speaking, the frequency of this type of radar changes over time and sweeps across a set bandwidth. The difference in frequency between the transmitted and received signal is determined by mixing the two signals, producing a new signal which can be measured to determine the distance and velocity. In other words, FMCW radar is able to measure the distance and speed of any reflective objects unlike pulsed radar system in RF and Microwave technology. The journey of making/ engineering this fascinating radar started in the beginning of fall quarter. During these two quarters long journey we have learned various techniques to solve some of the hardest and complex RF & analog circuits. This particular project just took us one step further in becoming efficient electrical engineers.

III. COMPONENT SELECTING

Antenna

The antenna chosen for both the transmitting and receiving portion of the design was a Yagi antenna centered around 2.4 GHz with an average gain of 10 dB. These antennas helped provide a lightweight set of components that feature high gain as well as high directionality. Soldering of these antennas were to be done by connecting them to SMA connectors in order to connect the antennas onto the RF portion of the overall design. Once the antennas were connected through an RF friendly cable, they were to be positioned towards any potential target and held in place by an external structure such as cardboard or styro-foam.

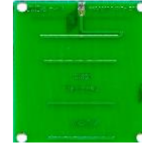


Fig.1- Element Yagi Antenna

Baseband Components

- The baseband components chosen for the initial and final stage of the radar needed to provide the radar with a means for modulating the signal, and then finally, a means of safely feeding the signal to the DSP stage in order to extract the results. A Teensy 3.2 in combination with an external DAC, the MCP- 4921, was used in order to generate a ramping voltage wave for controlling a VCO on the RF portion of the design. An output of 2 to 5V was needed from the DAC. The output of the external DAC was to be fed into a non-inverting amplifier design using a TL-972 op amp with a gain of two in order to meet the VCO tuning range necessary for an output of 2349.5 – 2545.9 MHz.
- The quarter one low pass filter was modified to include the original design for the TL-972 op amp gain stage with a new filter, the max7410-EUA+-ND, in order to filter out unwanted signals as well amplify the signal enough to feed into a signal processor such as a computer or Teensy 3.6 MCU.
- A power leveling system was created in order to power up the baseband and RF boards. From a power source, 10V would need to be drawn and fed into the powerline of the noninverting amplifier placed after the external DAC. From the same 10V line, a voltage divider network would need to yield an output of 8V to be fed into an LM317 voltage regulator. The regulator would output a clean 5V of power to be used as a general powerline for the system. An LT1009 diode would next be needed to

recreate the quarter 1 2.5V reference voltage design using the 5V power rail as VCC.

Component	Part Number	Quantity
Reference Voltage	595-LT1009ILP	3
LPF	MAX7410EUA+-ND	3
Voltage Regulator	595-LM317LCLP	3
DAC	MCP4921-E/P-ND	3
Op amp	595-TL972ID	
Resistors		
220 ohm	(311-220GRCT-ND)	10
2K Pot	(987-1065-ND)	10
3.6K ohm	(311-3.60KHRCT-ND)	10
10K ohm	(311-10KGRCT-ND)	10
20K pot	(3362P-203LF-ND)	5
Capacitors		
0.1uF	(478-1129-1-ND)	10
1uF	(587-1251-1-ND)	10
20pF	(311-1424-1-ND)	5

Fig 2-Baseband Component Selection

RF Components

- The BFCN-2435+ bandpass filter is centered around 2.4GHz with a bandwidth of 2340 to 2530 GHz. This filter is placed after the Rx to filter out unwanted signals to and from both directions.
- The ROS-2490+ VCO was chosen in order to work with the baseband's DAC output used as a VTune of 4-10V for a sweep of 2349.5- 2545.9 GHz.
- The SP-2U1+ splitter was used for its low insertion loss of around 0.5dBm at 2.4 GHz.
- A MCA1-42LH+ mixer was used for its easy to achieve LO level of 10 as well as the low conversion loss of 6dB. An operating frequency of 2300-2500 GHz showed great compatibility with our radar.
- Three 2.5 GHz centered LNAs with large bandwidths were used in the RF block of the radar. Two TAMP-272LN+ LNAs were used to boost the received signal going into the RF port in the mixer. One TAMP-272LN+ LNA was used after the VCO output in order to boost the VCO signal going into the splitter as the inputs of the mixer LO port and Tx antenna.
- One 5dB attenuator, the 1284-1848-2-ND from DigiKey, was placed between the splitter and an LNA in order to precisely match the power requirements of the LO port from the mixer.

Component	Parr number	Quantity
VCO	ROS-2490+	3
LNA	TAMP-272LN+	3
Power Splitter	sp-2u1+	3
Mixer	MCA1-42LH+	3
Bandpass Filter	BFCN-2435+	3

Fig 3- RF Component Selection

The components were put into ADIsimRF in order to calculate power, noise, and to make sure nothing would saturate. The following diagrams show the receiving side and the transmitting side respectively.

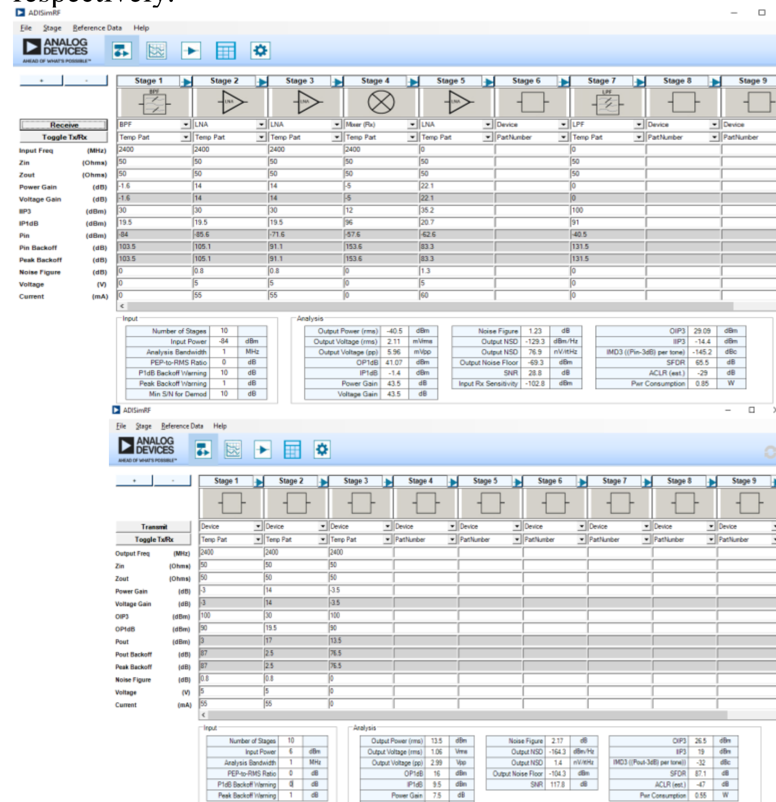


Fig 4 - ADIsimRF

ADIsimRF(Fig 4) was also vital in calculating how much power we were actually outputting in order to check if our calculations were correct so that we could have enough power to detect the target.

The resulting system diagram was made into a block diagram (Fig 5) showing the different

components and how they fit into the system and also the power level at each step.

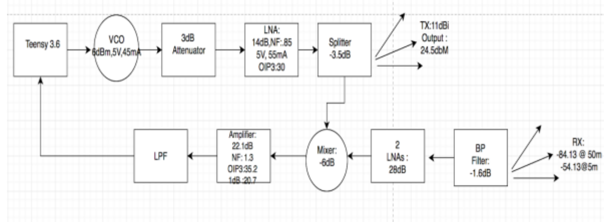


Fig.5 - Block Diagram

IV. PCB DESIGN AND ASSEMBLY

Footprints for the components selected were created and the schematics for the PCBs were assembled. The datasheets were used to figure out the values and locations of the passive components. Although we originally planned on stacking the PCBs, it seemed like an unnecessary complexity. The baseband and RF PCB schematics are shown below in Fig 6 and Fig 7.

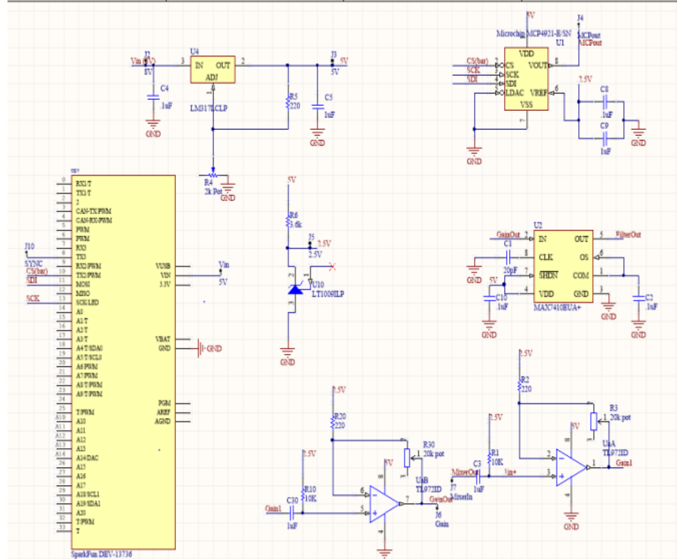


Fig.6 - Baseband PCB schematic

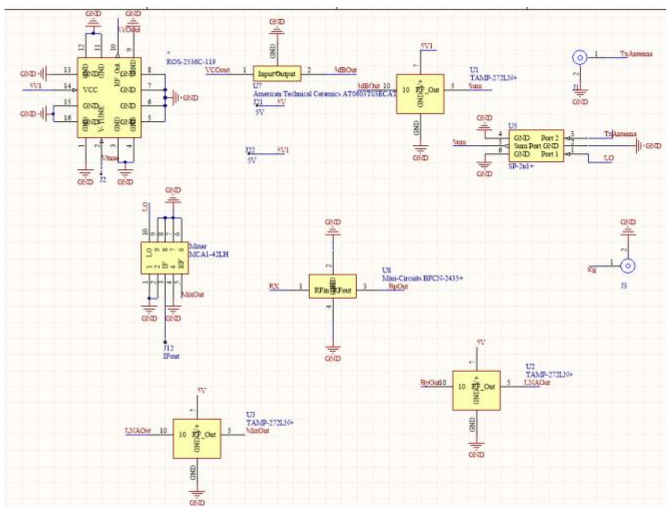


Fig.7 - RF PCB Schematic

One mistake that can be seen from the schematics is the lack of ground points and an output pin for the baseband PCB. These were luckily, relatively easy fixes to make.

The PCB design layouts were created to be as compact as possible and can be seen below in Fig 8 and Fig 9.

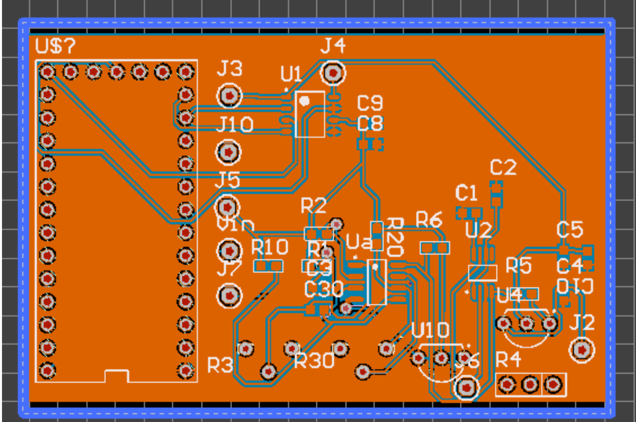


Fig.8 - Baseband PCB Layout

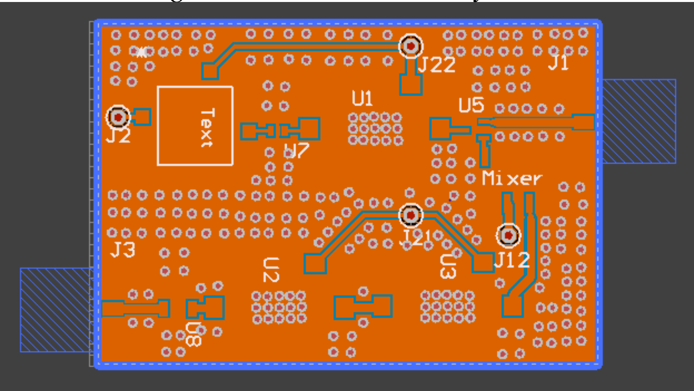
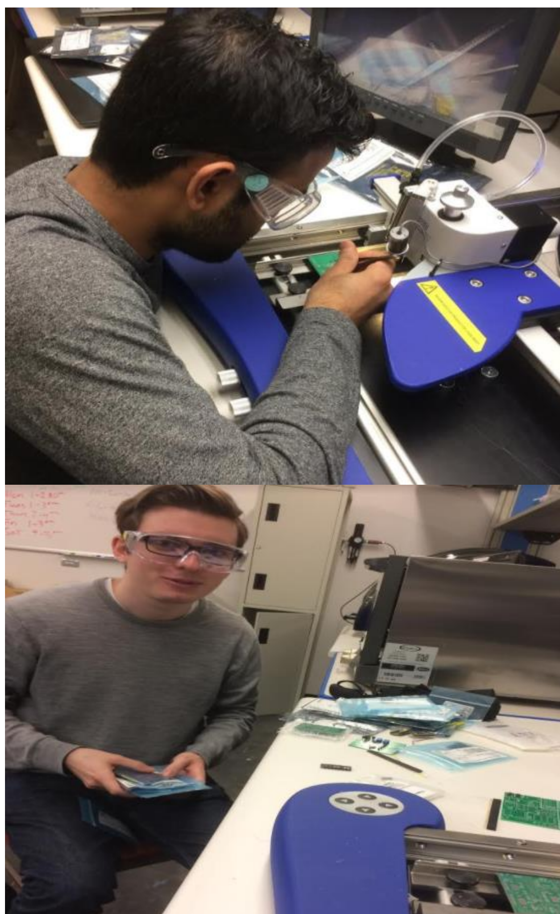


Fig.9 - RF PCB Layout

The baseband PCB used two variable amplifiers so that we could have maximum control of the gain, but it ended up complicating the gain adjustment. We used an IC for the low pass filter in order to have minimum amount of passive components for easier assembly. The RF PCB ended up having no passive components at all which made assembly much more straightforward. The RF PCB ended up being very compact which was because we utilized very short traces and also kept them as straight as possible.

PCB Assembly Process



After receiving the components we rushed ourselves down to the electronics lab where most of the assembly process of our two PCBs took place. First off, we solder the baseband pcb which had more components. Since it was just a baseband we didn't have to think about signal distortions. During the design of the baseband PCB layout we made sure to put as many test points as possible to test the PCBs part by part. However, we noticed there was not a ground test pin which was a design error. To deal with that we hand soldered an external test pin to a ground pin on one of the ICs which later on tested and performed as expected. During the soldering process, we carefully used the pick and place machine in the electronics lab to perform the soldering job more efficiently. After carefully putting the components on their assigned footprints we then placed the board into the oven to complete the soldering process for the baseband printed circuit board.

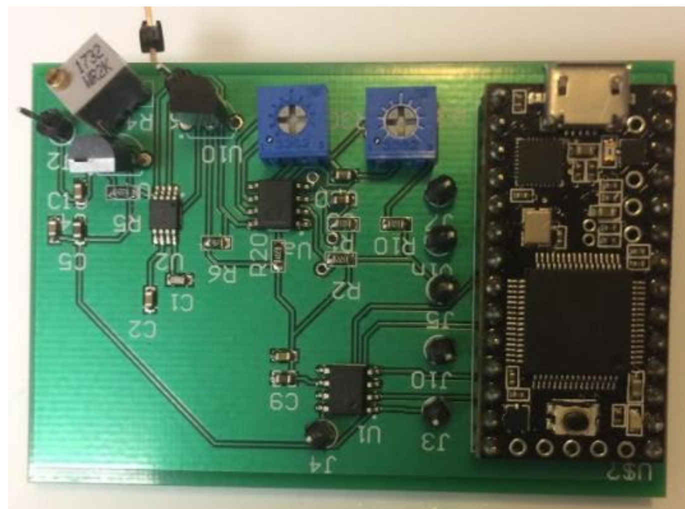


Fig .10 - Assembled Baseband PCB

After finishing the assembly of the baseband PCB we moved on to the RF board. This time we had to make sure the soldering job was perfect knowing the fact that this circuit was designed to work on high frequency signals. Any agitation may interfere with the high frequency signal. All the traces had to be 50 Ω transmission lines to deal with the miss match. Unlike the first board, we used the hot air gun to fulfill the soldering job because the components on this board are mostly SMD packaged and are fairly big. So we focused on one component at a time, by putting solder paste on the pads then carefully place the component on the assigned pad. After carefully placing the component, we then used the hot air to to blow air directly above the component. After a few minutes the paste started to boil which means it working as expected. As soon as the solder paste turns metallic

we stopped the hot air gun and let it cool for a few minutes until it was done. After we finished soldering one, we moved onto the next one and the process continued.

Fig 11- Assembled RF PCB

After fulfilling the task, we put those PCBs up a microscope to check the continuity. It is very essential to check the continuity of the circuit before testing it performance. We used two different methods to check the continuity, one was just to check under a microscope and the second method was done using the digital multi-meter. Both worked fairly well and all the visible minor errors were taken care of afterwards.

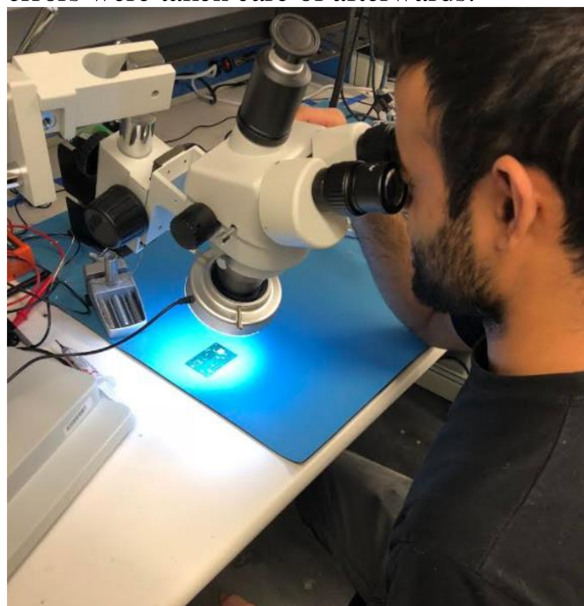


Fig 12 – Testing the PCB.

Performance of assembled PCBs

The RF PCB ended up not working, the VCO was tested with the spectrum analyzer and was functioning properly, the gain from the LNAs were all about 14 dBm as expected, but we were not getting an IF signal. The solder job looked good so we are unsure of what went wrong, but decided on using the RF blocks from quarter one since we did not have spare parts to make a second RF PCB. It seems likely that are mixer was not working, since each of the other components were working as they should have been working.

The baseband PCB worked as expected, but we had an unnecessary amount of gain. It took some time to fine-tune the gain in order to get a clear signal at 50 meters and less. The five volts from the regulator was used to power all the RF blocks. A VTune voltage of .5 to 4.5 Volts was used in order to meet the LNA frequency requirements.

V. TESTING

Before going out on the field to perform range testing we made sure to check the continuity of the two boards. First we used a microscope in the lab to investigate solid connections between the pads and pins of all the electrical elements. Once the solid connections were investigated, we used a multimeter to check again. Since some of the pins were so small we perform the continuity check using needle tipped probes to test. Also during a continuity test of our RF board we learned that it wasn't very accurate because all the RF traces were 50 Ω terminations. It was too low for the multimeter to read. The traces had to be at least 100 Ω for the multimeter to read.

Although, it was quite easy to test the baseband printed circuit board part by part; however, RF board was little difficult to deal with. The amplifiers on the RF board each had a gain of about 15dB. When we measured the power at the input of 5dB attenuator we were getting a good reading however, at the output of the 5dB attenuator the power level was attenuated by almost 15dB which was quite irregular.

Field testing

As mentioned early in the report, our RF PCB didn't work as the output didn't have enough power for the signal to go through. The IF output also appeared to not be working Thus, we replaced that part of the design with regular conventional RF blocks which were provided to us in quarter I (Fig 13). It is worth mentioning that we used the baseband PCB which work well for the intended purpose. We first tried to test the system in the hallway at Kemper Hall. which seemed to interfere with other reflected objects such as walls. We did make a plan on going out to an open field right after, however during the bad weather that didn't happen until the week of 10.

Fig .13A - Test setup with coffee cans

Fig 13B – Testing Radar system in the lab

Fig .13C – Testing in the hallways

Fig. 13D – Test setup side angle

We saw that the results of the hallway testing resulted in amplitude differences depending on where we had our metal plate which indicated that our radar was working. After bringing the radar outdoors we produced some promising results, the picture below is from out last test which happened on Saturday (3/17/18).

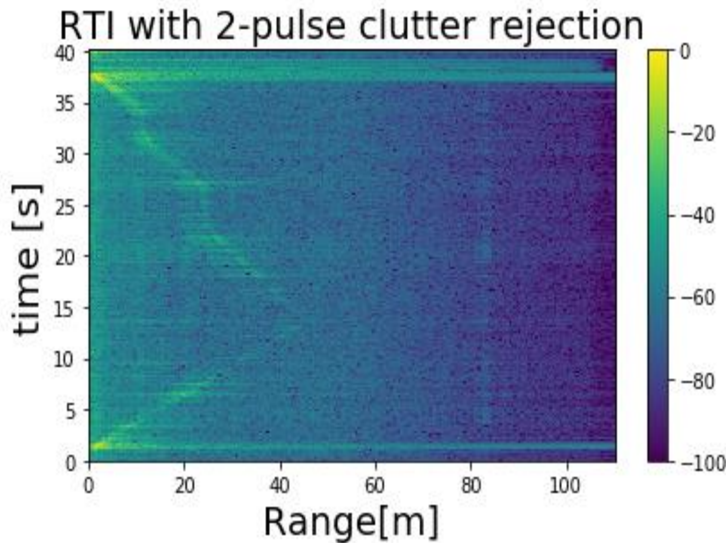
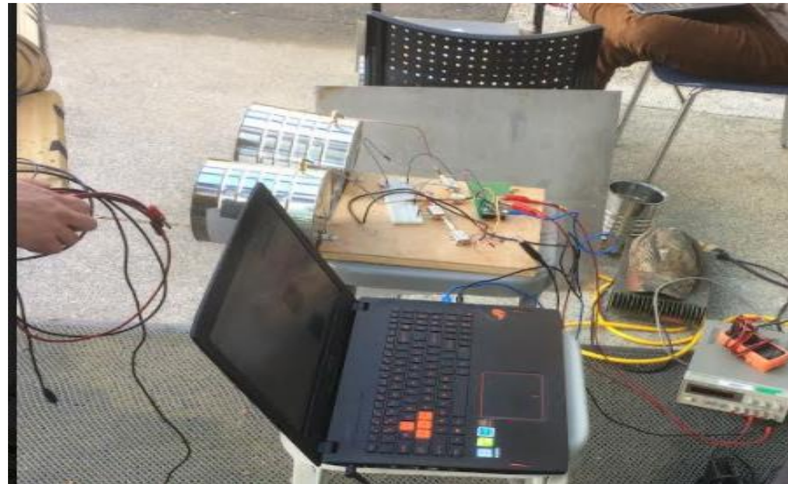


Fig.14 - Testing outside Kemper

As seen in the picture above in Fig 14, the metal plate was moved away from the radar, then stopped, then moving and stopped again, finally returning from about 45 meters back to the radar. These results appeared very promising and we noted down the gain we used on the baseband PCB. The first pot was at 6.6k and the second pot was at 3.9k which gives a gain of about 50dB. These settings were used for our final test, the setup is shown below:



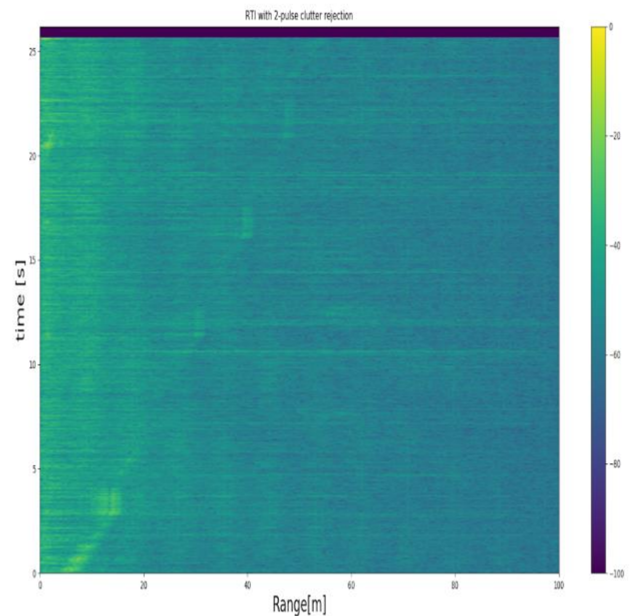


Fig.15 – Final results

Our results are shown below

Actual	Guesses
42.672	48 meters
35.3568	39 meters
26.8224	31 meters
19.2024	21 meters
11.8872	15 meters

It appeared our data was shifted by about 4 meters, which could be due to our teensy going from 0 to 5 volts instead of .5 to 4.5 volts since we had to reset the teensy during the testing. Because of this we did not know our period and frequency range exactly which is very important for our python script which did the signal processing.

VI. ON BOARD SIGNAL PROCESSING

The task of Signal processing involved the accumulation of data and processing them through special functions available through an embedded platforms library. For our purpose we chose to implement the task using a Teensy 3.6 microcontroller in addition to the Teensy 3.2, both of which are Arduino based MCUs. For the case of the embedded signal processing we had put forth a flowchart and worked on implementing parts of the flowchart piece by piece so that we can not only gain a modular system understanding but also process refinement opportunities. For the signal processing aspect, we had decided to allow the Teensy 3.2 to transmit the signal and the 3.6 to processing the incoming IF signal. This meant that through a two MCU combination we need not worry about a difficult scenario of both transmitting and receiving on one MCU. Logic in the form of Transmit/Process switches were developed to signal when the 3.6 would be able to process the required data.

The task was split into two major areas, sampling the signal and processing the sampled signal. For the case of sampling the signal we used the Audio library to aid us in this task. Functions that are native to the library allowed us to receive signal from the ADCs on board the Teensy 3.6. Further, we were able to control the sampling rate which we set to a high frequency mode, 44.1kHz, so that our resolution was not impacted. Attached below(Fig 16A) is a code snippet from our sampling section.

```
ADC *adc = new ADC();// Creating ADC object
File sync_file;//
File data_file;
File readFile;// Two file variables
const int chipSelect = BUILTIN_SDCARD;

void setup() {

  Serial.begin(9600);// Serial baud rate set to 9600
  // put your setup code here, to run once:
  int Pin= A2;// Pin where data is coming in
  int SyncPin = 27;// Place holder for the sync pin
  int start_pin = 24;// This is the pin that we're using for controlling when we can sample the data.


 //******************************ADC properties are being set here *************************
  adc->setConversionSpeed(ADC_CONVERSION_SPEED::HIGH_SPEED,ADC_0);// High conversion speed setting
  adc->setResolution(12);// Using 12 bit resolution
  adc->setSamplingSpeed(ADC_SAMPLING_SPEED::HIGH_SPEED,ADC_0);// High sampling speed setting


  int transmit_flag;
  pinMode(Pin,INPUT);
  pinMode(SyncPin, INPUT);
  pinMode(start_pin,INPUT_PULLUP);// Using the pullup mode to perform a digital read for our transmit flag setting
  Serial.println("Initializing SD Card");
  // Checking to see if SD card is accessable.
  //The main issue with on board signal processing is the memory part. Using an SD card to store for a Teensy is the only option however it's quite complex to work with
  // As you have to deal with Files and treat those files as matrices.
  if (!SD.begin(chipSelect)){

    Serial.println("initialization failed!");
    return;
  }
  else{
    Serial.println("initialization done.");
```

Fig.16A – Sampling section 1 code

This part of the code created the ADC object and we were able to sample at our desired rate. To store the data we used an SD card as we realized that the amount of data to be dealt with is much more than the 1MB flash memory on board the Teensy 3.6. Using the built in SD card we were able to use much more memory to our advantage to store the necessary data as text files. Attached here is the next half of our code (Fig 16B) that enabled us to read the data from the ADC and then transport it to an SD card for permanent storage.

```
//****************THIS LOOP CHECKS TO SEE IF THE CONDITION NECESSARY FOR SAMPLING IS MET. ONCE WE HAVE TRANSMITTED WE CAN SAMPLE THE SIGNAL.**************
while(transmit_flag != 0){
 transmit_flag = digitalRead(start_pin);
 Serial.println(digitalRead(start_pin));
 delay(200);
}
Serial.println("Ready to Sample");


while (transmit_flag != 1){
   Serial.println("In while");
   bool adc_status =  adc->startContinuous(Pin,ADC_0); // Obtaining status of ADC

   if( adc_status == 1){
     int SyncValue = digitalRead(SyncPin);
     int pinValue = adc->analogReadContinuous(Pin);// ADC Value being read
     sync_file.println(SyncValue);
     data_file.println(pinValue);// The sync and the ADC output are written to the text file



   }
   else{
       continue;
     }
   Serial.print("At the end of while");
   transmit_flag = digitalRead(start_pin);
}

sync_file.close();
data_file.close();// Store the data and sync file in 2 different files
Serial.print("The text files have been completed.");
```

Fig.16B-Sampling Section 2 code

Once the sampling portion of the task was taken care of the next big goal was to process the data stored on the SD card. A python script called "range_wav.py" available on the course's GitHub page was converted to a C based environment so that we could process the data on the Teensy 3.6 MCU.

For this part we had decided to just output the final value of the distance of the target. The procedure in the "range_wav.py" was followed with slight differences; we did not use the " .wav" format for our data as that was a redundant step in our implementation as we used the SD card to store the sampled ADC values of the IF signal. To make significant strides in implementation we had to understand the functionality and flow of the "range_wav.py" script. It was here where we realized where the difficulty of the on-board signal processing comes from.

The size of the matrices created by this script for analysis would occupy more than the amount the Teensy 3.6 could provide. The solution meant that we had to perform every matrix operation such as writing and reading from text files that we treated as matrices, in short, the SD card was used as main memory. Attached below (Fig.17A) represents the original idea of an implementation without the memory constraint problem.

```
for (int i = trnc_samples; i < num_frames; i+=2) {
  //left[0] = data[4 * i];// Obtaining left data individually
  //left[1] = data[4 * i + 1];
  if ( i == 0){
    right[0] = data[4 * i + 2];// obtaining right data individually
    right[1] = data[4 * i + 3];//
  }
  else{
    right[0] = data[4 * (i-1) + 2];// obtaining right data individually
    right[1] = data[4 * (i-1) + 3];//
   }

  s[i] = right[0]/4095;// storing data
  s[i+1] = right[1]/4095;//

}
// let sync signal be in trig. read from the SD card.

double s2[(sizeof(s)/sizeof(s[0])) / N][N];
double s3[(sizeof(s)/sizeof(s[0])) / N][N];
double s4[(sizeof(s)/sizeof(s[0])) / N][N];
double s5[(sizeof(s)/sizeof(s[0])) / N][N];
double s6[(sizeof(s)/sizeof(s[0])) / N][N];

int rows = 0;

for (int j = 10; j < sizeof(trig)/sizeof(trig[0]); j++) {// for loop for checking on rising edge of sync data
  int q = j - 10;
  int sum_trig = 0;
  while (q!=j) {
    sum_trig += trig[q];// calculates the sum of those individual trig values
    q += 1;// Updating loop counter so that it breaks when ten points are added
  }
  int avg_trig = sum_trig / 10;// average of those trig values
  if (trig[j] == 1 && avg_trig == 0) { // tests the mean of ten samples being 0 and the element of the sync data being logical 1
    if ( j + N <= sizeof(trig)) {
      for (int p = 0; p <= N; p++) {
        s2[rows][p] = s[j + p];// storing into s2
      }
      rows += 1;//
    }
  }

}

for (int a = 0; a < rows ; a++) {
  for (int b = 0 ; b < N ; b++) {
      s3[a][b] = s2[a][b];// storing into s3 from s2. Populating the values of s2 into s3.
```

Fig .17A – Original logical array operations

```
for j in range(trnc_smp,numframes):
    # get the left (SYNC) channel
    left = data[4*j:4*j+2]
    # get the right (Data) channel
    right = data[4*j+2:4*j+4]
    #.wav file store the sound level information in signed 16-bit integers stored in little-endian form
    #The "struct" module provides functions to convert such information to python native formats, in th

    if len(left) == 2:
        l = unpack('h', left)[0]
    if len(right) == 2:
            r = unpack('h', right)[0]
        #normalize the value to 1 and store them in a two dimensional array "s"
    trig[j-trnc_smp] = l/32768.0
    s[j-trnc_smp] = r/32768.0

#trigger at the rising edge of the SYNC signal
trig[trig < 0] = 0;
trig[trig > 0] = 1;

#2D array for coherent processing
s2 = np.zeros([int(len(s)/N),N])

rows = 0;
for j in range(10, len(trig)):
    if trig[j] == 1 and np.mean(trig[j-10:j]) == 0:
        if j+N <= len(trig):
            s2[rows,:] = s[j:j+N]
```

Fig.17B – Python script range_wav.py

From a logic point of view, these two code snippets represent the same logic as what the "range_wav.py" performs in these following lines of Fig .17B besides the absence of using ".wav" files in the Arduino implementation.
Upon realizing the memory problem we transformed our current simple logical array operation into a file based data extraction and writing process. This meant that more logic had to be thought of and implemented to get our system functional. Attached here is a representation of the new logic (Fig 18A and B) that was developed for a simple array based operation.

```
bool TRUE;

data = SD.open("data.txt");  // open file locatoin where data is stored ***********
trig = SD.open("sync.txt"); // open file location where sync info is stored ***********
s = SD.open("s1.txt",FILE_WRITE) // First instance of file to be created in SD card ***********

num_frames = data.size()/(sizeof(int)) // Divide by size of int to get number of elements ***********

for (int i = trnc_samples; i < num_frames; i++) {


  // Data is already sorted at end of ADC code since we are recording onto two files at *********
  // the same time, now only need to format it with our 13 bit resolution **********

 buffer = data.parseInt(); // Read into a temporary buffer for computing *****
 s.println(buffer/(2^12)); // Divide by number of bits of resolution minus one, in this case 13 bits goes to 12 bits (0 to 4095)
                           // Print to s1.txt file in lines (columns) *********


}

s.close(); // Close the "s.txt" file to ropen it up later and read from it *******
s = SD.open("s1.txt");


// let sync signal be in trig. read from the SD card.

// Assume all file pointers are created as File objects in setup when adding ADC code **********
// As a reference  file.fseek(position) file.fposition()  *********

//int s2[(sizeof(s)/sizeof(s[0])) / N][N]; ******** obsolete
s2 = SD.open("S2.txt",FILE_WRITE):
s2temp = SD.open("s2temp.txt",FILE_WRITE);

//int s3[(sizeof(s)/sizeof(s[0])) / N][N]; ******** obsolete
s3 = SD.open("S3.txt",FILE_WRITE);

//int s4[(sizeof(s)/sizeof(s[0])) / N][N]; ******* obsolete
S4 = SD.open("S4.txt",FILE_WRITE);


for (int j = 10; j < num_frames; j++) {// for loop for checking on rising edge of sync data
  int q = j - 10;
  int sum_trig = 0;

  trig.seek(2*(j-10)); // Moves index to 10 before the value of the current j while skipping indexes taken up by spaces.**************


  while (q!=j) {
    // sum_trig += trig[q];// calculates the sum of those individual trig values ******** obsolete
    sum_trig += trig.parseInt();
    q += 1;// Updating loop counter so that it breaks when ten points are added
  }

  int avg_trig = sum_trig / 10;// average of those trig values

  trig.seek(2*j); // File point currently at j+2 after being read with parse at index j earlier--> (j)value [space] (j+1)value, Reset file pointer back to j ********
  catchup = 0: // reset the catchup index
```

Fig .18A-Logical conversion using SD card

```
  trig.seek(2*j); // File point currently at j+2 after being read with parse at index j earlier--> (j)value [space] (j+1)value, Reset file pointer back to j ********
  catchup = 0; // reset the catchup index

  while (catchup <=  j){  // This loop allows the s array to be at the same index as the trig array so we can keep track of the pairs of values between them.

    s.parseInt(); // Keeps reading int values and stops after reading a space. THis helps make sure we are at the right "element" of the array.
    catchup += 1;


  }

  if ( (buffer = trig.parseInt()) == 1 && avg_trig == 0 ) { // tests the mean of ten samples being 0 and the element of the sync data being logical 1
                                                          // As parseInt() is called, the file pointer already moves down below as if  **********
                                                         // going through a column array ***********

    //if ( j + N <= sizeof(trig)) { ************ obsolete
    if (j + N <= num_frames){ //*********** the num_frames already has the number of elements that would normally be stored in a column array
      for (int p = 0; p <= N; p++) {
        // s2[rows][p] = s[j + p];// storing into s2   ************* obsolete
        if (p != N){

          s2temp.print(s.parseInt()); // This reads the entire numerical int number for the column element at the specified row until the space character is read, indicating the entire number is read. ********

        }else{

          s2temp.println(s.parseInt()); // This is the creation of a new row, technically. ************
                                        // Due to the way files work, we are printing columns as rows and vice verse. Will correct this in a loop after by switching them ***********
        }
        columns = p; //keeps track of how many columns were created thus far ************
      }

      rows += 1;// keeps track of how many rows were created thus far. ***********

    }
  }
}

// This matrix will switch the rows with the columns to store the matrix correctly inside the file called "s2" ******************
s2temp.close();
s2temp = SD.open("s2temp.txt");

s2temp.seek(0) ;// Reset file pointer to start of the entire file just incase *************

for (COLUMNS = 1; COLUMNS <= columns; COLUMNS ++){

correction = 1;

  for (ROWS = 1; ROWS <= rows; ROWS++){

    while ( (COLUMNS - correction) ){

      s2temp.parseInt(); // Moves the file pointer over by a column as columns are finished recording *********************
      // THe logic here is that as the COLUMN variable moves away from correction factor of 1, previous columns have been recorded onto s2 already so we need to
      // skip these as we again move down the rows to record for s2. ****************
      correction += correction;
    }

    buffer = s2temp.parseInt();
    s2.print(buffer);  // This is the chosen value to record onto the s2 & s3 matrix. Anything after, if not at end of row already, is skipped onto next iteration of row change ******
    s3.print(buffer);

    if (COLUMNS != columns){  //This line will only implement if the last column value wasn't read. **************

    s2temp.readStringUntil('\n'); // Will read an entire row where file pointer is positioned as a string until a newline is read. then places the file pointer in beginning of next row ***********
```

Fig.18B-Logical Conversion continued

The use of file pointers and file operations made the task complex as some of the sections required reading from a column which is not easily done in C , which reads elements of a file one row at a time. The complex file method of implementation had to be performed in order to get a functional system. It would have been possible to avoid this complexity through the use of using less samples and reducing the sampling rate but that would mean that our data would not be accurate and our resolution would suffer. This was a tradeoff that we could not afford and hence we proceeded with an arduous implementation.

A function/procedure (Fig 19) was implemented to aid us with our task of extracting data from a file and replacing it with another data in the same exact location.

```
int readFile(FILE *file, int row, int col) {
    int val = 0;
    int count = 0;
    char line[100];
    //fpos_t position;

    while (row > count)
    {
        fgets(line, 100, file);// File pointer
        count++;
    } // skip to the row
    //printf("Count: %d\n", count);

    // now read char by char
    int num_ws = 0; // number of whitespaces
    char ch = '_';
    while ((ch = fgetc(file)) == '\n' || col > num_ws)// Checks to see if there is a new line character in the file
    {
        if(ch == ' ')
        {
            num_ws++;
        }
    }
    ungetc(ch, file);
    long pos = ftell(file); // save position

    // read
    while ((ch = fgetc(file)) != ' ' && ch != EOF)// reads the file till the end and till no space
    {
        putchar(ch);
    }
    printf("%c\n", val);

    // now write
    fseek(file, pos, 0);
    fputs("99", file);

    return val;
}

int main(int argc, const char * argv[]) {
    FILE *in_file = fopen("file.txt", "r+");
    if(in_file == NULL)
    {
        perror("Error opening file");
        return(-1);
    } // check if file exists

    int row = 2;
    int col = 1;
    readFile(in_file, row, col);
```

Fig.19- Function to write and replace data in file

While testing we were sure of this working for a small data set and text file however we were unsure of how this would work with bigger files generated when the entire radar system is hooked up. Due to a shortage of time along with uncertainty we could not finish our goal of completing the on board signal processing and thus went for a processing approach through the use of a laptop. We understand that signal processing is a crucial part of Radar system we take complete responsibility in not being able to deliver on this part of the project. On the other hand we are fortunate to learn the inner workings of an embedded system and this improved our algorithm thinking and learned embedded system skills.

If time does permit over the course of Spring Quarter we do hope to show the TAs and the Professor a working model with the signal processing performed on an embedded platform (Teensy 3.6).

VII. CONCLUSIONS

When it comes to designing and testing a radar there are many key design procedures one must consider at all times. Creating a design with components that work together while taking into account how each added component impacts the overall performance of the design is the first step in the process. Compacting the design and modifying it to improve its efficiency on paper is the next step. Finally, testing each aspect of the design in order to explore areas where performance expectations might not be where one expects is crucial for ensuring a working system is in order is the last step.

This design course allowed the team to test their design skills in more ways than one and found ways to challenge our approach to the physical real life problems found within engineering outside the classroom. There were many ideas brought up and discarded throughout the quarter, and often times one finds themselves settling on alternatives rather than improving upon initial design choices. Despite not being able to pursue choices like embedded system DSP or compacted/stacked design boards, the radar was able to be completed with what was left at hand. Every obstacle encountered should be looked upon as a chance for a new perspective and ultimately these things are what helps shape our roles in engineering in the future.

VIII. REFERENCES

1. https://github.com/ucdart/UCD-EEC134/tree/master/labs/lab6/code

2. MiniCircuits website

IX. ACKNOWLEDGEMENTS

Team DiodeHard3 would like to acknowledge Professor Leo, Mahmoud Nafe, Hind Reggad and Songjie Bi in their help and guidance throughout the two quarters.