Music Synthesizer Senior Project: Individual Report

Bryan Bellin

Evan Lew

Vikrant Marathe

Jordan Wong

Advisor: Dr. Wayne Pilkington

June 2017

Cal Poly Electrical Engineering

Abstract

The Danalog is a 25 key portable digital music synthesizer that uses multiple synthesis methods and effects to generate sounds. Sound varieties included three synthesis methods including FM, subtractive, and sample-based, with up to eight adjustable parameters, at least four effects, including reverb, chorus, and flange, with five adjustable parameters, and at least two note polyphony, and a five band equalizer. The user would be able to adjust these effects using digital encoders and potentiometers and view the settings on two LCD screens.

The finals project was unable to meet the original design requirements. The FM synthesis method was primarily working in the end product. The synthesizer was built to produce two note polyphony. The LCD screens displayed the information about the synthesis method as the user plays.

I. Introduction

The purpose of this project was to create a portable, inexpensive digital music synthesizer for amateur musicians. The intended customer base consists of young, amateur musicians who don't have a big budget for a more expensive music synthesizer.

The market requirements for this product are as follows:

- The Danalog Synthesizer will be inexpensive at less than $200
- The design will be sleek and lightweight to promote portability
- Up to eight adjustable synthesis parameters
- Up to five adjustable effects parameters
- Five band equalizer

Our intended customer is an amateur musician seeking an inexpensive digital synthesizer to create a wide array of user-defined sounds.

Several other companies have their own digital synthesizers equipped with numerous features. The Danalog's main competitors would be the Yamaha Reface, Korg Minilogue, Roland Boutique, and Arturia MicroBrute. The lowest price of these is the Arturia MicrBrute at $299 - which the Danalog has beat by $100. The Danalog digital synthesizer is also smaller than the other competitor's options.

II. Product Design Engineering Requirements

<u>Functional and Feature Requirements</u>

- The Danalog Synthesizer will produce notes over a 2 octave range via Frequency Modulation Synthesis, with two note polyphony.

- The chassis will be made from lightweight plastic that is easy to carry and hold.
- All components, peripherals, and circuit boards are industry standard and well supported.
- The encoders, potentiometers, and switches will be strategically placed in a manner that follows the logical path of the signal from generation, to equalizing, to modulating.
- The processing will be split among two IC's: The ATmega2560 for peripheral information, and the TMS320C5535 for Digital Signal Processing.

Performance Specifications

- Low latency (<3ms delay) production of notes
- Instant visual feedback (<3ms) on pressed note
- Internal rechargeable battery of 5 hour life
- Able to run on 5V 500mA USB power
- Low noise audio outputs. 90dBc S/N with +4dBu max output

Level 0 Blackbox Diagram

User Input: All of the user inputs are translated into 8-bit signals, handled by the ATMega2560 microcontroller. This includes MIDI protocol, potentiometer positions, encoder rotation direction, switch and button positions (via mux), for a total of 30 bytes. All of them are traced to an Arduino Mega development board on a PCB where the ATMega chip resides.

5V Power Supply: The synthesizer is powered via 5V 500mA USB power or a 5 volt battery. There is a level shifter as well, because the Arduino Mega board runs on 5V while the TMS320C5535 runs on 3.3V.

III. Independent Work

Output Buffer

To meet the specification of having both headphone outputs and balanced outputs the DAC output would have to be buffered to allow the headphone output to not interfere with the balanced outputs. To create a buffer a voltage follower is made. In order to determine specifications the audio demo that came preinstalled with the DSP board was utilized to max out the audio codec. This demo created a playback device on the computer so the DSP board could act as the DAC for all sound being made from the computer. By playing youtube at max volume and turning on persistence on an oscilloscope the open circuit voltage is determined to never surpass ±1.5 Volts. So to cover all possible signal range the FSR will be considered to be plus or minus 1.5 volts.

Since the DSP board and arduino deal with 5 Volt supplies I had the rails to my buffer go from 0-5 Volts. This is admittedly not enough to satisfy Professor Prodanov's general rule of having the rails of an amplifier at least 2 volts beyond the maximum signal voltage but testing later proved the buffer was good enough.

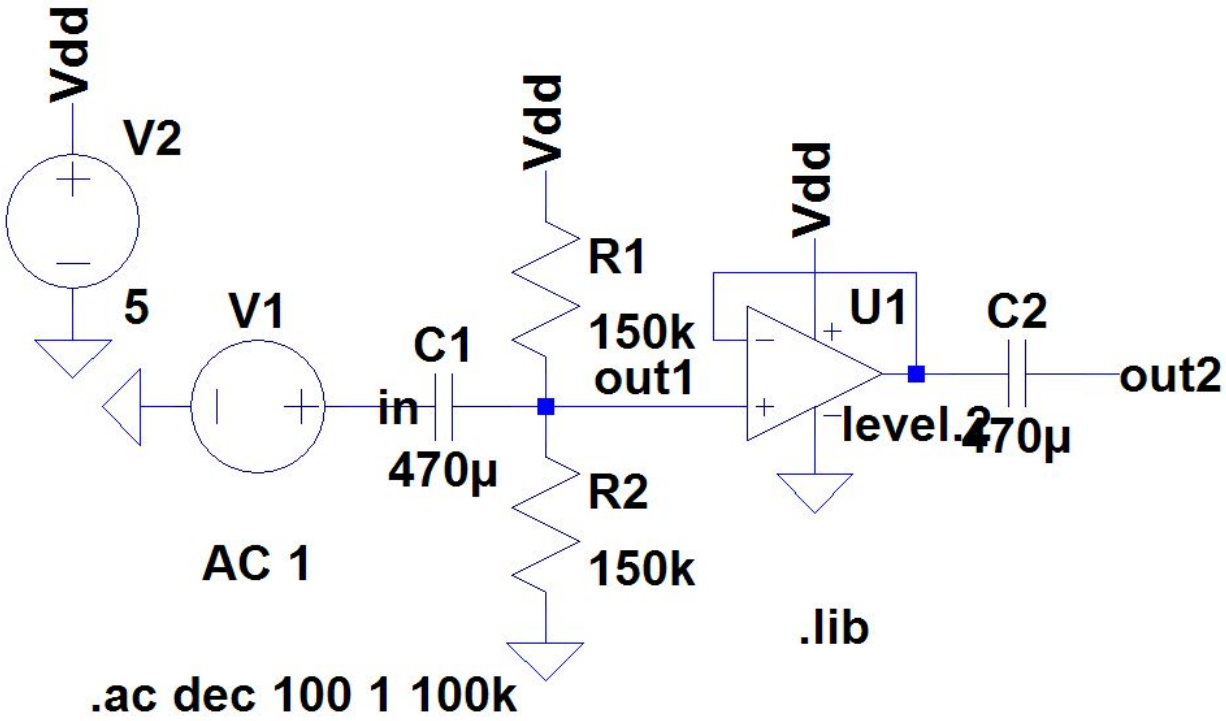


Figure 1. Separated Circuit Test Schematic

The IC ultimately chosen was LM833. It houses two op amps which would be convenient for left and right channels. The subsystem was tested separately with a function generator and it was found that slight peak compression occurred but was only a few millivolts so it was finalized.

FFT Implementation

In order to have an LED display of the audio being outputted and easily equalize any input from audio generation it would be necessary to have a Fourier transform function that performed in real time. The first attempt to perform an FFT within the DSP board was done by reading the manual called spruh87h-trm.pdf. The manual provided example codes using the hwafft_Npts() and hwafft_br() functions. With the exception of different input arrays the code worked as shown below.

==========Hwafft Code==========

```
#define N ( 16 )  //must change funct name and DATA_LEN_X if this is changed

#define ALIGNMENT 2*N //ALIGNS data_br_buf to an address with log2(4*N) zeros in the
least significant bits of the byte address
#include "stdio.h"
#include "hwafft.h"
#include "ezdsp5535.h"
/*
 * main.c
 */
int main(void) {

//#include <std.h>

//#include <log.h>

//#include "hellocfg.h"

#pragma DATA_SECTION(data_buf, "data_buf");
       //static allocation to section: "data_buf : > DARAM" in Linker CMD File
//Int32 data_buf[N] =
{0,2147483647,0,2147483647,0,2147483647,0,2147483647,0,2147483647,0,2147483647,0,21474
83647,0,2147483647};
Int32 data_buf[N] =
{0,821755904,1518469120,1983971328,2147418112,1983971328,1518469120,821755904,0,296923
9552,3665952768,4131454976,4294901760,4131454976,3665952768,2969239552};
Int32 *data = data_buf;

#pragma DATA_SECTION(data_br_buf, "data_br_buf");
                                          //Allocation to Section: "data_br_buf : >
DARAM" in Linker CMD File
#pragma DATA_ALIGN (data_br_buf, ALIGNMENT);
Int32 data_br_buf[N];
Int32 *data_br = data_br_buf;

#pragma DATA_SECTION(scratch_buf, "scratch_buf");
       //static allocation to section: "scratch_buf : > DARAM" in Linker CMD File
Int32 scratch_buf[N];
Int32 *scratch = scratch_buf;
//Int32 *scratch;

Int32 *result;
```

```
//Int32 *data_br:
Uint16 fft_flag;
Uint16 scale_flag;

Uint16 out_sel;

fft_flag = FFT_FLAG;
scale_flag = SCALE_FLAG;

//LOG_printf(&trace, "hello world!");

/* Bit-Reverse 1024-point data, Store into data_br, data_br aligned to 12-least
significant binary zeros*/

hwafft_br(data, data_br, DATA_LEN_16); /* bit-reverse input data, Destination buffer
aligned */

data = data_br;

/* Compute 1024-point FFT, scaling enabled. */

out_sel = hwafft_16pts(data, scratch, fft_flag, scale_flag);

if (out_sel == OUT_SEL_DATA) {

result = data;

}else {

result = scratch;

}
       return 0;
}
```

========== End Hwafft Code ==========

This did not work. The project would compile but output would always be zero or unchanged from the initial values.

Going back to square one. A DSP library for the C5535 DSP chip was found online and installed into a project. An example for this FFT was already provided and worked perfectly. for its test data.

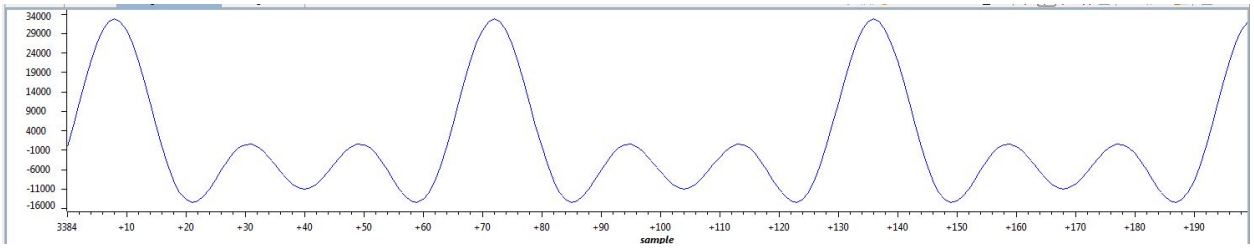


Photo of test data input time domain

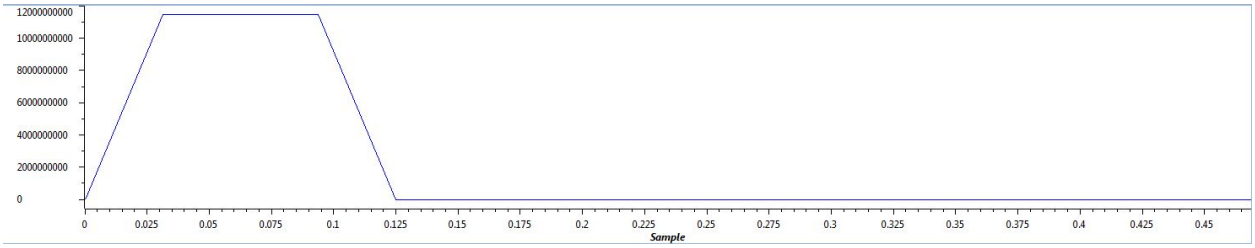


Photo of test data FFT by debugger

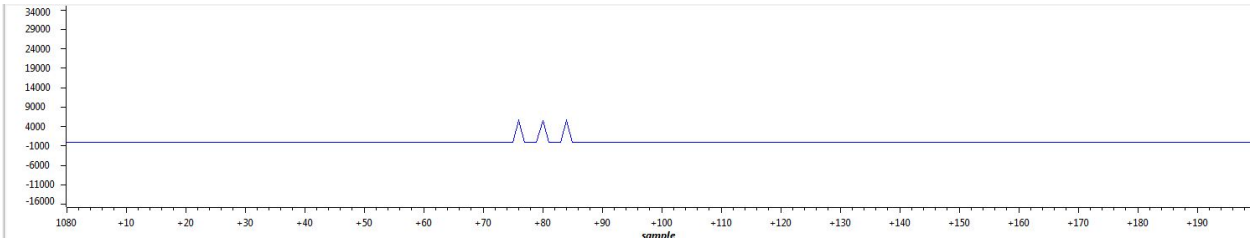


Photo of test data FFT by function

==========cfft example code==========

```
//***************************************************************************
//  Filename: cfft_t.c
//  Version:  1.0
//  Description: test for cfft routine
//***************************************************************************

#include <stdlib.h>
#include <math.h>
#include <tms320.h>
#include <stdio.h>
#include <dsplib.h>

//#include "t1_SCALE.h" //8
//#include "t2_SCALE.h" //16
//#include "t3_SCALE.h" //32
//#include "t4_SCALE.h" //64
//#include "t5_SCALE.h" //128
//#include "t6_SCALE.h" //256
//#include "t7_SCALE.h" //512
//#include "t8_SCALE.h" //1024
```

```
//#include "t2_NOSCALE.h"
//#include "t3_NOSCALE.h"
//#include "t4_NOSCALE.h"
//#include "t5_NOSCALE.h"
//#include "t6_NOSCALE.h"
//#include "t7_NOSCALE.h"
#include "t8_NOSCALE.h"
#include "Dsplib.h"
#include "Dsplib_c.h"


#ifndef SCALING
#define SCALING 0
#endif

short test(DATA *r, DATA *rtest, short n, DATA maxerror);

short eflag = PASS;

void main()
{
    // compute
#if SCALING
    cfft(x, NX, SCALE);
#else
    cfft(x, NX, NOSCALE);
#endif

    cbrev(x,x,NX);

    // test
    eflag = test(x, rtest, NX, MAXERROR);

    if(eflag != PASS)
    {
        exit(-1);
    }

    return;
}
```

==========End cfft code==========

However changing the input to a normal sine wave made output a series of zeros with the exception of one heavily negative imaginary value.

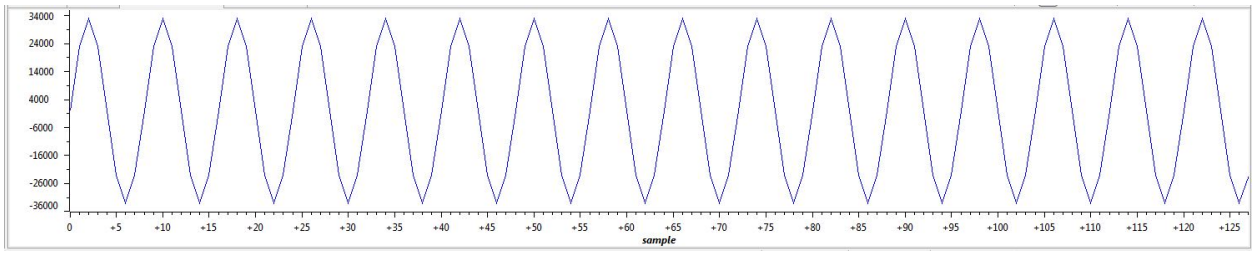


Photo of test sine wave

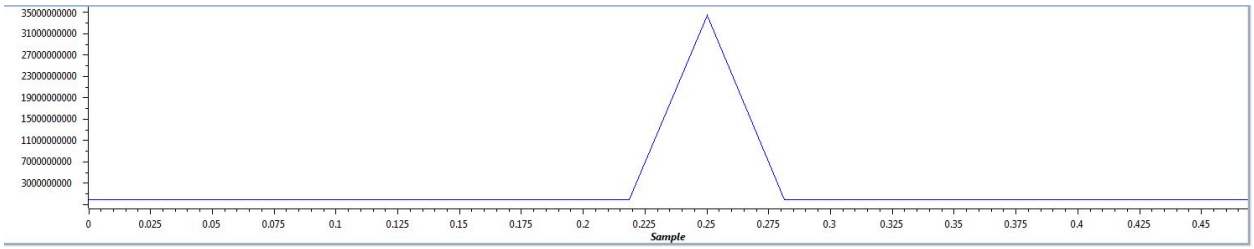


Photo of test sine wave FFT by debugger

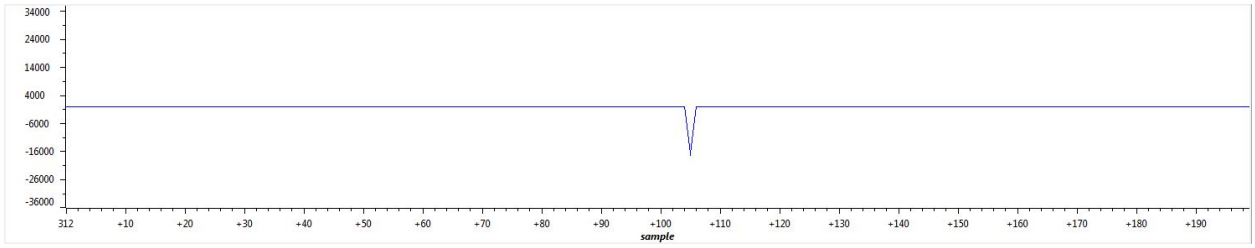


Photo of test sine wave FFT by function

A different approach was found in an example project posted on the Texas Instruments Forums. The fft function used there was called CFFT_SCALE.

==========CFFT_SCALE CODE==========

```
#include <stdio.h> //keeping for printf()
#include <math.h> //not touching till sin() is gone
//#include "TMS320.H"
#include "Dsplib.h" //critical for data declaration
#include "usbstk5505.h"  // Do not touch
#include "Application_1_Modified_Registers.h" //no touch
#include "Audio_To_MIDI_Using_DMA_and_CFFT.h" //no touch
//#include "hwafft.h"
//#include "Output_MIDI.h"

Int16 OverlapInL[WND_LEN];
Int16 OverlapInR[WND_LEN];
Int16 OverlapOutL[OVERLAP_LENGTH];
Int16 OverlapOutR[OVERLAP_LENGTH];
```

```
Int8 BBITOverlapInL[WND_LEN];
Int8 BBITOverlapInR[WND_LEN];
Int8 BBITOverlapOutL[OVERLAP_LENGTH];
Int8 BBITOverlapOutR[OVERLAP_LENGTH];
Int8 BBITOutput_to_LEDS[8];

/* --- buffers required for processing  ----*/
#pragma DATA_SECTION(BufferL,"BufL");
Int8 BufferL[FFT_LENGTH];
#pragma DATA_SECTION(BufferR,"BufR");
Int8 BufferR[FFT_LENGTH];
#pragma DATA_SECTION(realL, "rfftL");
Int8 realL[FFT_LENGTH];
#pragma DATA_SECTION(realR, "rfftR");
Int8 realR[FFT_LENGTH];
#pragma DATA_SECTION(imagL, "ifftL");
Int8 imagL[FFT_LENGTH];
#pragma DATA_SECTION(imagR, "ifftR");
Int8 imagR[FFT_LENGTH];
#pragma DATA_SECTION(PSD_Result, "PSD");
Int8 PSD_Result[FFT_LENGTH];
#pragma DATA_SECTION(PSD_Result_sqrt, "PSD_sqrt");
Int8 PSD_Result_sqrt[FFT_LENGTH];
/* ----------------------------------------*/
/* --- Special buffers required for CFFT ---*/
#pragma DATA_SECTION(complex_data,"cmplxBuf");
DATA complex_data[2*FFT_LENGTH];
/* ----------------------------------------*/

int Audio_To_MIDI_Using_DMA_and_CFFT(void) {

	int i = 0;
	int j = 0;
	//int f = 0;
	//DATA Peak_Magnitude_Value = 0;
	//DATA Peak_Magnitude_Index = 0;
	//int MIDI[256] = {0};

	printf("Initializing Buffers\n");
	/* Initialize buffers */
	for (i = 0; i < WND_LEN; i++) {
		OverlapInL[i] = 0;
		OverlapInR[i] = 0;
	}

	for (i = 0; i < OVERLAP_LENGTH; i++) {
		OverlapOutL[i] = 0;
		OverlapOutR[i] = 0;
	}
```

```
       //printf("Entering infinite loop\n");
       /* Begin infinite loop */
       //while (1)
       //{
        /* Get new input audio block */
              printf("Overriding Mic input with own sine wave\n");

              if (PingPongFlagInL && PingPongFlagInR)  // Last Transfer complete was
Pong - Filling Ping
              {
                     for (i = 0; i < HOP_SIZE; i++) {
                           /* Copy previous NEW data to current OLD data */
                           OverlapInL[i] = OverlapInL[i + HOP_SIZE];
                           OverlapInR[i] = OverlapInR[i + HOP_SIZE];

                           /* Update NEW data with current audio in */
                           //OverlapInL[i + HOP_SIZE] = DMA_InpL[i + AUDIO_IO_SIZE];
// CPU Copies Second Half of index values ("Pong"), while DMA fills First Half
("Ping")
                           //OverlapInR[i + HOP_SIZE] = DMA_InpR[i + AUDIO_IO_SIZE];
                           OverlapInL[i + HOP_SIZE] = 32767*sin(2*22/7*i);
                           OverlapInL[i + HOP_SIZE] >>= 5; //shift value 8 bits over
for data transfer from 16 bit to 8 bit
                           OverlapInR[i + HOP_SIZE] = 32767*sin(2*22/7*i);
                           OverlapInR[i + HOP_SIZE] >>= 5; //shift value 8 bits over
for data transfer from 16 bit to 8 bit
                    }
              }
              else                                                    // Last
Transfer complete was Ping - Filling Pong
              {
                     for (i = 0; i < HOP_SIZE; i++) {
                           /* Copy previous NEW data to current OLD data */
                           OverlapInL[i] = OverlapInL[i + HOP_SIZE];
                           OverlapInR[i] = OverlapInR[i + HOP_SIZE];

                           /* Update NEW data with current audio in */
//                         OverlapInL[i + HOP_SIZE] = DMA_InpL[i];
//                         OverlapInR[i + HOP_SIZE] = DMA_InpR[i];
                           OverlapInL[i + HOP_SIZE] = 32767*sin(2*22/7*i);
                           OverlapInL[i + HOP_SIZE] >>= 5; //shift value 8 bits over
for data transfer from 16 bit to 8 bit
                           OverlapInR[i + HOP_SIZE] = 32767*sin(2*22/7*i);
                           OverlapInR[i + HOP_SIZE] >>= 5; //shift value 8 bits over
for data transfer from 16 bit to 8 bit
                    }
              }

              /* Create windowed/not windowed buffer for processing */
        for (i = 0; i < WND_LEN; i++) {
```

```
            BufferL[i] = OverlapInL[i];
            BufferR[i] = OverlapInR[i];
        }

            /* Convert real data to "pseudo"-complex data (real, 0) */
            /* Int32 complex = Int16 real (MSBs) + Int16 imag (LSBs) */
            for (i = 0; i < FFT_LENGTH; i++)
            {
                  complex_data[2*i] = BufferR[i]; // place audio data (Real) in each
even index of complex_data
                  complex_data[2*i+1] = 0;         // place a 0 (Imag) in each odd
index of complex_data
            }


            /* Perform FFT */
            //cfft32(complex_data, HOP_SIZE, SCALE);
            cfft_SCALE(complex_data, FFT_LENGTH); //11841 cycles later...

            /* Perform bit-reversing */
            cbrev(complex_data, complex_data, FFT_LENGTH);


            /* Extract real and imaginary parts */
            for (i = 0; i < FFT_LENGTH; i++) {
                  realR[i] = complex_data[2*i];
                  imagR[i] = complex_data[2*i+1];

            }

            // Find the Power of the audio signal using the cfft results and scale by
1/2
            for(i = 0; i < FFT_LENGTH; i++) { // square the real vector and the
imaginary vector
                  realR[i] = realR[i] * realR[i];
                  imagR[i] = imagR[i] * imagR[i];
            }

            // ADD
            for(i = 0; i < FFT_LENGTH; i++) {
                  PSD_Result[i] = realR[i] + imagR[i];
            }
            // Scale result by dividing again, because im not sure if I have the sqrt
C library runtime fuction
            printf("FFT Result Points = ");
            for(i = 0; i < FFT_LENGTH; i++) {
                  PSD_Result_sqrt[i] = sqrt(PSD_Result[i]); // WARNING DOUBLE USED
IN SQRT
                  printf("%i, ", PSD_Result_sqrt[i]);
            }
```

==========END CFFT_SCALE CODE==========

This function worked. It was able to produce an frequency spectrum with two distinct peaks representing a tone reflected.

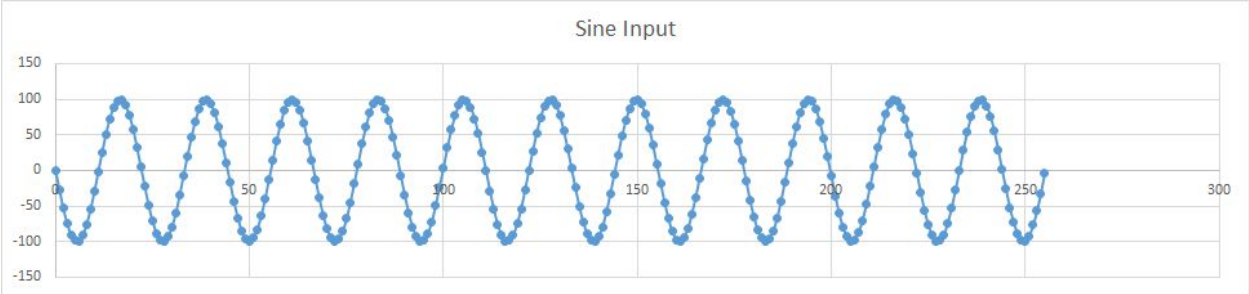


Photo of test sine wave

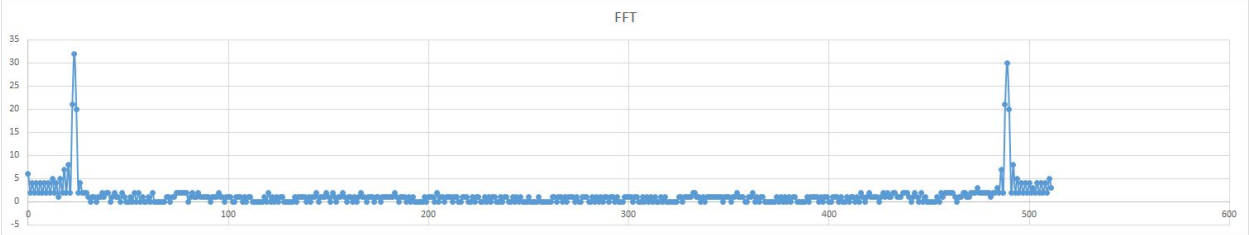


Photo of FFT produced. The noise here is a bit too much but since the values were being bit shifted to avoid perform faster I can minimize the presence of noise by letting back on bit shifting until the tone is at its maximum representation.

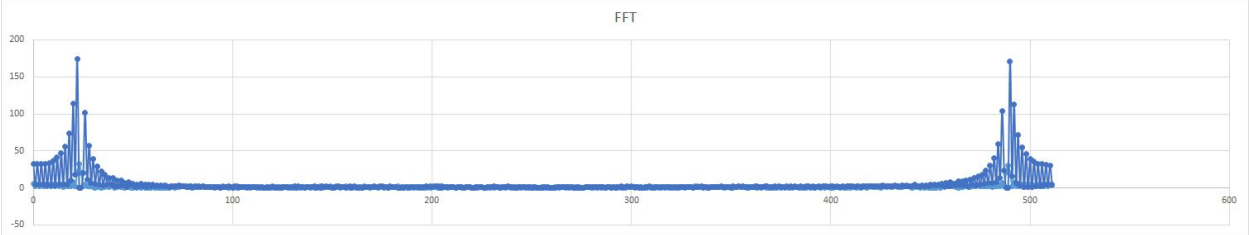


Photo of bit adjusted FFT spectrum. The drawback of letting back on bit shifting back 5 instead of 8 can be seen as higher magnitudes at the tone frequency. This is fine since the LED display will not need much accuracy.

CFFT_SCALE was tracked to an older version of the same DSP library used previously so it was reasoned that installing the older library and using those functions would be successful. However installing the DSP library into the master project was unsuccessful.

The DSP library for the ezDSPc5535 was found here
http://software-dl.ti.com/libs/c55_dsplib/latest/index_FDS.html

Graphic Equalizer

A requirement for the synthesizer was having a graphic equalizer. That is a set of knobs that filtered the frequency bands of the audio to the levels defined by the user.
The first method of approach was creating a frequency sampling filter. It would have logarithmically spaced bands that would appear to have linear interpolation between each set knob frequency on a log scale. These bands had to be logarithmically spaced at intervals that would line up exactly on the frequency points being specified by the knobs. This was already achieved in matlab thanks to a previous class. It had to be converted however for two reasons. Matlab code had to become C++ code and the values of everything had to be readjusted for only five values which made five bands between values. This was achieved and can be seen as code below.

==========Graphic Equalizer Code via Frequency Sampling==========

```
//Create 128 point Equalizer filter
DATA Knob_Values[5] = {1, 10, 100, 200, 255};
Uint8 EqKIndex[5] = {1, 3, 9, 27, 81};
Uint8 Interval_Lengths[4] = {2, 6, 18, 54};
DATA EqualizerFD[256];
EqualizerFD[0] = Knob_Values[0];
printf("\n%i,",EqualizerFD[0]);
EqualizerFD[1] = Knob_Values[0];
printf("%i,",EqualizerFD[1]);
Uint8 l;
Uint8 m;
   Uint8 n;
   int Interp_M;
   for(l = 0; l < 4; l++) {
    //make array
    n = 1;
    Interp_M = (Knob_Values[l+1] - Knob_Values[l]) / Interval_Lengths[l];
       for(m = EqKIndex[l]+1; m < EqKIndex[l+1]; m++) {
       EqualizerFD[m] = Knob_Values[l] + Interp_M * n;
       n++;
       printf("%i,",EqualizerFD[m]);
       }
       EqualizerFD[m] = Knob_Values[l+1];
       printf("%i,",EqualizerFD[m]);
       }
       for(l = 82; l < 128; l++) {
       EqualizerFD[l] = Knob_Values[4];
       printf("%i,",EqualizerFD[l]);
       }
       for(i = 0;i < 128; i++) {
       EqualizerFD[255-i] = EqualizerFD[i];
       printf("%i, ",EqualizerFD[128+i]);
       }
//=============================================================================
       for(i = 0;i < 256; i++) {
              if( EqualizerFD[i] )
```

```
            realR[i] = realR[i]*EqualizerFD[i];
        }
```

========== End Graphic Equalizer Code via Frequency Sampling==========

Using printf statements we were able to port multiple scenarios of inputs into excel and observe the linear interpolation on the log scale. Unfortunately this had to be dropped out as FFT along with IFFT would not run without the DSP library in the master project.

The second approach to making an equalizer was with the constraints of not having a DSP library taken into account. Instead of having a frequency spectrum of points as a filter the equalizer would have to be in the form of difference equations with predetermined coefficients. These difference equations could account as five bandpasses with knob values determining magnitude of effect. The bandpass results could then be averaged together essentially creating a variable comb filter. To make sure any difference equation worked a simple lowpass difference equation was created and tested.

==========Simple Lowpass Difference Equation Code==========

```
/* Standard C
includes */
                #include <std.h>
                #include <stdio.h>
                #include <math.h>
                #include <tsk.h>
                #include "Dsplib.h"
                #include "Dsplib_c.h"
                #include "MISC.H"
                #include "audio/singen.h"
                #include "global_vars.h"
                #include "TMS320.H"


                #pragma DATA_SECTION(Buffer, "Buffer");
                Int16 Buffer[256];
                //#pragma DATA_ALIGN (complex_data, 4)


                Int32 x[128];
```

```
Int32 Bk[3] = {10, 14, 10};

Int32 Ak = 34;

Int32 y[128];

Int32 First_Term = 0;

Int32 Second_Term = 0;

Int32 Third_Term = 0;

//Int16 nx = 128;

//Int16 nh = 3;

//DATA x[128];

//DATA h[3] = {10,14,10};

//DATA r[128];

//DATA dbuffer[5] = {0,0,0,0};

//Int16 oflag;

Int16 var = 0;



//Int16 delay = 500;

//Int16 nx = 128;

//Int16 nh = 501;

//DATA x[128];

//DATA h[501];

//DATA r[128];

//DATA dbuffer[501];

//Int16 oflag;

//Int16 var = 0;



Void eq_tsk( Void ) {

        volatile Int16 counter = 0;

        SinState EQSine;

        sin_compute_params(&EQSine, 6000);
```

```
Int16 i;
while(1) {
	// eq code
	printf("\nSine Wave ");
	for (i = 0; i < 128; ++i) {
		x[i] = sin_gen(&EQSine, 0);
		x[i] = x[i]/2048;
	//	printf("%d,", x[i]);
	}

	printf("\nFilter ");
	First_Term = Bk[0]*x[0];
	Second_Term = 0;
	Third_Term = 0;
	y[0] = First_Term/Ak;
	First_Term = Bk[0]*x[1];
	Second_Term = Bk[1]*x[0];
	Third_Term = 0;
	y[1] = First_Term + Second_Term;
	y[1] = y[1]/Ak;
	First_Term = Bk[0]*x[2];
	Second_Term = Bk[1]*x[1];
	Third_Term = Bk[2]*x[0];
	y[2] = First_Term + Second_Term + Third_Term;
	y[2] = y[2]/Ak;

	for (var = 3; var < 128; ++var) {
		First_Term = Bk[0]*x[var];
		Second_Term = Bk[1]*x[var-1];
```

```
                        Third_Term = Bk[2]*x[var-2];
                        y[var] = First_Term + Second_Term + Third_Term;
                        y[var] = y[var]/Ak;
//                      y[i] = (Bk[0]*x[i] + Bk[1]*x[i-1] +
Bk[2]*x[i-2])/Ak;
                        //printf("%d,", y[i]);
                }


////            Uint8 Knob_Values[5] = {pots[0], pots[1], pots[2], pots[3],
pots[4]};
//              Int16 Knob_Values[5] = {1, 10, 20, 40, 30};
//              //printf("%d %d %d %d %d\n", Knob_Values[0],
Knob_Values[1], Knob_Values[2], Knob_Values[3], Knob_Values[4]);
//              Int16 EqKIndex[5] = {1, 3, 9, 27, 81};
//              Int16 Interval_Lengths[4] = {2, 6, 18, 54};
//              Int16 EqualizerFD[256];
//              EqualizerFD[0] = Knob_Values[0];
//              //printf("\n%i,",EqualizerFD[0]);
//              EqualizerFD[1] = Knob_Values[0];
//              //printf("%i,",EqualizerFD[1]);
//              Int16 l;
//              Int16 m;
//              Int16 n;
//              Int16 Interp_M;
//              for(l = 0; l < 4; l++) {
//              //make array
//                      n = 1;
//                      Interp_M = (Knob_Values[l+1] - Knob_Values[l]) /
Interval_Lengths[l];
//                      for(m = EqKIndex[l]+1; m < EqKIndex[l+1]; m++) {
//                              EqualizerFD[m] = Knob_Values[l] + Interp_M *
n;
```

```
//                              n++;
//                              //printf("%i,",EqualizerFD[m]);
//                      }
//                      EqualizerFD[m] = Knob_Values[l+1];
//                      //printf("%i,",EqualizerFD[m]);
//              }
//              for(l = 82; l < 128; l++) {
//                      EqualizerFD[l] = Knob_Values[4];
//                  //printf("%i,",EqualizerFD[l]);
//              }
//              for(i = 0;i < 128; i++) {
//                      EqualizerFD[255-i] = EqualizerFD[i];
//                      //printf("%i,",EqualizerFD[128+i]);
//              }
//              printf("\n");
//              for(i=0;i<256;i++) {
//                      complex_data[2*i] = EqualizerFD[i];
//                      complex_data[2*i+1] = 0;
//                      //printf("%d, ", complex_data[2*i]);
//              }
//              for(i=0;i<512;i++) {
//                      printf("%d, ", complex_data[i]);
//              }
//              cifft_SCALE(complex_data, 256);
                //cfft_SCALE(complex_data, 256);
                //cbrev(complex_data, complex_data, 256);
                TSK_sleep(100);
        }
}
```

==========End Simple Lowpass Difference Equation Code==========

Using matlab the coefficients for the difference equation were determined so using this code and a sine wave input and matlab the code was verified to be a working FIR filter. It was then made into an IIR filter with three Aks and three Bks and verified again. Again matlab was used to figure out how many Aks and Bks were expected to be handled. Then the code was expanded to work as five bandpasses. These bandpasses were made based off the knob frequencies in the frequency sampling filter before and had to be IIR to have a faster transition or rather steeper peak in the frequency spectrum.
==========Difference Equation Graphic Equalizer Code==========

```
/*
Stand
ard C
inclu
des
*/
        #include <std.h>
        #include <stdio.h>
        #include <math.h>
        #include <tsk.h>
        #include "Dsplib.h"
        #include "Dsplib_c.h"
        #include "MISC.H"
        #include "audio/singen.h"
        #include "global_vars.h"
        #include "TMS320.H"


        #pragma DATA_SECTION(Buffer, "Buffer");
        Int16 Buffer[256];
        //#pragma DATA_ALIGN (complex_data, 4)
        #pragma DATA_ALIGN (left_ping_eq, 4)
        Int16 left_ping_eq[I2S_DMA_BUFFER_SIZE];


        #pragma DATA_ALIGN (left_pong_eq, 4)
```

```
Int16 left_pong_eq[I2S_DMA_BUFFER_SIZE];


#pragma DATA_ALIGN (right_ping_eq, 4)
Int16 right_ping_eq[I2S_DMA_BUFFER_SIZE];


#pragma DATA_ALIGN (right_pong_eq, 4)
Int16 right_pong_eq[I2S_DMA_BUFFER_SIZE];


Int32 x[128];
Int32 Ak1[3] = {10000, -19794, 9801};
Int32 Bk1[3] = {102, 0, -102};
Int32 Ak2[3] = {10000, -17752, 7921};
Int32 Bk2[3] = {1043, 0, -1043};
Int32 Ak3[3] = {10000, -15416, 6241};
Int32 Bk3[3] = {1880, 0, -1880};
Int32 Ak4[3] = {10000, -10880, 4761};
Int32 Bk4[3] = {2618, 0, -2618};
Int32 Ak5[3] = {10000, 4782, 3481};
Int32 Bk5[3] = {3257, 0, -3257};
Int32 y1[128] =
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0};
Int32 Next_y1[128];
Int32 y2[128];
Int32 Next_y2[128];
Int32 y3[128];
Int32 Next_y3[128];
Int32 y4[128];
```

```
Int32 Next_y4[128];
Int32 y5[128];
Int32 Next_y5[128];
Int32 y[128];
Int32 First_Term = 0;
Int32 Second_Term = 0;
Int32 Third_Term = 0;
//Int16 nx = 128;
//Int16 nh = 3;
//DATA x[128];
//DATA h[3] = {10,14,10};
//DATA r[128];
//DATA dbuffer[5] = {0,0,0,0};
//Int16 oflag;
Int16 var = 0;


//Int16 delay = 500;
//Int16 nx = 128;
//Int16 nh = 501;
//DATA x[128];
//DATA h[501];
//DATA r[128];
//DATA dbuffer[501];
//Int16 oflag;
//Int16 var = 0;


Void eq_tsk( Void ) {
        volatile Int16 counter = 0;
        SinState EQSine;
```

```
        sin_compute_params(&EQSine, 6000);
        Int16 i;
        for (i = 0; i < 128; ++i) {
                x[i] = sin_gen(&EQSine, 0);
        }
        Next_y1[0] = Bk1[0]*x[0]/Ak1[0];
        Next_y1[1] = (Bk1[0]*x[1]+Bk1[1]*x[0]-Ak1[1]*y1[0])/Ak1[0];
        Next_y1[2] =
(Bk1[0]*x[2]+Bk1[1]*x[1]+Bk1[2]*x[0]-Ak1[1]*y1[1]-Ak1[2]*y1[0])/Ak1[0];
        Next_y2[0] = Bk2[0]*x[0]/Ak2[0];
        Next_y2[1] = (Bk2[0]*x[1]+Bk2[1]*x[0]-Ak2[1]*y2[0])/Ak2[0];
        Next_y2[2] =
(Bk2[0]*x[2]+Bk2[1]*x[1]+Bk2[2]*x[0]-Ak2[1]*y2[1]-Ak2[2]*y2[0])/Ak2[0];
        Next_y3[0] = Bk3[0]*x[0]/Ak3[0];
        Next_y3[1] = (Bk3[0]*x[1]+Bk3[1]*x[0]-Ak3[1]*y3[0])/Ak3[0];
        Next_y3[2] =
(Bk3[0]*x[2]+Bk3[1]*x[1]+Bk3[2]*x[0]-Ak3[1]*y3[1]-Ak3[2]*y3[0])/Ak3[0];
        Next_y4[0] = Bk4[0]*x[0]/Ak4[0];
        Next_y4[1] = (Bk4[0]*x[1]+Bk4[1]*x[0]-Ak4[1]*y4[0])/Ak4[0];
        Next_y4[2] =
(Bk4[0]*x[2]+Bk4[1]*x[1]+Bk4[2]*x[0]-Ak4[1]*y4[1]-Ak4[2]*y4[0])/Ak4[0];
        Next_y5[0] = Bk5[0]*x[0]/Ak5[0];
        Next_y5[1] = (Bk5[0]*x[1]+Bk5[1]*x[0]-Ak5[1]*y5[0])/Ak5[0];
        Next_y5[2] =
(Bk5[0]*x[2]+Bk5[1]*x[1]+Bk5[2]*x[0]-Ak5[1]*y5[1]-Ak5[2]*y5[0])/Ak5[0];
        while(1) {
//              Ak1[i]*y1[i]+Ak1[i-1]*y1[i-1]+Ak1[i-2]*y1[i-2] =
Bk1[i]*x[i]+Bk1[i-1]*x[i-1]+Bk1[i-2]*x[i-2]
//              y1[i] =
(Bk1[0]*x[i]+Bk1[1]*x[i-1]+Bk1[2]*x[i-2]-Ak1[1]*y1[i-1]-Ak1[2]*y1[i-2])/Ak1[0];
                y1[0] = Next_y1[0];
                y1[1] = Next_y1[1];
                y1[2] = Next_y1[2];
                for (i = 3; i < 128; i++) {
```

```
                        y1[i] =
(Bk1[0]*x[i]+Bk1[1]*x[i-1]+Bk1[2]*x[i-2]-Ak1[1]*y1[i-1]-Ak1[2]*y1[i-2])/Ak1[0];
                }

                Next_y1[0] = y1[125];

                Next_y1[1] = y1[126];

                Next_y1[2] = y1[127];

                //there is only one Bandpass done now

                y2[0] = Next_y2[0];

                y2[1] = Next_y2[1];

                y2[2] = Next_y2[2];

                for (i = 3; i < 128; i++) {

                        y2[i] =
(Bk2[0]*x[i]+Bk2[1]*x[i-1]+Bk2[2]*x[i-2]-Ak2[1]*y2[i-1]-Ak2[2]*y2[i-2])/Ak2[0];
                }

                Next_y2[0] = y2[125];

                Next_y2[1] = y2[126];

                Next_y2[2] = y2[127];

                // there is now two bandpasses done

                y3[0] = Next_y3[0];

                y3[1] = Next_y3[1];

                y3[2] = Next_y3[2];

                for (i = 3; i < 128; i++) {

                        y3[i] =
(Bk3[0]*x[i]+Bk3[1]*x[i-1]+Bk3[2]*x[i-2]-Ak3[1]*y3[i-1]-Ak3[2]*y3[i-2])/Ak3[0];
                }

                Next_y3[0] = y3[125];

                Next_y3[1] = y3[126];

                Next_y3[2] = y3[127];

                // there is now three bandpasses done

                y4[0] = Next_y4[0];

                y4[1] = Next_y4[1];

                y4[2] = Next_y4[2];
```

```
		for (i = 3; i < 128; i++) {
			y4[i] =
(Bk4[0]*x[i]+Bk4[1]*x[i-1]+Bk4[2]*x[i-2]-Ak4[1]*y4[i-1]-Ak4[2]*y4[i-2])/Ak4[0];
		}
		Next_y4[0] = y4[125];
		Next_y4[1] = y4[126];
		Next_y4[2] = y4[127];
		// fourth bandpass complete
		y5[0] = Next_y5[0];
		y5[1] = Next_y5[1];
		y5[2] = Next_y5[2];
		for (i = 3; i < 128; i++) {
			y5[i] =
(Bk5[0]*x[i]+Bk5[1]*x[i-1]+Bk5[2]*x[i-2]-Ak5[1]*y5[i-1]-Ak5[2]*y5[i-2])/Ak5[0];
		}
		Next_y5[0] = y5[125];
		Next_y5[1] = y5[126];
		Next_y5[2] = y5[127];
		// fifth bandpass complete
		//combine into y[]
		for (i = 0; i < 128; i++) {
			y[i] = y1[i]/5 + y2[i]/5 + y3[i]/5 + y4[i]/5 + y5[i]/5;
		}
		// determine which buffer to fill
		Int16 *left_output, *right_output;
		if (CSL_DMA1_REGS->DMACH0TCR2 & 0x0002) { // last xfer: pong
			left_output = left_pong_eq;
			right_output = right_pong_eq;
		} else {
			left_output = left_ping_eq;
			right_output = right_ping_eq;
```

```
                }
                for (i = 0; i < I2S_DMA_BUFFER_SIZE; i++) {
                        left_output[i] = y[i];
                        right_output[i] = y[i];
                }
//              First_Term = Bk[0]*x[0];
//              y[0] = First_Term/Ak;
//              First_Term = Bk[0]*x[1];
//              Second_Term = Bk[1]*x[0];
//              Third_Term = 0;
//              y[1] = First_Term + Second_Term;
//              y[1] = y[1]/Ak;
//              First_Term = Bk[0]*x[2];
//              Second_Term = Bk[1]*x[1];
//              Third_Term = Bk[2]*x[0];
//              y[2] = First_Term + Second_Term + Third_Term;
//              y[2] = y[2]/Ak;
//
//              for (var = 3; var < 128; ++var) {
//                      First_Term = Bk[0]*x[var];
//                      Second_Term = Bk[1]*x[var-1];
//                      Third_Term = Bk[2]*x[var-2];
//                      y[var] = First_Term + Second_Term + Third_Term;
//                      y[var] = y[var]/Ak;
////                    y[i] = (Bk[0]*x[i] + Bk[1]*x[i-1] + Bk[2]*x[i-2])/Ak;
//                      //printf("%d,", y[i]);
//              }


////            Uint8 Knob_Values[5] = {pots[0], pots[1], pots[2], pots[3],
pots[4]};
```

```
//             Int16 Knob_Values[5] = {1, 10, 20, 40, 30};
//             //printf("%d %d %d %d %d\n", Knob_Values[0], Knob_Values[1],
Knob_Values[2], Knob_Values[3], Knob_Values[4]);
//             Int16 EqKIndex[5] = {1, 3, 9, 27, 81};
//             Int16 Interval_Lengths[4] = {2, 6, 18, 54};
//             Int16 EqualizerFD[256];
//             EqualizerFD[0] = Knob_Values[0];
//             //printf("\n%i,",EqualizerFD[0]);
//             EqualizerFD[1] = Knob_Values[0];
//             //printf("%i,",EqualizerFD[1]);
//             Int16 l;
//             Int16 m;
//             Int16 n;
//             Int16 Interp_M;
//             for(l = 0; l < 4; l++) {
//             //make array
//                    n = 1;
//                    Interp_M = (Knob_Values[l+1] - Knob_Values[l]) /
Interval_Lengths[l];
//                    for(m = EqKIndex[l]+1; m < EqKIndex[l+1]; m++) {
//                           EqualizerFD[m] = Knob_Values[l] + Interp_M * n;
//                           n++;
//                           //printf("%i,",EqualizerFD[m]);
//                    }
//                    EqualizerFD[m] = Knob_Values[l+1];
//                    //printf("%i,",EqualizerFD[m]);
//             }
//             for(l = 82; l < 128; l++) {
//                    EqualizerFD[l] = Knob_Values[4];
//                 //printf("%i,",EqualizerFD[l]);
//             }
```

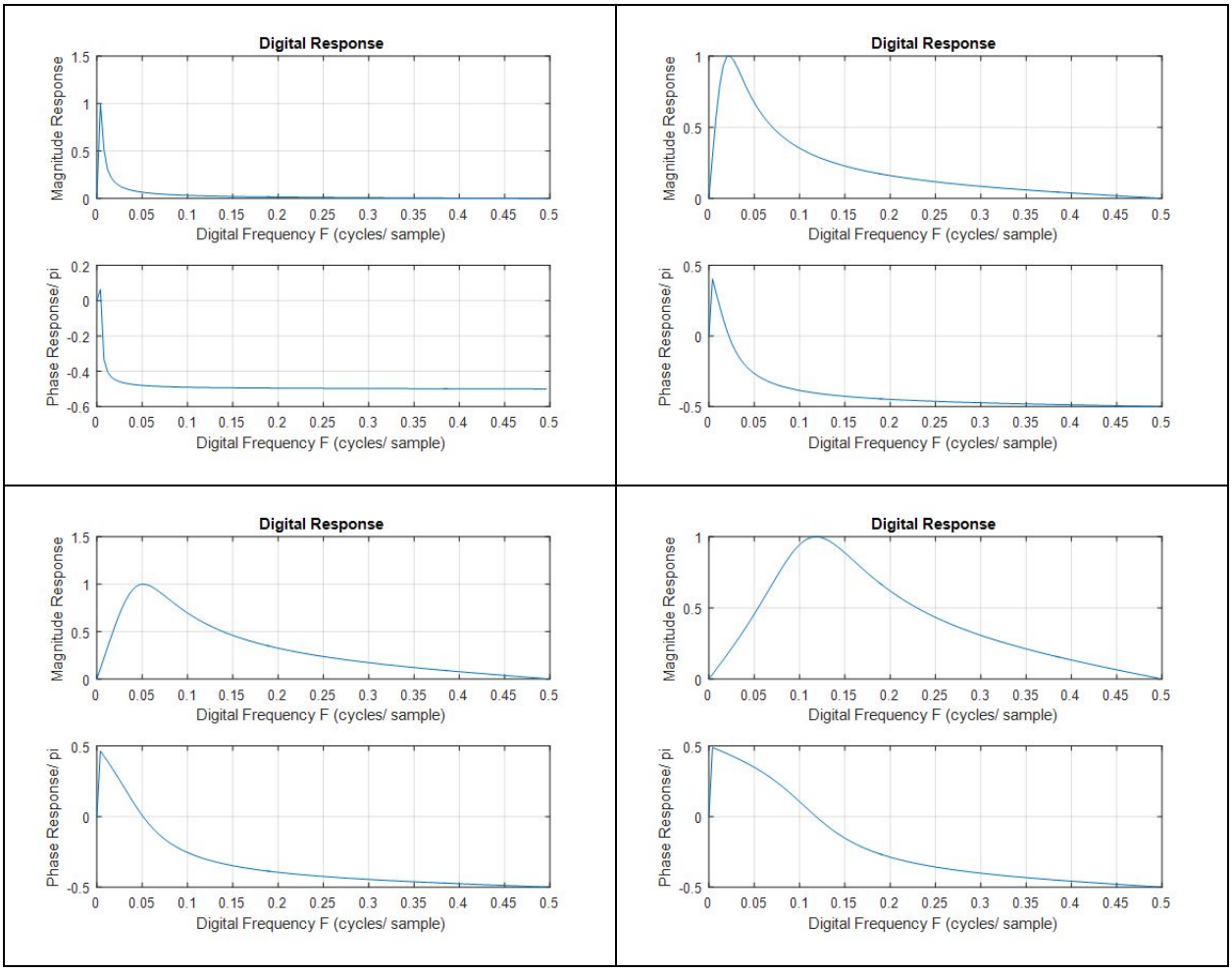
```
//              for(i = 0;i < 128; i++) {
//                      EqualizerFD[255-i] = EqualizerFD[i];
//                      //printf("%i,",EqualizerFD[128+i]);
//              }
//              printf("\n");
//              for(i=0;i<256;i++) {
//                      complex_data[2*i] = EqualizerFD[i];
//                      complex_data[2*i+1] = 0;
//                      //printf("%d, ", complex_data[2*i]);
//              }
//              for(i=0;i<512;i++) {
//                      printf("%d, ", complex_data[i]);
//              }
//              cifft_SCALE(complex_data, 256);
                //cfft_SCALE(complex_data, 256);
                //cbrev(complex_data, complex_data, 256);
                TSK_sleep(100);
        }
    }
```

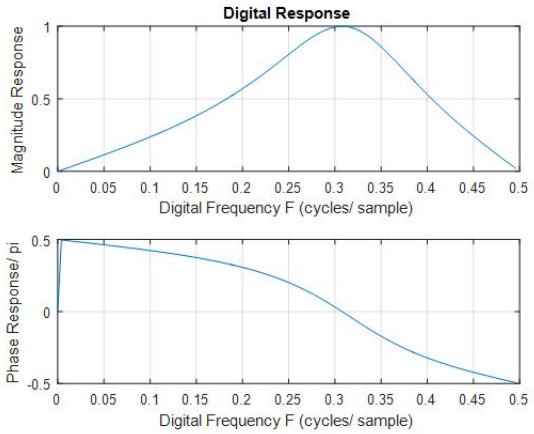
==========End Difference Equation Graphic Equalizer Code==========

Knob Freq	Digital Freq	Radians	Intensity	Pole Real	Pole Imaginary
187.5	0.00390625	0.024544	0.99	0.989701831	0.024295816
562.5	0.01171875	0.073631	0.89	0.887588506	0.065472462
1687.5	0.03515625	0.220893	0.79	0.770804683	0.17308998
5062.5	0.10546875	0.66268	0.69	0.543959035	0.424509798
15187.5	0.31640625	1.98803	0.59	-0.23909237	0.539383756

		9		5	

This table was used to calculate the location of poles and zeros for each bandpass





With the poles and zeros above these bandpasses are created. They are represented in the project's code as Aks and Bks below.

Ak	Bk	Scaled Ak	Scaled Bk
1.0000 -1.9794 0.9801	0.0102 0 -0.0102	10000 -19794 9801	102 0 -102
1.0000 -1.7752 0.7921	0.1043 0 -0.1043	10000 -17752 7921	1043 0 -1043
1.0000 -1.5416 0.6241	0.1880 0 -0.1880	10000 -15416 6241	1880 0 -1880
1.0000 -1.0880 0.4761	0.2618 0 -0.2618	10000 -10880 4761	2618 0 -2618
1.0000 0.4782 0.3481	0.3257 0 -0.3257	10000 4782 3481	3257 0 -3257

This table shows the results from matlab. Scaling was done so the fixed point DSP chip did not round out all the data

While the filters should have worked it was never verified as it turned out the audio memory was not easily transferable to another dual bank of "ping and pong registers." Writing the memory to the new bank caused massive corruption as it was verified by bypassing the filtration of the equalizer and directly inputting the data from audio generation. When this occurred the sound was garbled beyond recognition.

IV. Final State

System Design - Functional Decomposition (Level 1)

The system can be broken down as shown in figure 4.1. The operation of the system can be generalized as:

1. The user interacts with the device
    a. Presses a key on keyboard
    b. Sends a MIDI event
    c. Changes a parameter on the front panel
2. The ATmega reads in information from the user
3. The C5535 requests an update on the status of the system
4. The ATmega responds with the latest information on key presses, parameter changes and MIDI information
5. The C5535 generates a waveform based on the state of the system
6. The sound is enjoyed by the listener

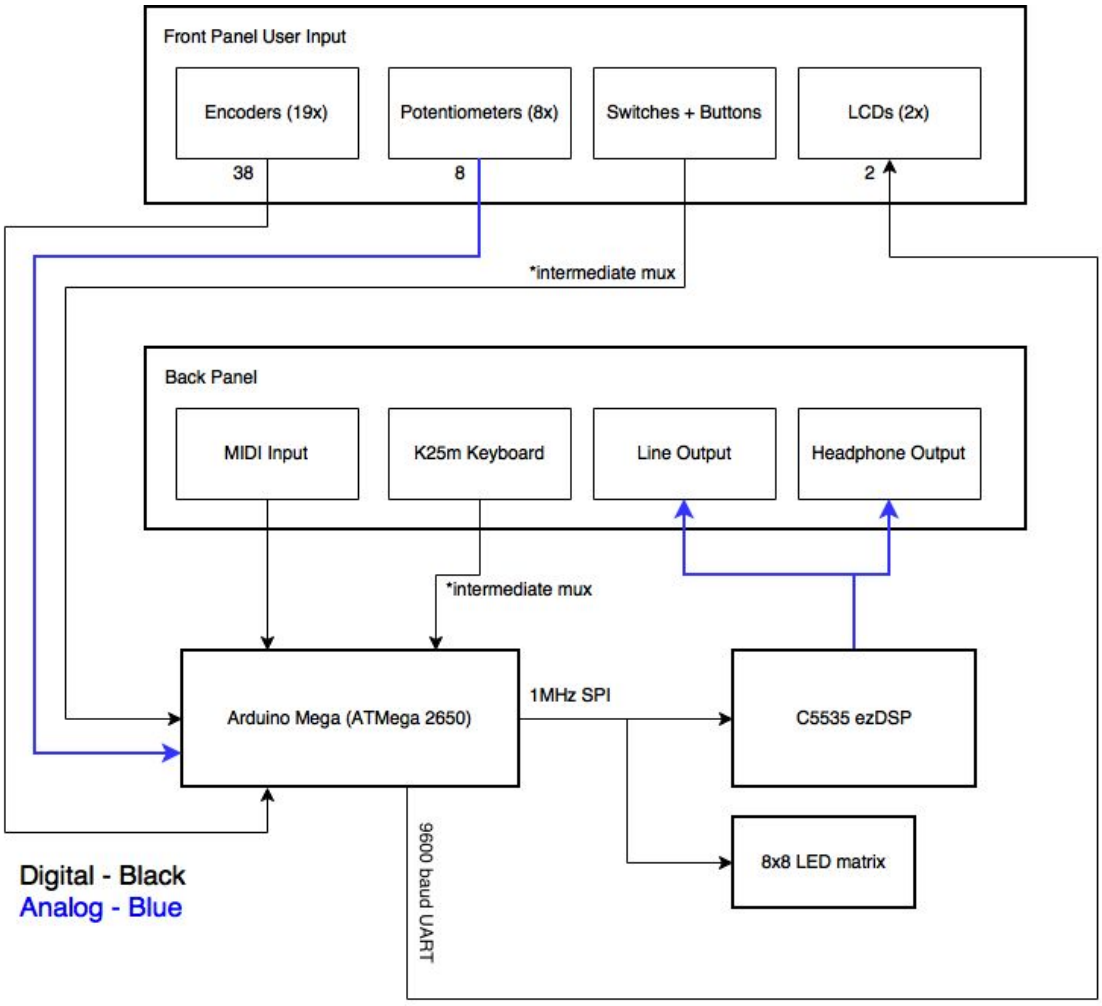


Figure 4.1: Level 1 block diagram of system

VII. Physical Construction and Integration

Physically the Danalog synthesizer consists of

1. Main PCB: The PCB connects all the devices together and functions mechanically to hold all the components neatly in place inside the enclosure.

2. Arduino Mega: The Arudino Mega functions to interface with all the user input controls. It communicates all the fundamental information to the C5535 via a SPI communication bus

3. TI ezDSP C5535: This device is responsible for interpreting the information sent by the Arduino and generating sound.

4. 3D printed chassis: Encloses all components

The PCB functions as the harness for all the front panel interface controls, which consist of rotary quadrature encoders, rotary potentiometers, linear potentiometers, and rotary switches. The organization of the user interface was decided by the team during the initial planning phase. All interconnections on the PCB were made to accommodate the initial user interface design.

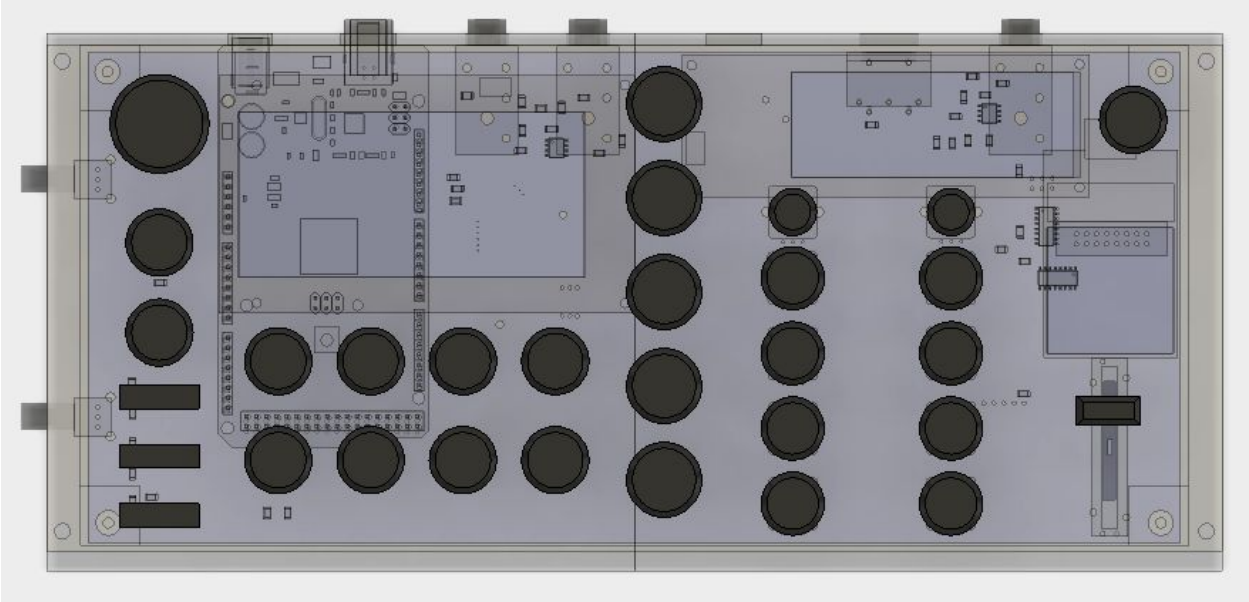


Figure 7.1: Overview of internal device construction and organization

Each device is routed to pins on the Arduino Mega board. Since the amount of IO needed was slightly more than the Arduino Mega provided we used multiplexers between the diode connected matrix keyboard (figure 7.2a) and between the rotary switches and front panel buttons (figure 7.2b)

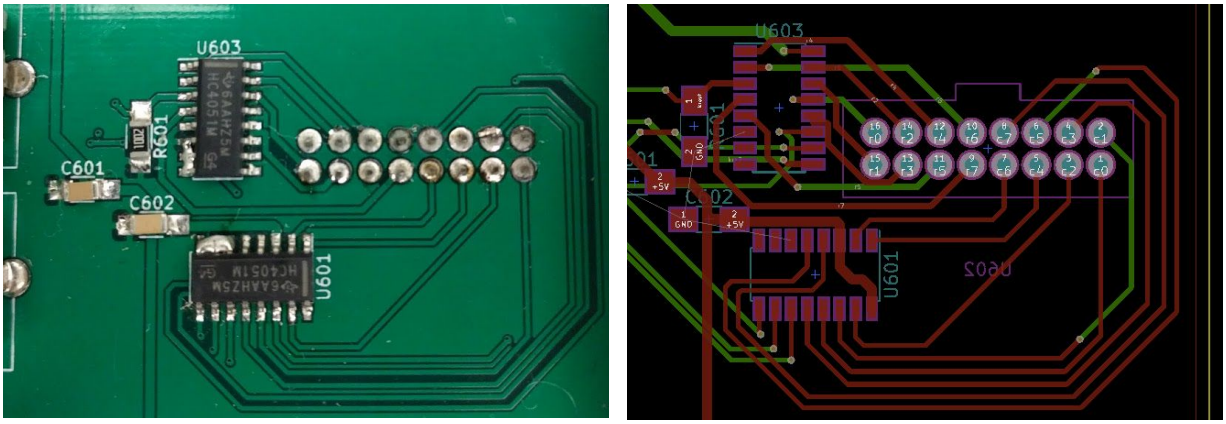


Figure 7.2a: Diode connected keyboard multiplexer layout

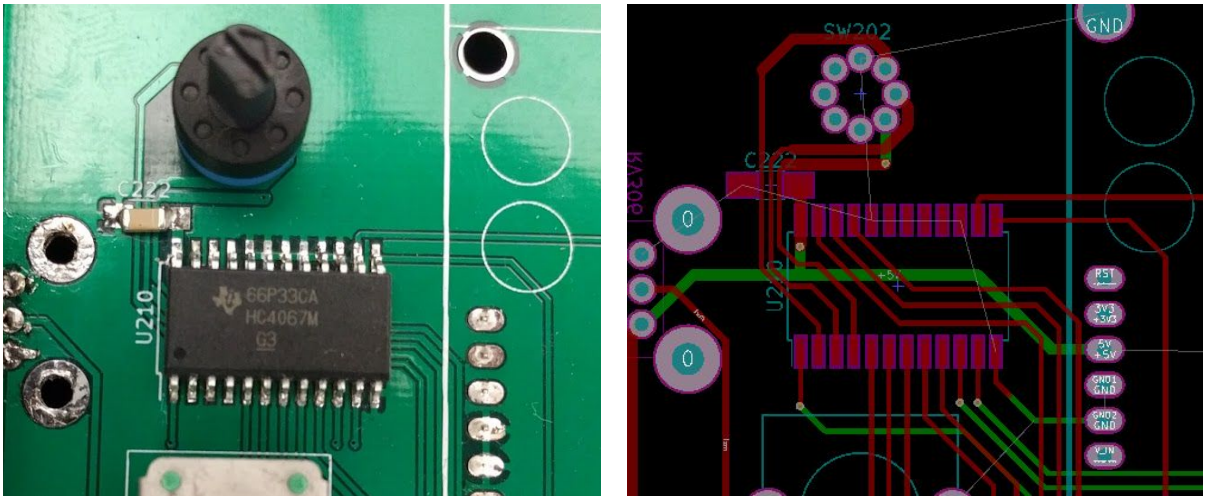


Figure 7.2b: Rotary switches and buttons multiplexer layout

The device also has two displays for outputting information about the state of the synthesis engine and the state of the effects processor. The displays were purchased from sparkfun as separate units not soldered to the the main circuit board. These displays were used because of their simple serial interface which allowed us to use a hardware UART to communicate with the display

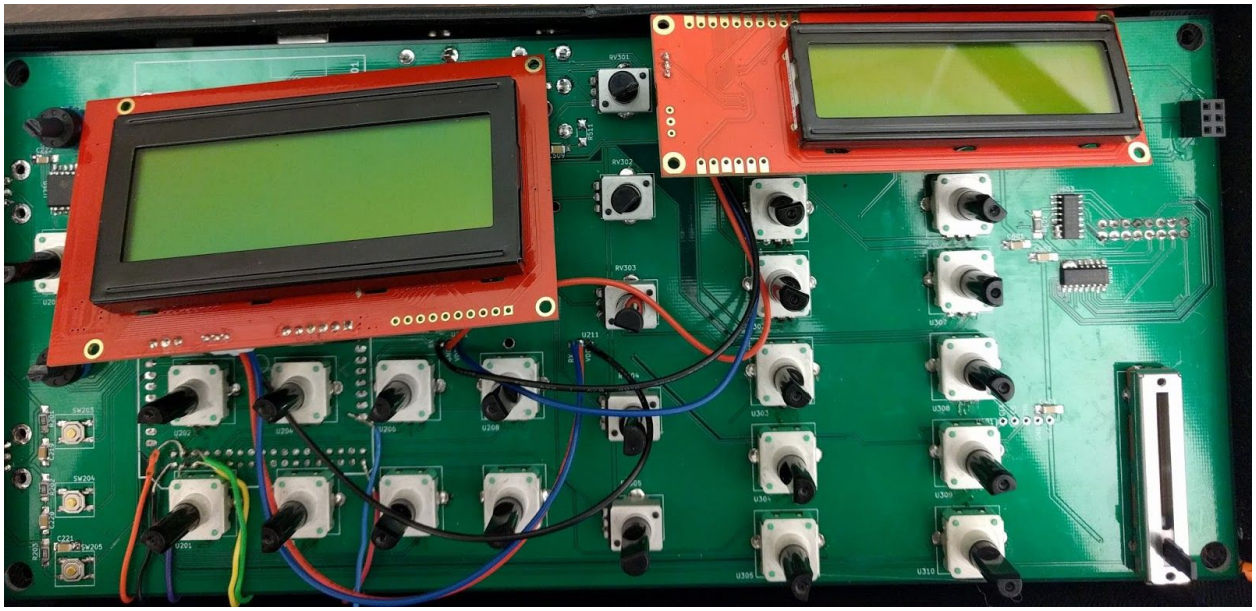


Figure 7.3: Both LCD displays connected to the main PCB via wires

Since the both the Arduino Mega and TI ezDSP both can be driven by 5 volts USB power there was no need to design any sort of power system. Additionally since the devices are low power, as USB devices usually are, there is no need for any form of heat sinking inside of the enclosure.

The chassis was 3D printed on Evan Lew's home 3D printer. Due to sizing constraints the chassis was printed in two halves and then glued together to form the final chassis. Figure 7.4 shows the 3D model of the chassis and figure 7.5 shows the real life chassis supported by the keyboard.

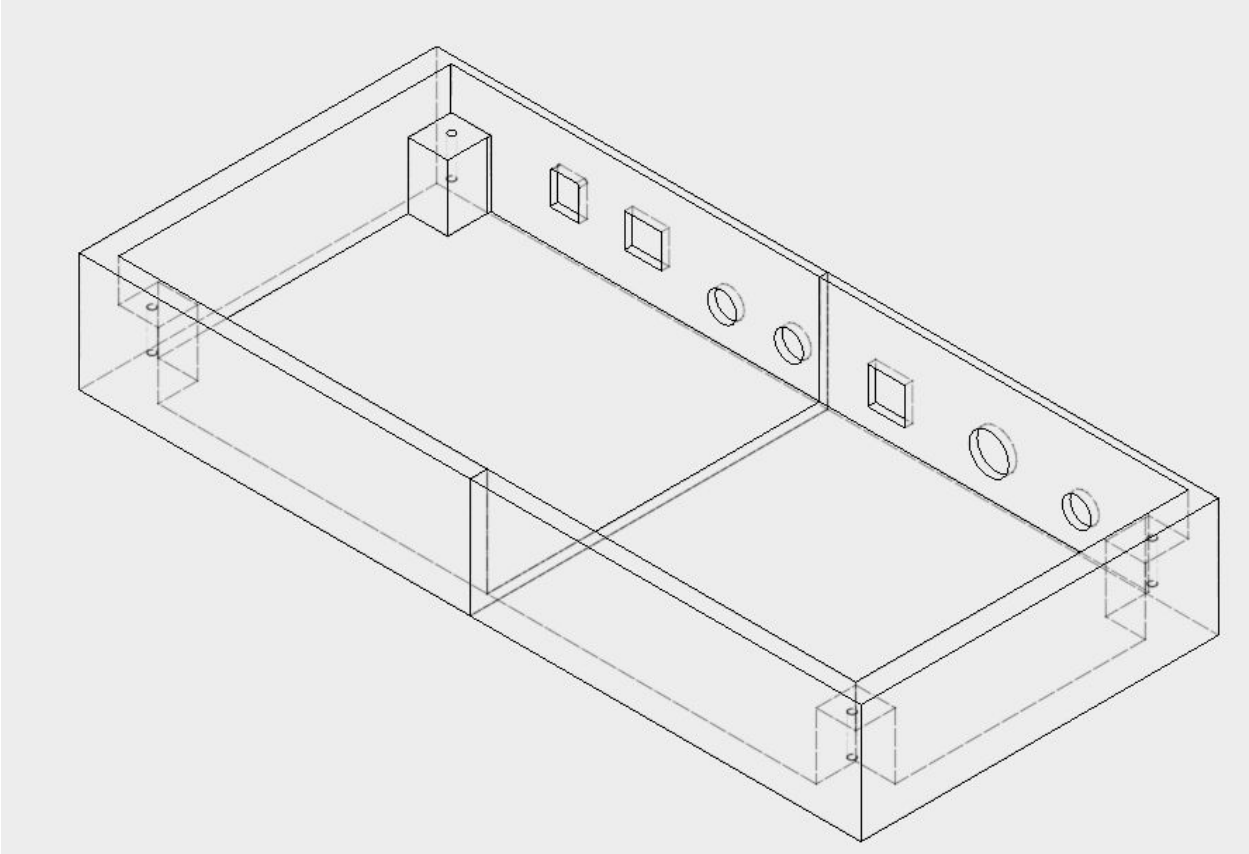


Figure 7.4: 3D model of the chassis

Figure 7.5: Chassis with internal hardware

V. Integrated System Tests and Results

Due to the ambitious and complex nature of the project we were not able to achieve all of our goals. However, we were able to have a product at the end that was on the path to achievement. At the end of our spring quarter, we had a functioning piece of hardware and functional synthesis engine.

The FM synthesis works. The latency is tested by having the arduino send a pin high when a key is pressed and measuring the delay between that transition and the start of the note being played.

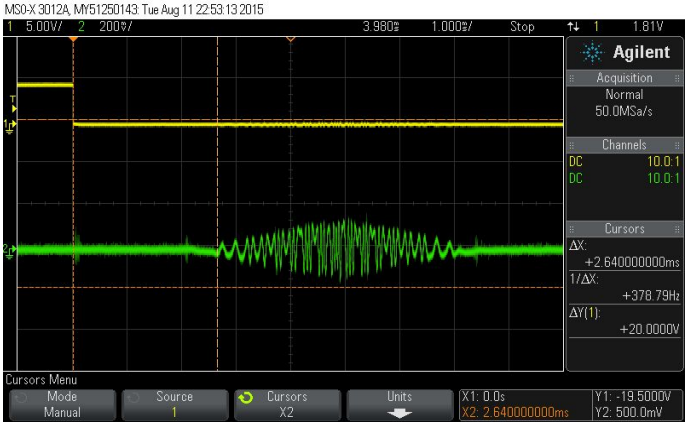


Figure XX. FM synthesis minimum latency test.

It should be noted that latency varied. We found the minimum latency to be 2.64 milliseconds while maximum extended to 5.36 milliseconds. This meets our specifications as it is not noticed by the human ear.

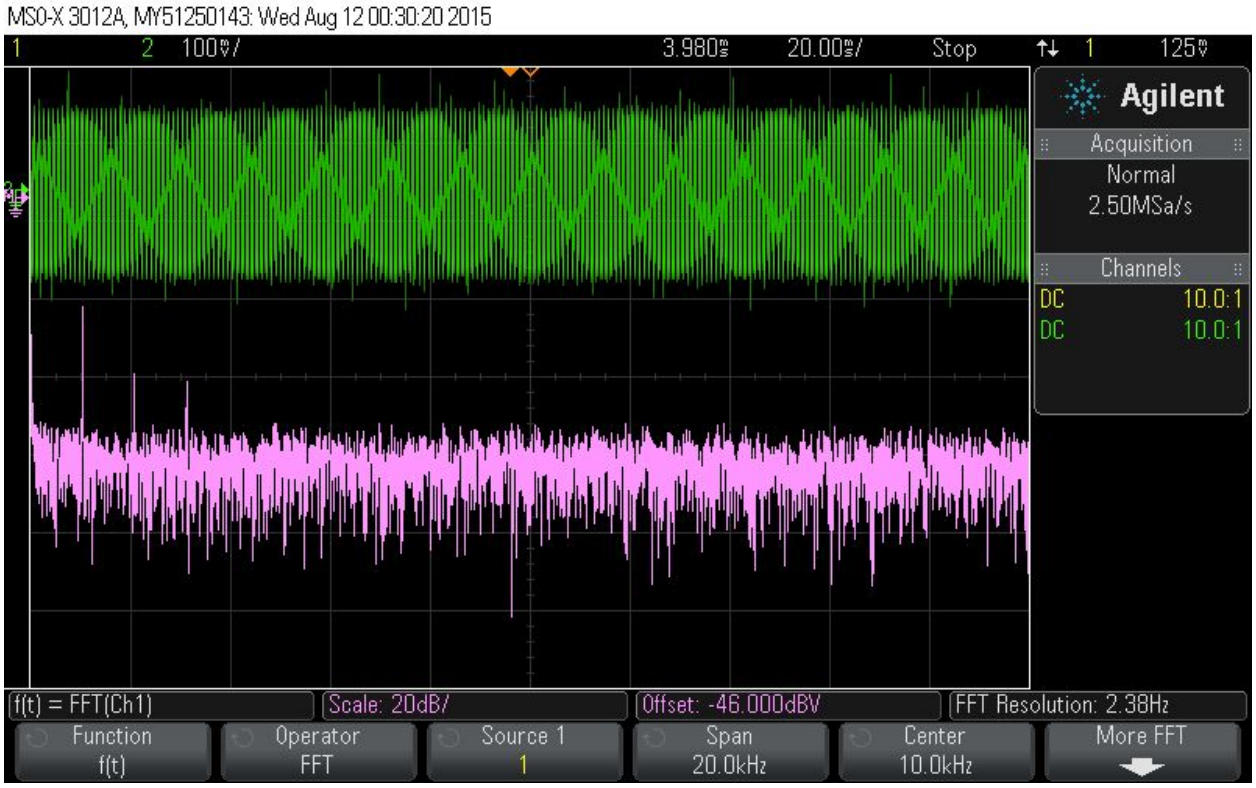


Figure XX. Signal to Noise Ratio Test.

The signal to noise ratio can be determined by having a singular tone play and performing an FFT on the signal. As shown above a tone peak appears but along with unintended harmonics. Using cursors the difference between the tone's peak and the noise level is around 45 dBV. This did not meet our original specification as we aimed to have 90 dBV. The output is admittedly a little noisy to the human ear but this could be due to the probes. While connecting the probes the noise became much more apparent with increased volume.

	Minimum	Maximum
Signal to Noise Ratio	44 dBV	44 dBV
Latency	2.64 milliseconds	5.36 milliseconds

VI. Summary: While not all the specifications were met the FM synthesis has been successfully created and works well. The output is a bit noisier when probing but sounds fine without. Latency is low enough for the synthesis to be considered in real time. Shortcomings were included to but not limited to failure to have working DSP library functions, losing progress with corrupted projects, and failure to keep audio data not corrupted upon transfer to a new bank. Having the DSP library work could be done by consulting Texas Instruments or posting on their forum about how to not just install the DSP library onto a computer but also on what is necessary to include the library in a project in code composer studio. To avoid having projects become beyond repair github would be used from the beginning so as to revert back to stable checkpoints easily. Avoiding audio corruption upon transfer could be potentially solved with data alignment or just having a simple project be used to debug the problem.