# Home Audio System Project
# ECE 2804

**Jacob Fast**

**Kofi Ofosu-Tuffour**

**Md. Adnan Sarker**

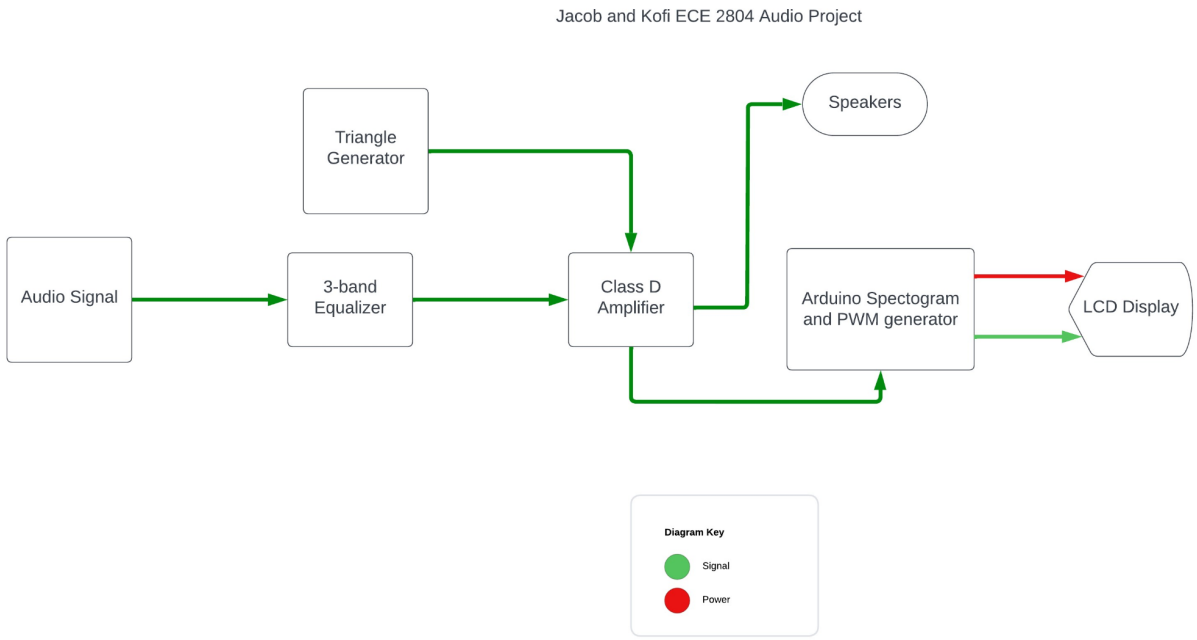
**November 28, 2023**

**Contents**

## Introduction

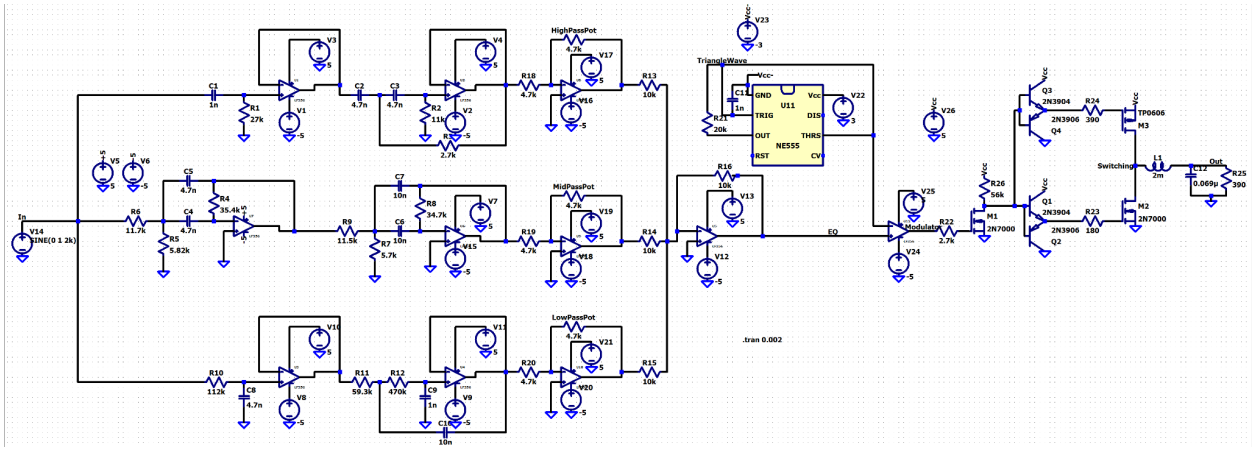
This semester we built a home audio entertainment system for use with headphones. In this audio system, we have a 3-band equalizer that takes an input audio signal, performs gain or attenuation on this signal, then re-combines the signals for the amplification. We also created a spectrogram of this signal to display to the user of what the equalizer is doing to the signal. One key aspect of this audio system is the choice of amplifier we are required to use. For this system, we are required to use a class D amplifier to amplify the audio signal from the equalizer. The class D amplifier is a topology that uses the power efficiency of switching mosfets in order to amplify the audio signal. It does this by modulating the incoming audio signal into a variable PWM wave, which is used to switch the output mosfets between power-rails and ground. This leverages the near zero power loss when using the mosfet as a switch rather than having a transistor in linear mode. The amplified PWM signal is then passed through a low-pass filter to reconstruct the original waveform for use for the headphones. In this document, we will go over how we formulated, tested, built, and validated our approach of this project.

Our overall system depicted below shows our general home audio structure.




Block Diagram of Project

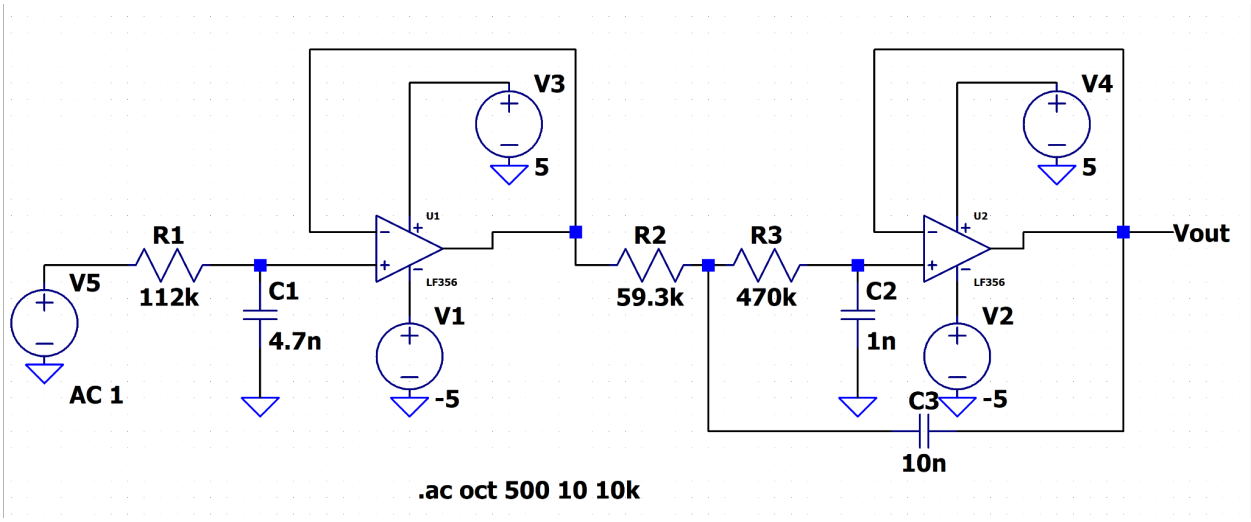
## Graphic Equalizer




Schematic of EQ and class D amp

The above image is the schematic of our entire circuit. This circuit consists of the graphic equalizer and class-D amplifier. The first subsystem is the graphic equalizer. The graphic equalizer consists of filters, variable gain amplifiers, and a summing amplifier. The input signal is split up and filtered by the three different bands. Then each band individually has its own variable gain, using a potentiometer. Finally, the outputs of the different variable gain are put back together using a summing amplifier. There are three different filters, a low-pass filter for the bass frequencies, a bandpass filter for the mid frequencies, and a high-pass filter for the treble frequencies. We decided to go with a low and high pass filter because we only have three bands and this allows us to cover a greater range of frequencies. Any noise generated at higher frequencies will be filtered out by the LC low-pass filter in the amplifier. The next decision we made was the cutoff frequencies and center frequencies for the filters. For our low-pass filter, we chose the cutoff frequency to be 300 Hz to preserve as much of the bass as possible For our mid filter we chose the center frequency to be 2kHz. This is right in the middle of what is considered middle frequencies and since our filter has a Q factor of 1 the bandwidth is the same as the center . For our high-pass filter we chose the cutoff frequency to be 6kHz to get the all treble frequencies. We chose to have our cutoff frequencies further apart because if they were close there would be huge dips of gain near the cutoff frequencies.

All of our filters were designed using the analog filter design tool provided to us. The low pass filter is pictured below.

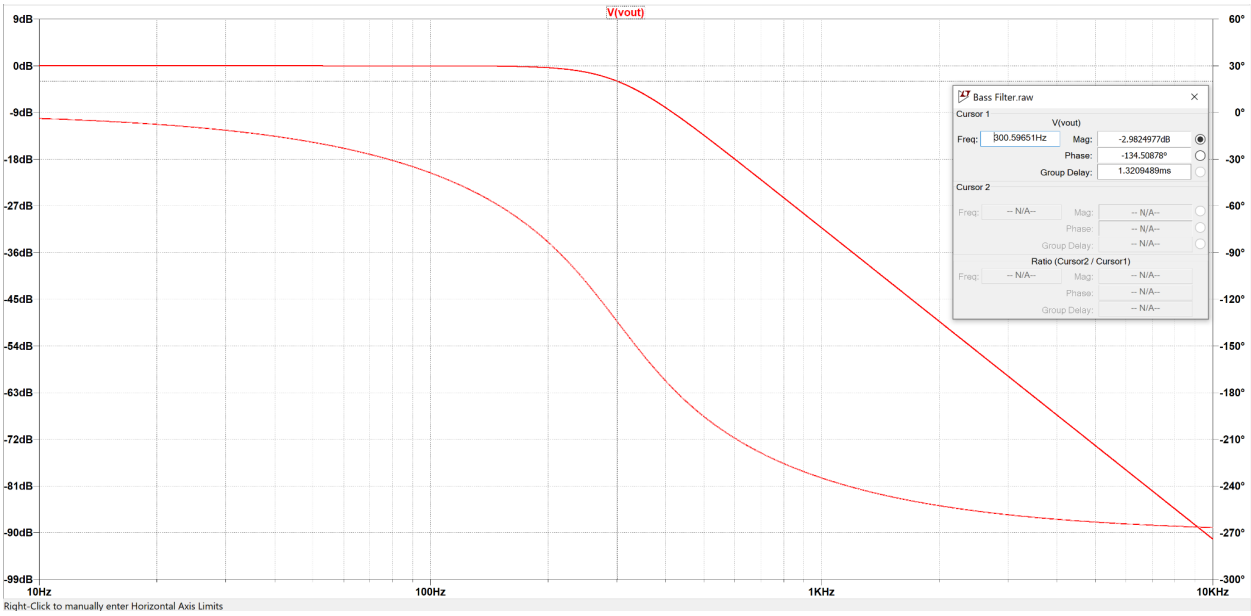



Low-pass Schematic

From this design we get the following frequency response, for the entire system.

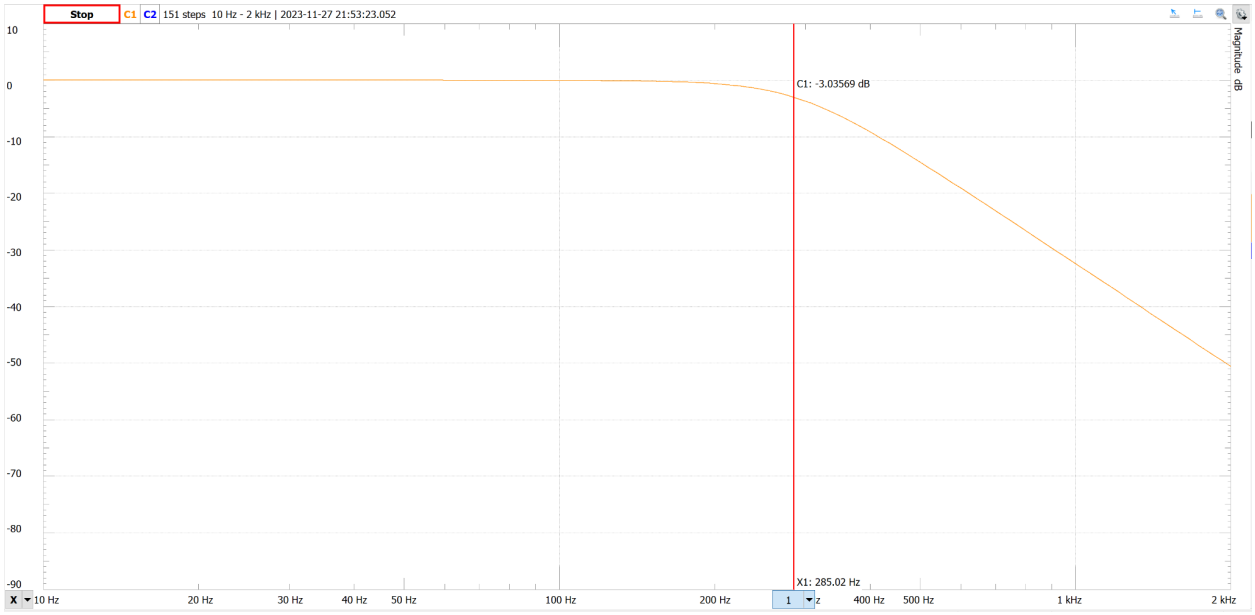
$$H(j\omega) = \frac{(\frac{1}{4.7nF * j\omega})}{\frac{1}{4.7nF * j\omega}+112k\Omega} * \frac{(\frac{1}{j\omega(10\,nF)})*(\frac{1}{j\omega(1\,nF)})}{(59.3k\Omega * 470k\Omega)+(\frac{1}{j\omega(10\,nF)})(59.3k\Omega * 470k\Omega)+(\frac{1}{j\omega 10\,nF})*(\frac{1}{j\omega(1\,nF)})}$$

To calculate the frequency response for an entire system you multiply the frequency responses of the individual stages. So the equation is the first stage's response multiplied by the second stage's response. Simulating in LTSpice we get the following frequency response.




Simulated Low-Pass Frequency Response

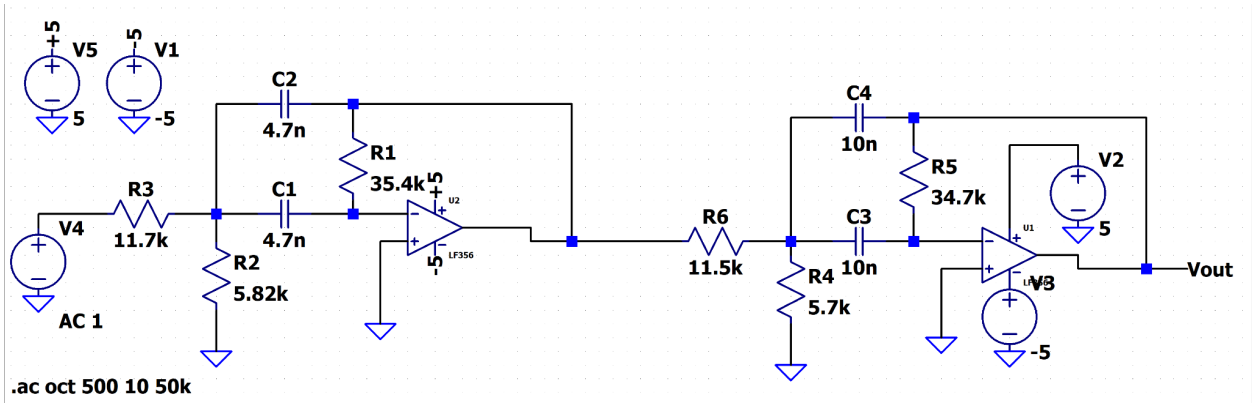
The shape matches what we would expect from a low-pass filter and the cutoff point is at the frequency we choose. The below image is the frequency response of the actual circuit.




Measured Low-Pass Frequency Response

The shape matches the simulation but the cutoff frequency is a little bit lower than the simulation. This can be explained by the fact that our components have tolerances and don't match the simulation value 100%. This small difference is acceptable.

The mid bandpass filter was also designed using the filter designing tool. We choose a center frequency of 2kHz. Our mid filter is pictured below.

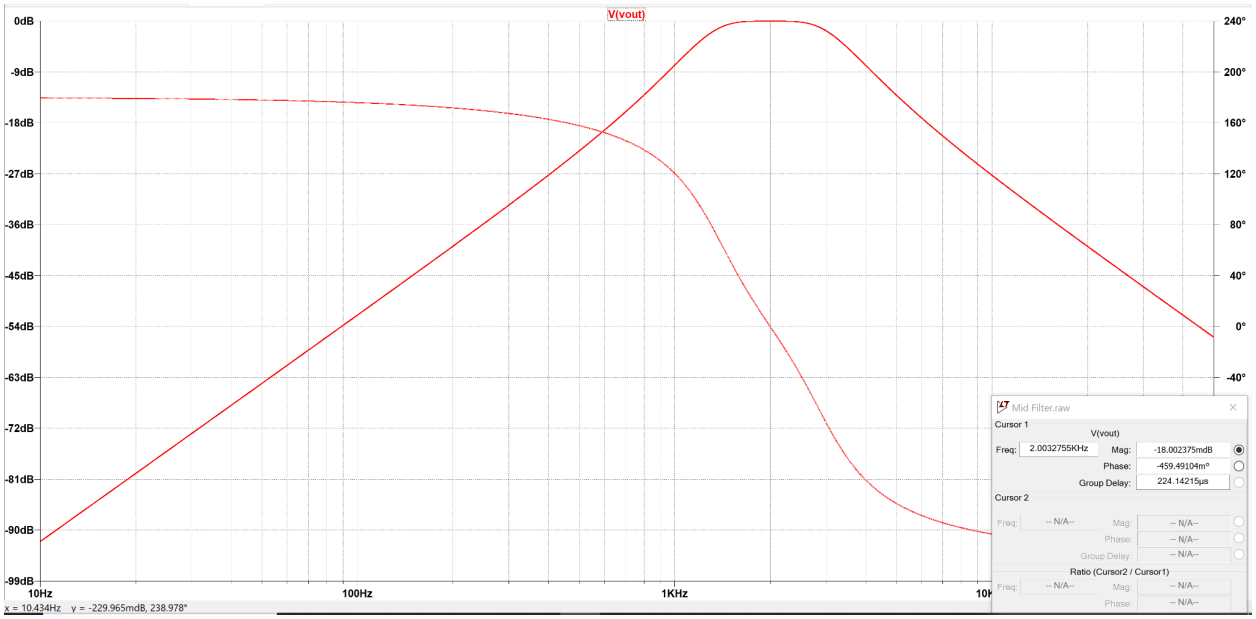



Mid Filter Schematic

The above filter has the following frequency response.

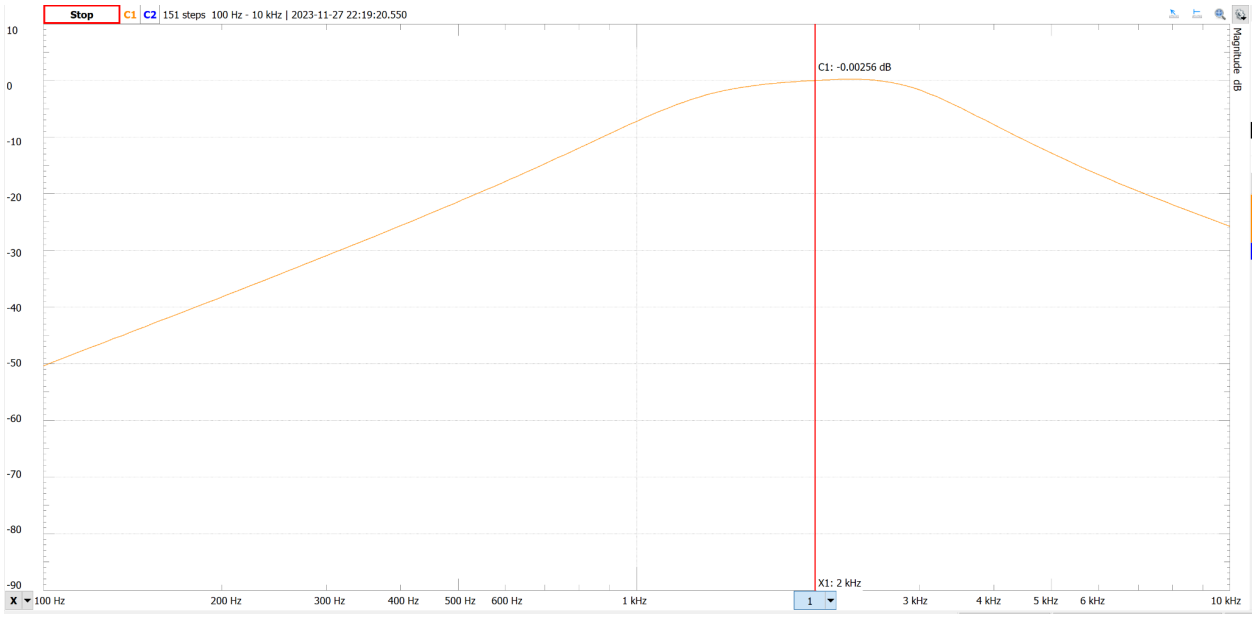
$$H(j\omega) = \frac{\frac{-j\omega}{(11.7k\Omega)(4.7nF)}}{(j\omega)^2+(j\omega)\frac{(4.7nF)+(4.7nF)}{(4.7nF)*(4.7nF)*(35.4k\Omega)}+\frac{1}{(35.4k\Omega)*(4.7nF)*(4.7nF)}\left(\frac{1}{(11.7k\Omega)}+\frac{1}{(5.82k\Omega)}\right)} * \frac{\frac{-j\omega}{(11.5k\Omega)(10nF)}}{(j\omega)^2+(j\omega)\frac{(10nF)+(10nF)}{(10nF)*(10nF)*(34.7k\Omega)}+\frac{1}{(34.7k\Omega)*(10nF)*(10nF)}\left(\frac{1}{(11.5k\Omega)}+\frac{1}{(5.7k\Omega)}\right)}$$

Simulating the circuit we get the following frequency response.




Mid Filter Simulated Frequency Response

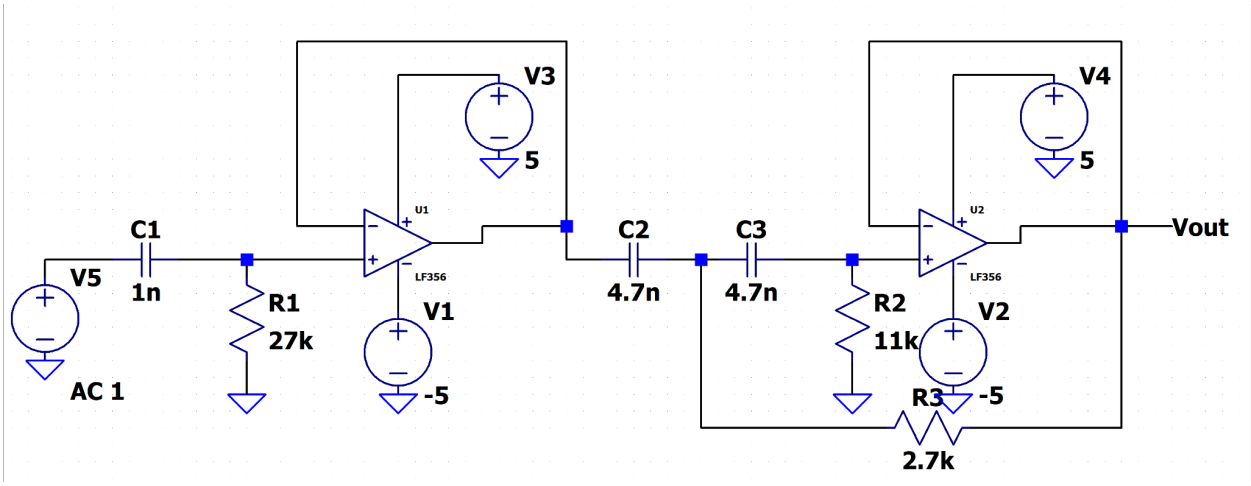
The shape of the graph is what we would expect for a bandpass filter and the center frequency is where we calculated it to be. Building the circuit we get the following frequency response.




Mid Filter Measured Frequency Response

The physical result matches our simulation.

The last filter in our graphic equalizer, is the high-pass (treble) filter. We designed the highpass filter to have a cutoff frequency of 6kHz. We chose 6kHz as the cutoff frequency as that is the bottom of the treble frequency range. Using the design tool provided to us we got the following circuit.

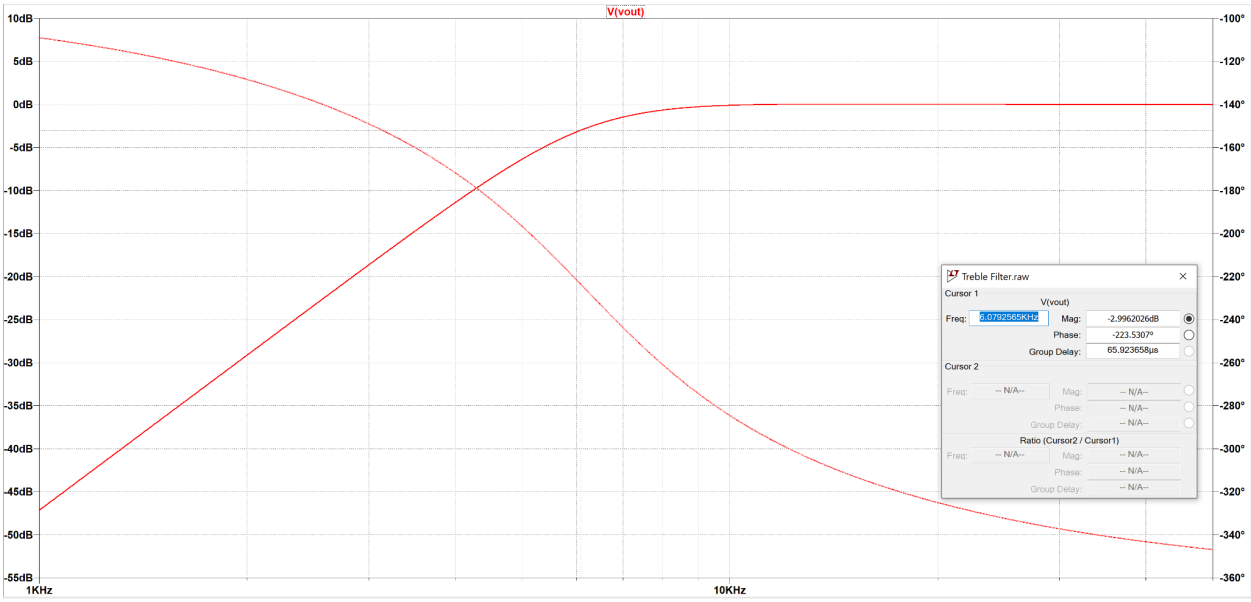



High-Pass Filter Schematic

The above circuit has the following frequency response.

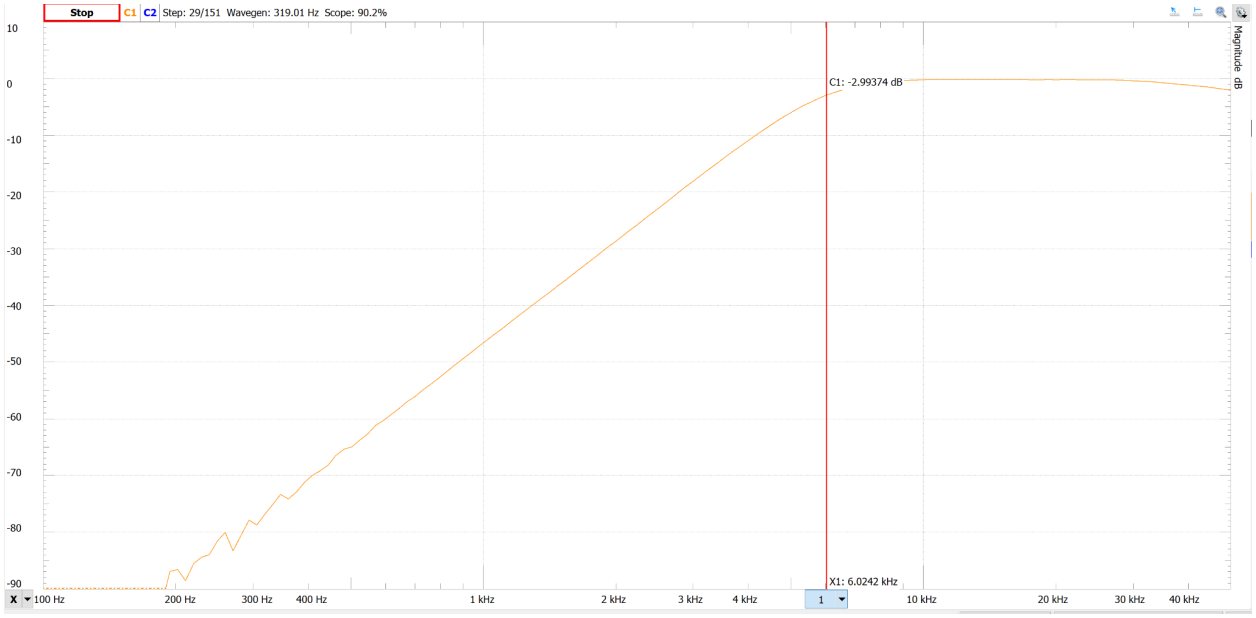
$$H(j\omega) = \frac{(27k\Omega)}{\frac{1}{j\omega(1nF)} + 27k\Omega} * \frac{2.7k\Omega * 11k\Omega}{(\frac{1}{j\omega(4.7\ nF)}) + 2.7k\Omega * (\frac{1}{j\omega(4.7\ nF)}) + (2.7k\Omega * 11k\Omega)}$$

Simulating the above circuit we get the following frequency response.




High-Pass Simulated Frequency Response

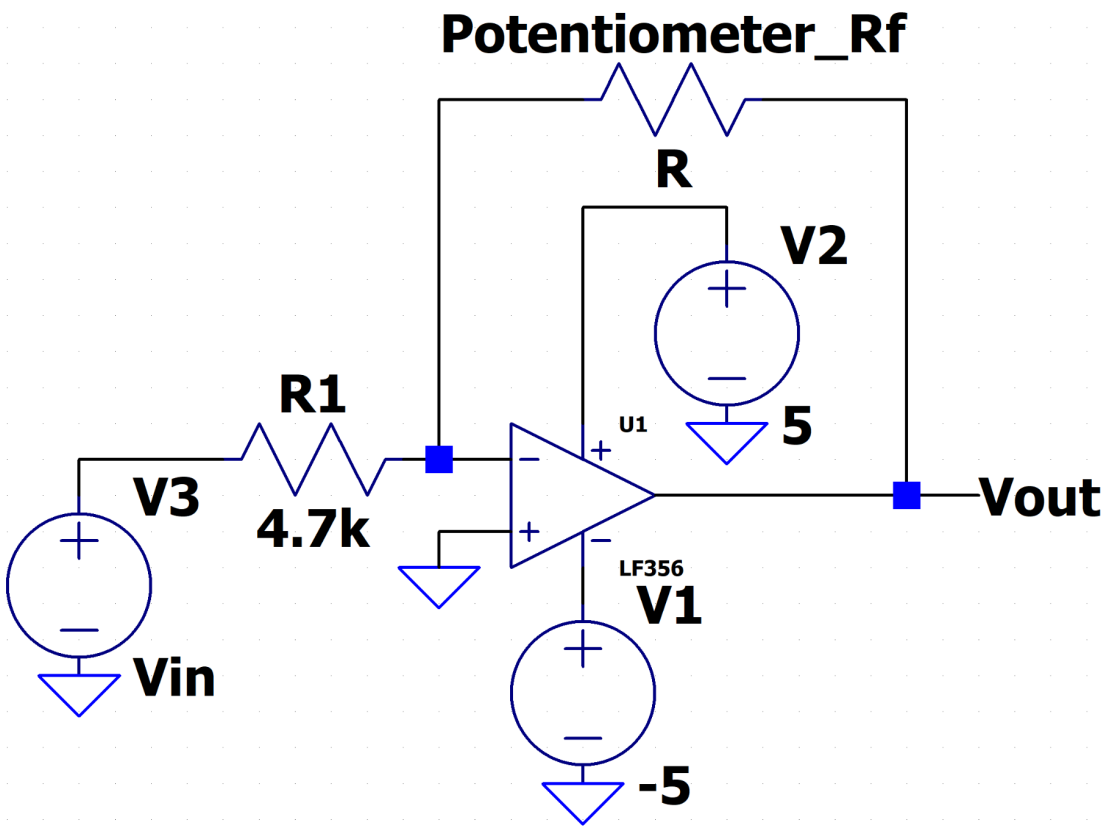
The shape matches what we would expect and our cutoff frequency is what we designed it to be. Building the above circuit we get the following frequency response.




High-Pass Measured Frequency Response

Our physical response matches our simulated response.

To achieve the variable gain with potentiometers, we came up with the idea of using an inverting amplifier where Rf is a potentiometer. The below is the schematic of our design.

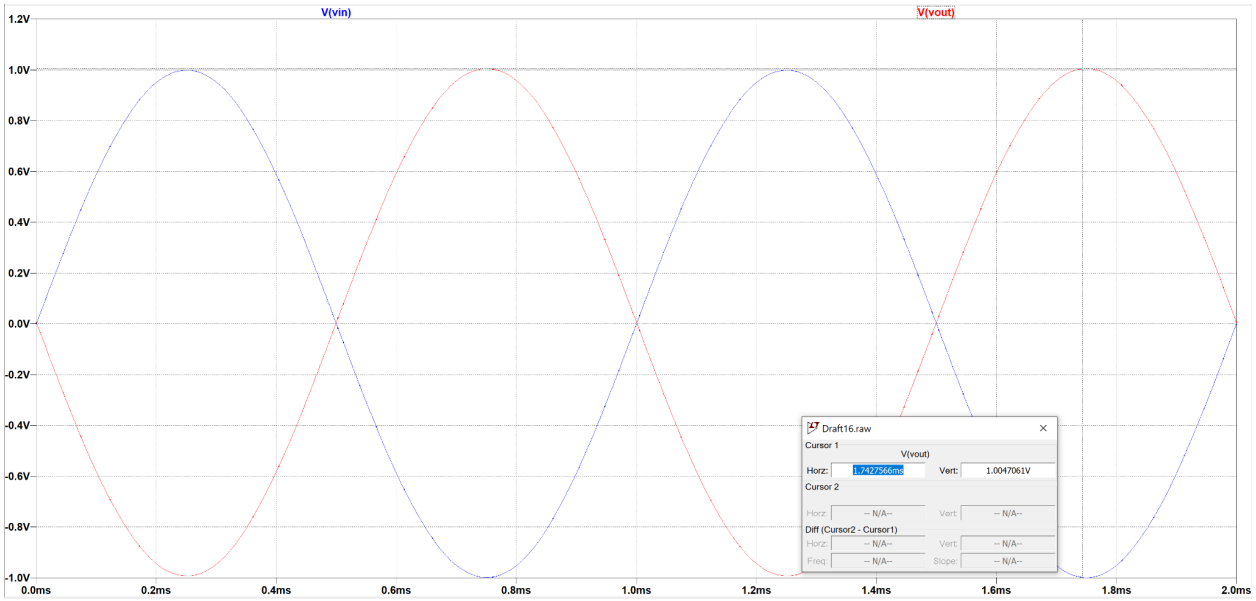



Variable Gain Amplifier Schematic

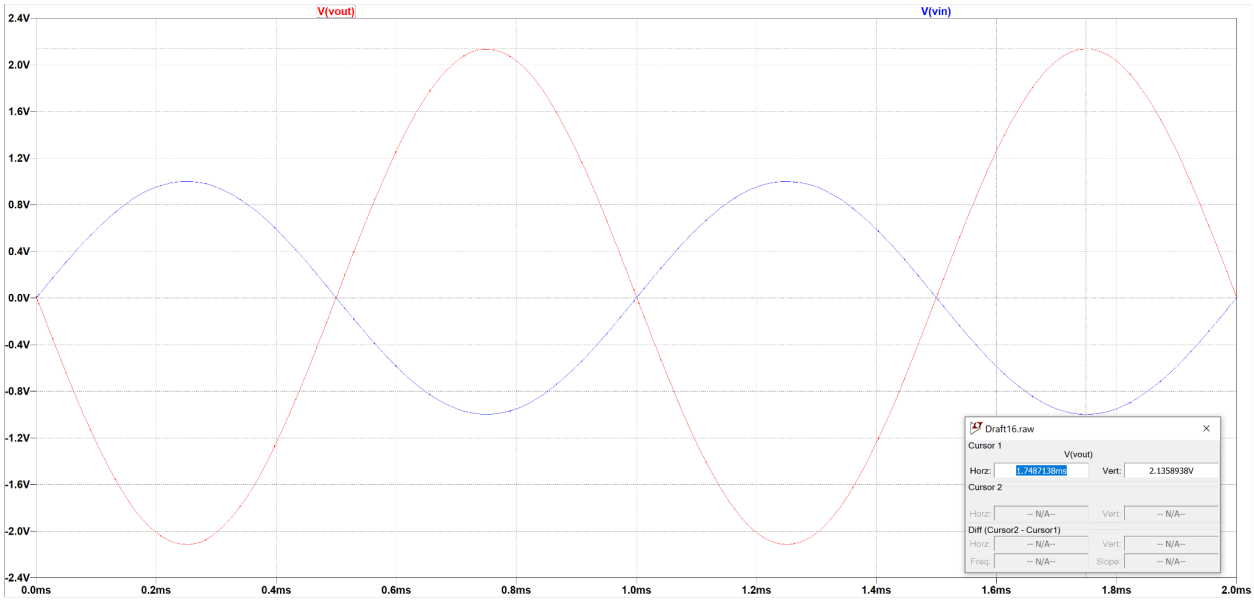
The transfer function for inverting amplifiers is written below.

$$H = \frac{Vout}{Vin} = -\frac{Rf}{Rin}$$

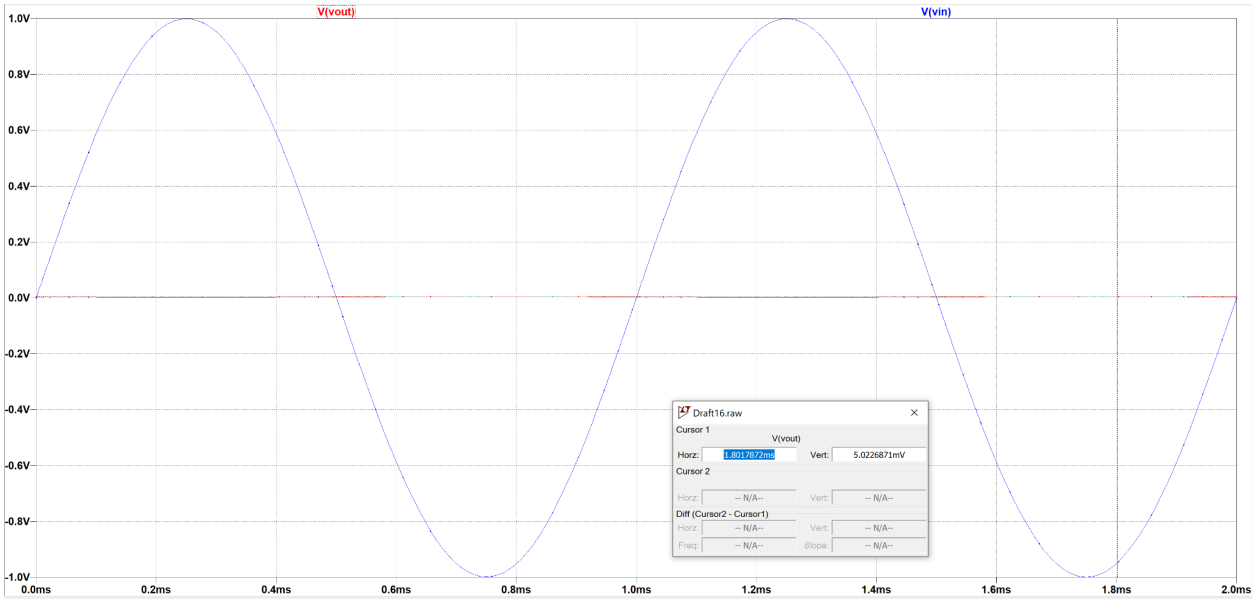
In our design the potentiometer acts as Rf. We wanted Rf to be the potentiometer so when you twist the knob the gain is proportional to where the slider is instead of exponential. We think this gives the user a better feel for how much they want to boost or cut the signal. We chose Rin to have a value of 4.7kΩ because we have potentiometers that range from 0 to 10kΩ and you should use resistors that are at least 1kΩ when working with op amps. This gives a range of linear gain from 0 to about 2. In decibels the max gain is about 6dB. The minimum gain in dB is hard to determine from just the circuit as you can't convert a linear gain of 0 to dB. We know that gain will be a large negative number but that depends on how much leakage current there is. Below are the simulations for the variable gain amplifier in a neutral position (Rf = 4.7kΩ), the maximum position (Rf = 10kΩ), and the minimum position (Rf = 10Ω). The input is 1V sine wave at 1kHz.




Variable Gain Amp Simulated Neutral Position

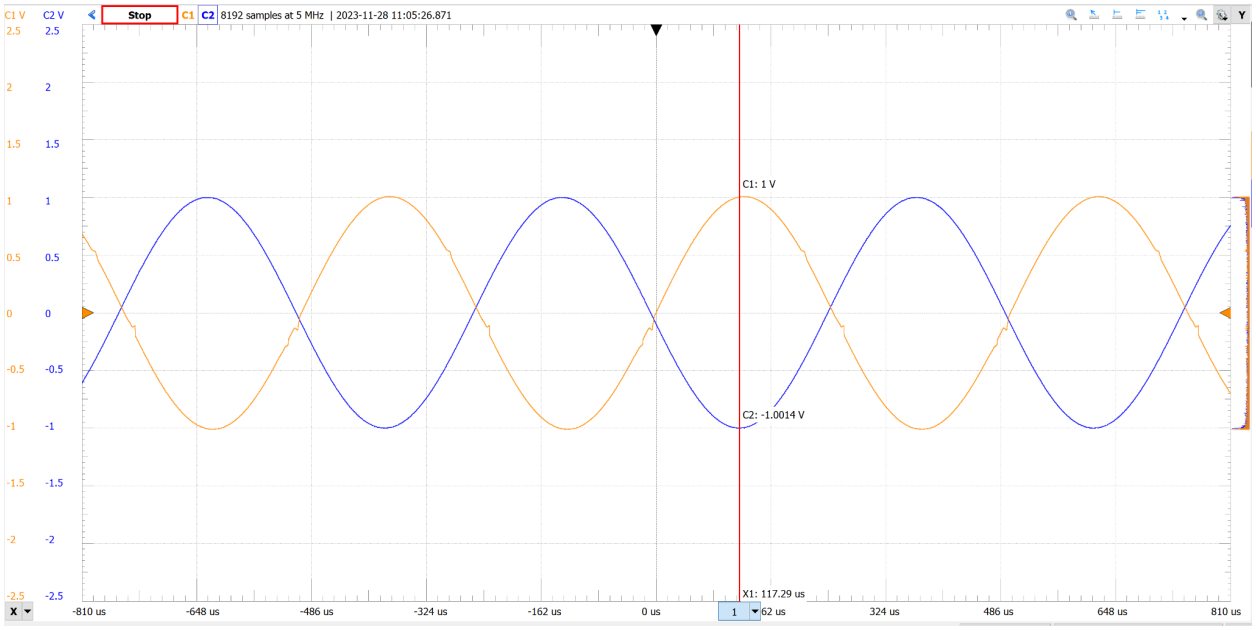



Variable Gain Amp Simulated Maximum Position

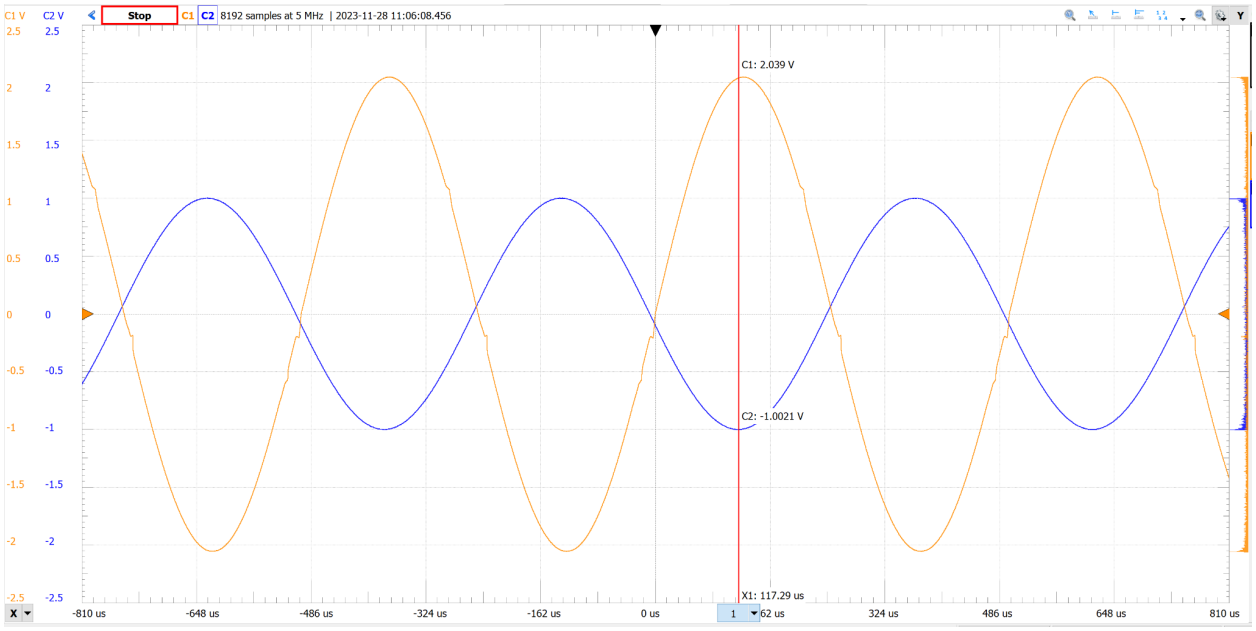



Variable Gain Amp Simulated Minimum Position

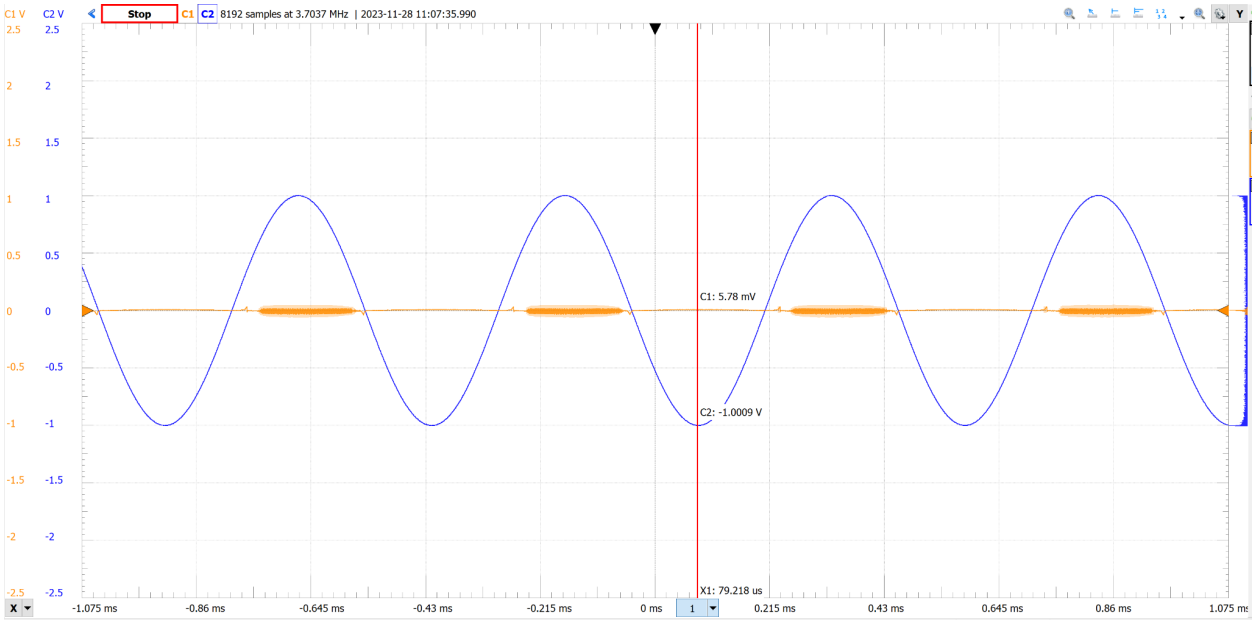
These simulations match what we would expect. In the neutral position the magnitude of the output voltage is the same as the input voltage. In max, the magnitude of the output voltage is roughly twice the input voltage. In min, the magnitude of the output voltage is close to zero. After building the circuit, we can scope to see if the simulation matches the real world. Below are the same positions as the simulation. The blue channel is the input sine wave and the yellow is the output.




Variable Gain Amp Measured Neutral Position




Variable Gain Amp Measured Maximum Position



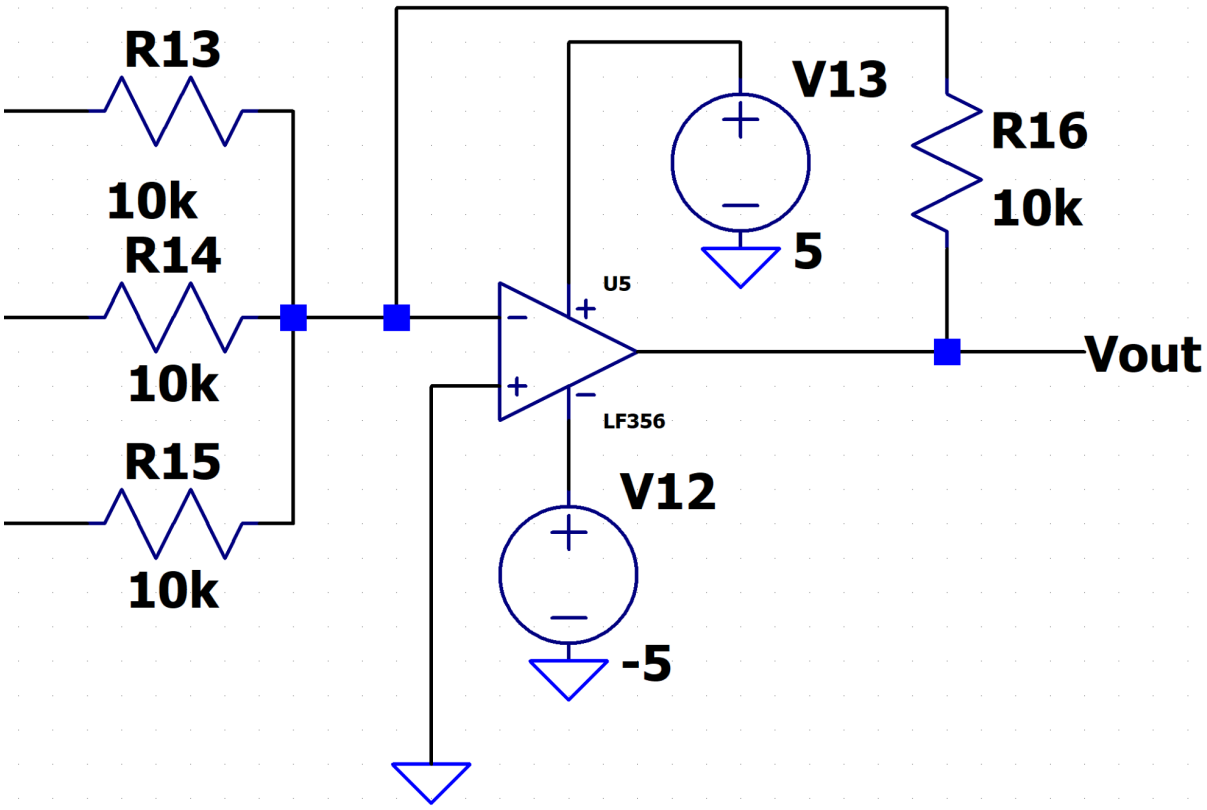

Variable Gain Amp Measured Minimum Position

From the above scopes we can see that our physical circuit matches our simulation.

The last subsystem of our graphic equalizer is a summing amplifier. We used a summing amplifier to add the filtered signals back together so that we could easily feed the output into the amplifier. We know the transfer function for a summing amplifier.

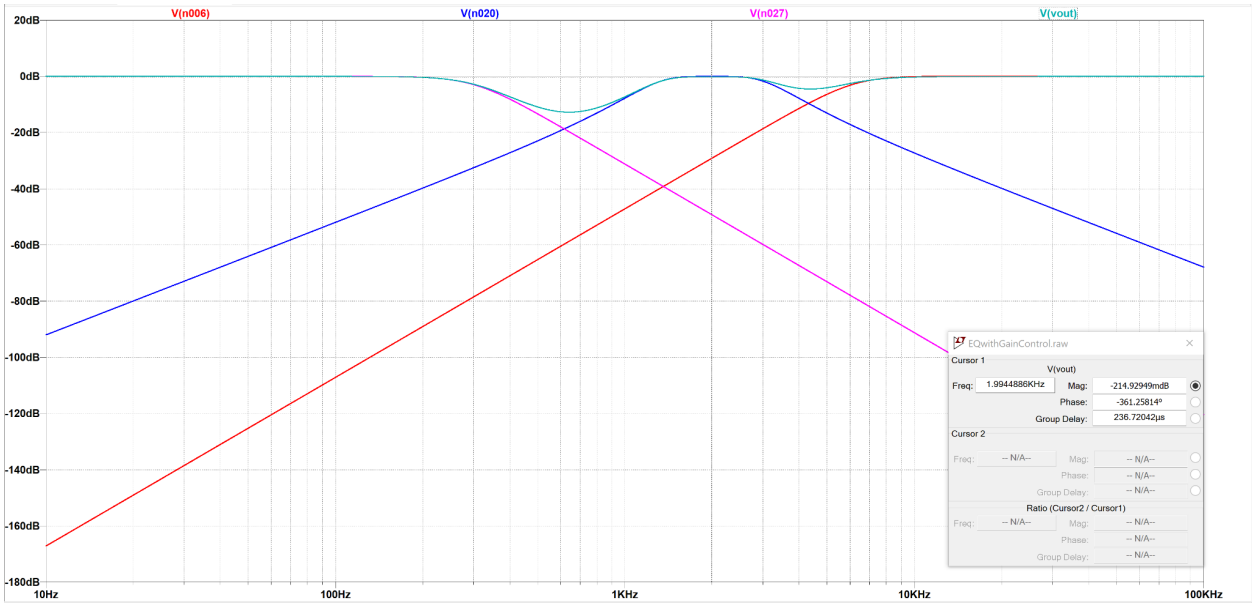
$$-Vout = \frac{Rf}{Rin}(V1 + V2 + V3)$$

We chose an inverting summing amp, because the variable gain amps are also inverting. Since the input signal gets inverted twice the output of the whole circuit isn't inverted. We made Rf equal to Rin so that Vout was just equal to the sum of the inputs. We chose Rf to be 10k□ but for this part you can really chose any value you like as long as Rf equals Rin. Below is the schematic for our summing amplifier.



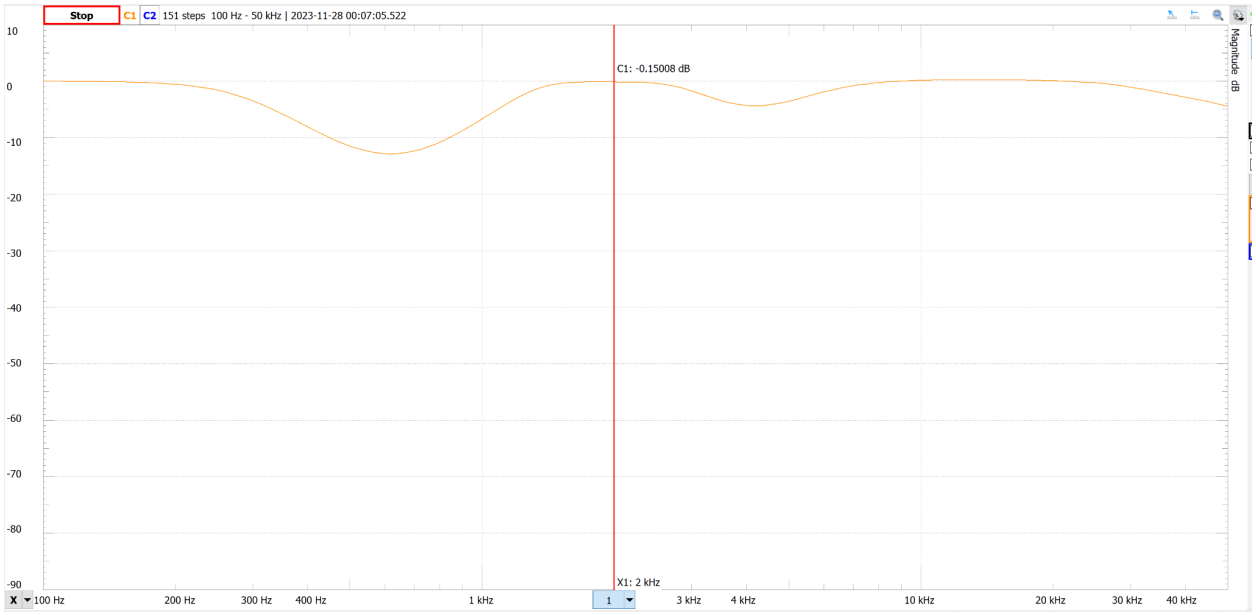

Summing Op-Amp Schematic

This circuit adds the bass, mid, and treble outputs together. Simulating the circuit, attached to the filters we can see how the amp adds the signals together.




Simulated Frequency Response of Equalizer

Building the above we can see the physical result below.

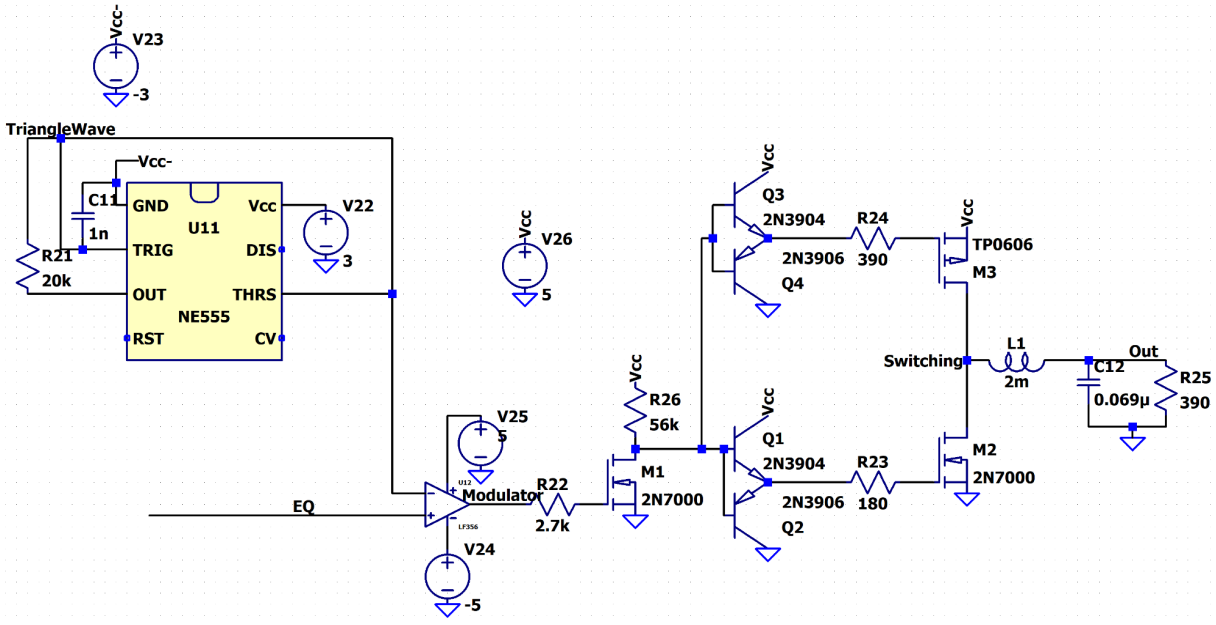



Measured Frequency Response of Equalizer

Our physical result matches our simulation. Next we will talk about our class-D amplifier.

## Class D Amplifier

The class-D amplifier subsystem consists of a modulator stage, a switching output stage, and a low-pass filter. The schematic for our amplifier is shown below.




Class D Op-Amp Schematic

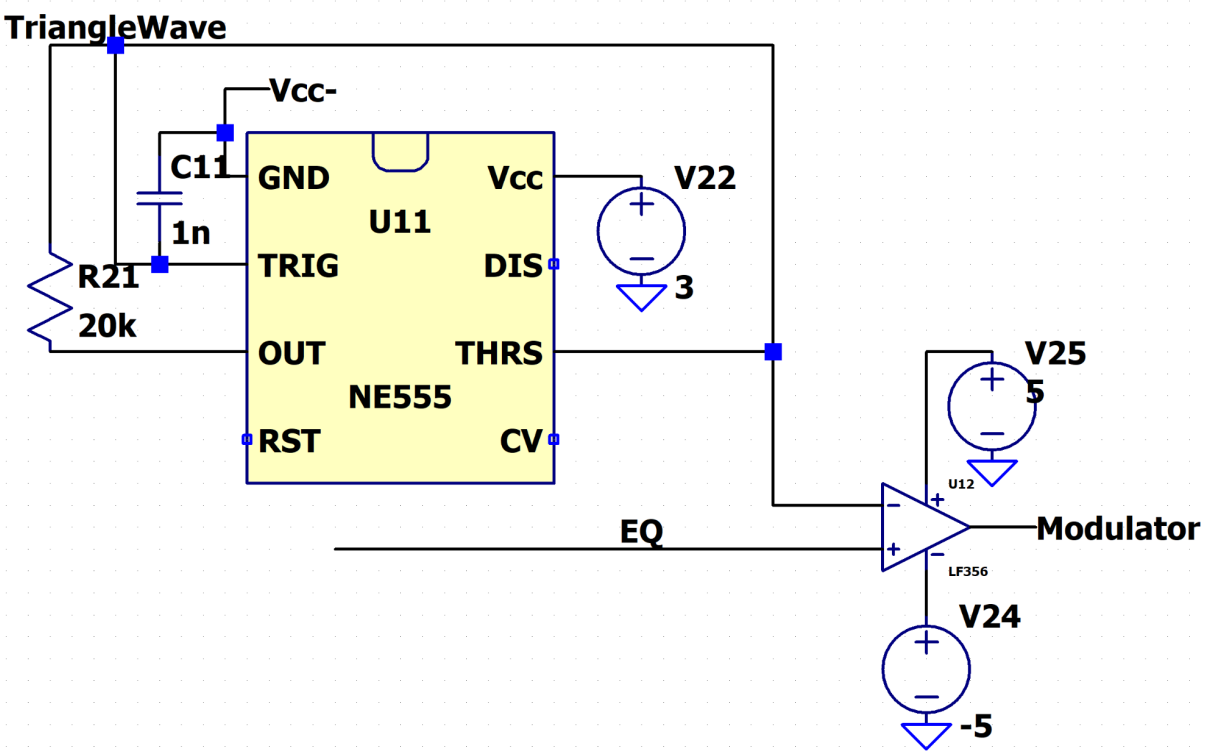
The audio signal coming from the equalizer goes through modulation so a PWM signal can be fed into the switching output stage. To modulate the audio signal we compare it to a triangle wave. The comparison is done by an op amp, which produces a varying PWM based on when the voltage of the audio signal is greater than the triangle wave. The triangle-wave generation is done using the TLC555 chip to charge and discharge a capacitor. While a charging and

discharging capacitor is not a perfect triangle it is very close as long as the charges and discharges for the same amount of time. The frequency of the triangle wave is determined by the time constant of the capacitor and resistors. This design allows us to easily calculate the frequency of the wave and change it just as easily by switching out either the capacitor or resistor.

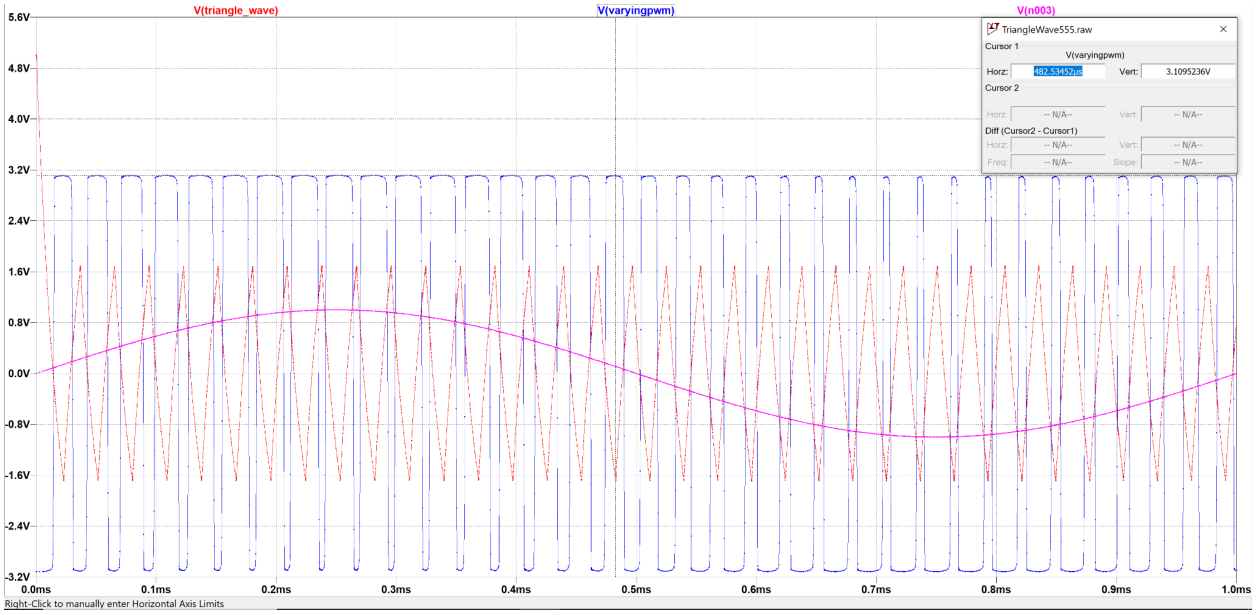
$$\tau = RC = (20k\Omega)(1\,nF) = 2.5\mu s$$

$$f = \frac{1}{\tau} = \frac{1}{2.5\mu s} = 50kHz$$

Our modulator schematic is pictured below.

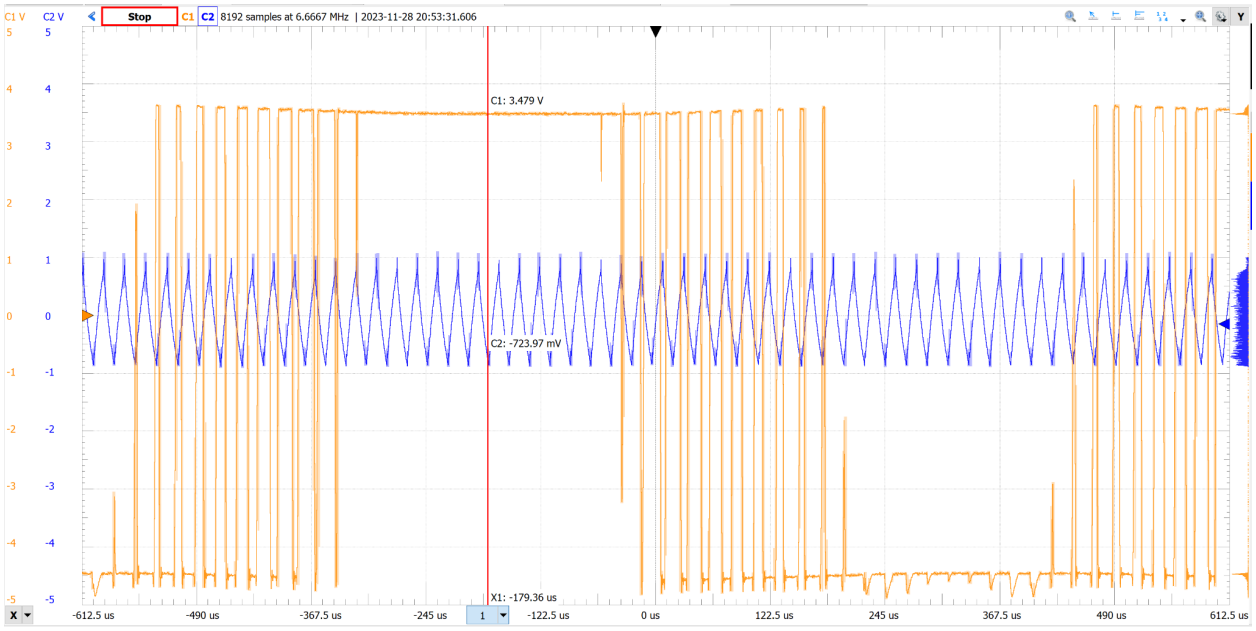



Our simulated results for the modulator are described below. We can see that the duty cycle near where the audio signal from the equalizer is above the triangle wave forms is when it is at its highest (close to 100% duty cycle), while when the audio signal is below the audio signal is when it is at its lowest (close to 0% duty cycle).




Simulated Modulator Output with 1kHz Sine Wave

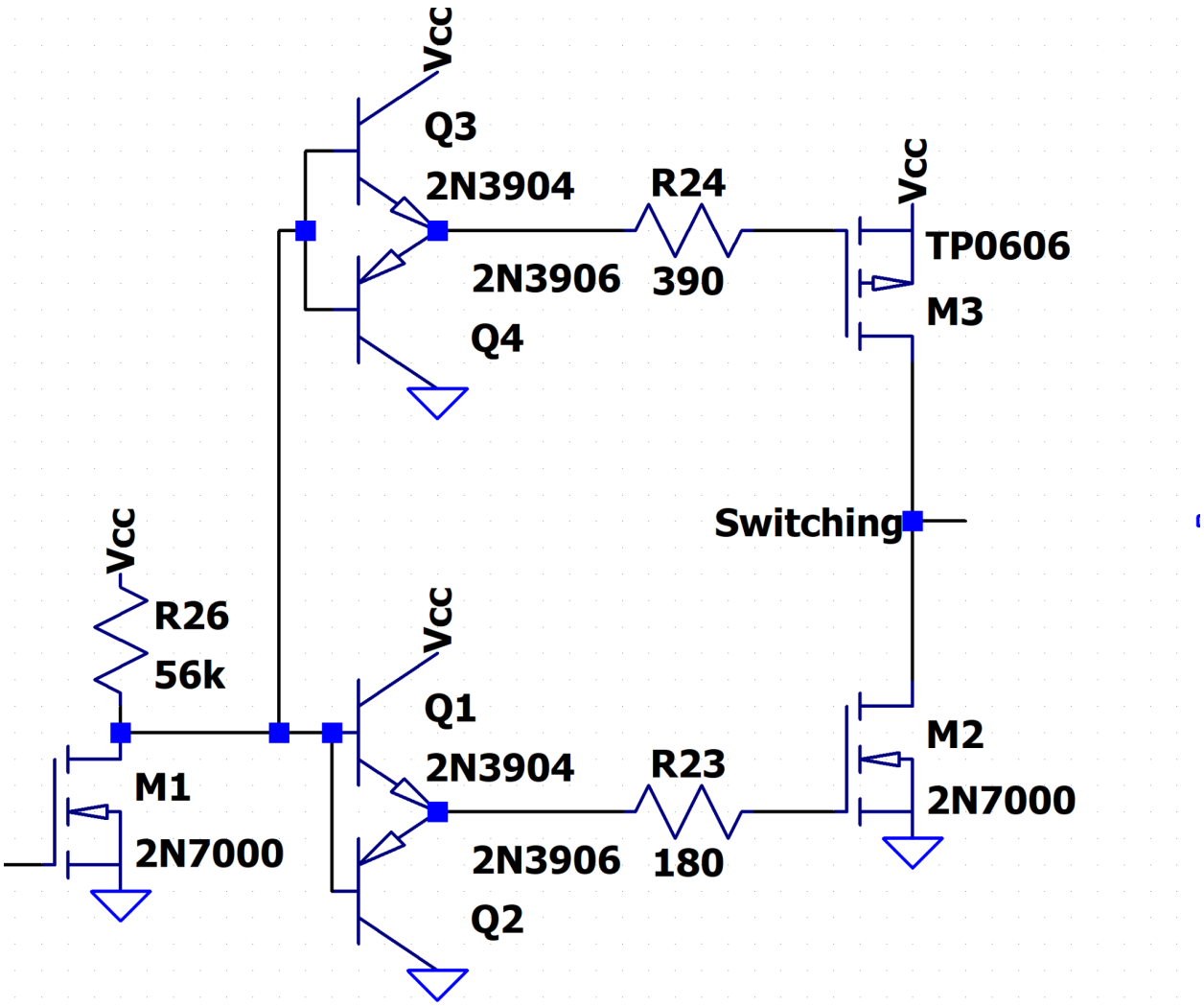
Our validation of this is shown below. We can see that the validate output of the modulator also has the same features as discussed for the LTspice counterpart




Measured Modulator Output with 1kHz Sine Wave

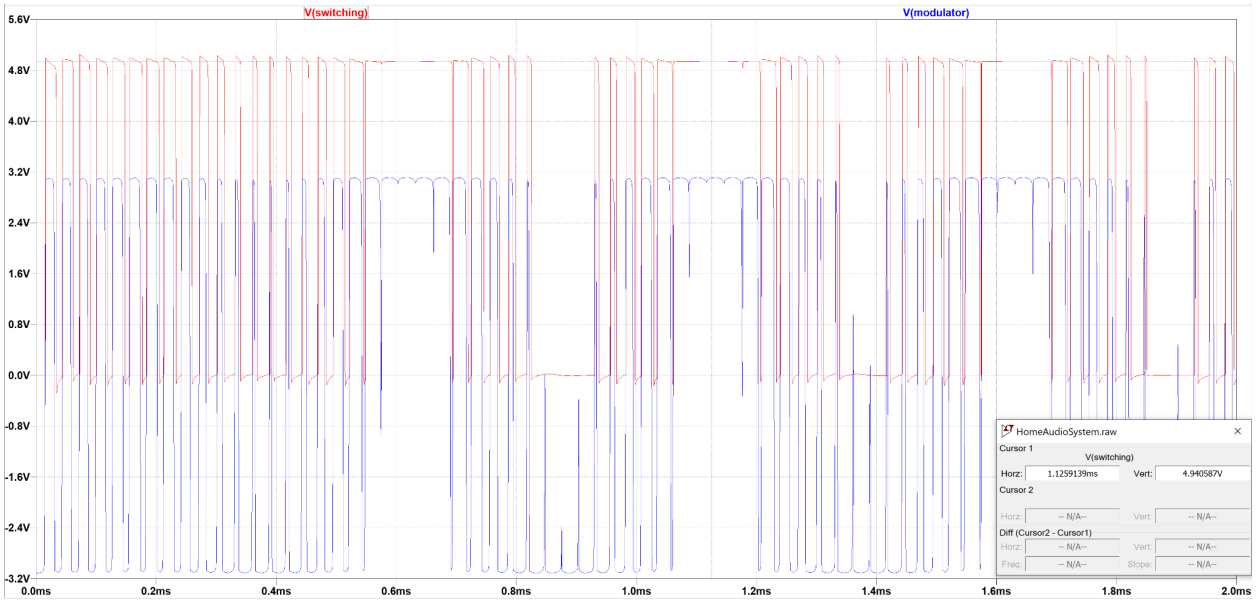
The measure output for the modulator matches the simulated. We aren't able to show the input sine wave since our AD2 only has two channels for scoping.

The output of the modulator circuit is then used as the input for the switching output stage. For this part, we opt-ed for the BJT push-pull configuration due to the current gain dependent on the small-signal current gain $\beta$. The push-pull configuration will push amplified current towards the output stage, and also acts as a sink for turn-off of the mosfet to ground. The circuit is depicted below.



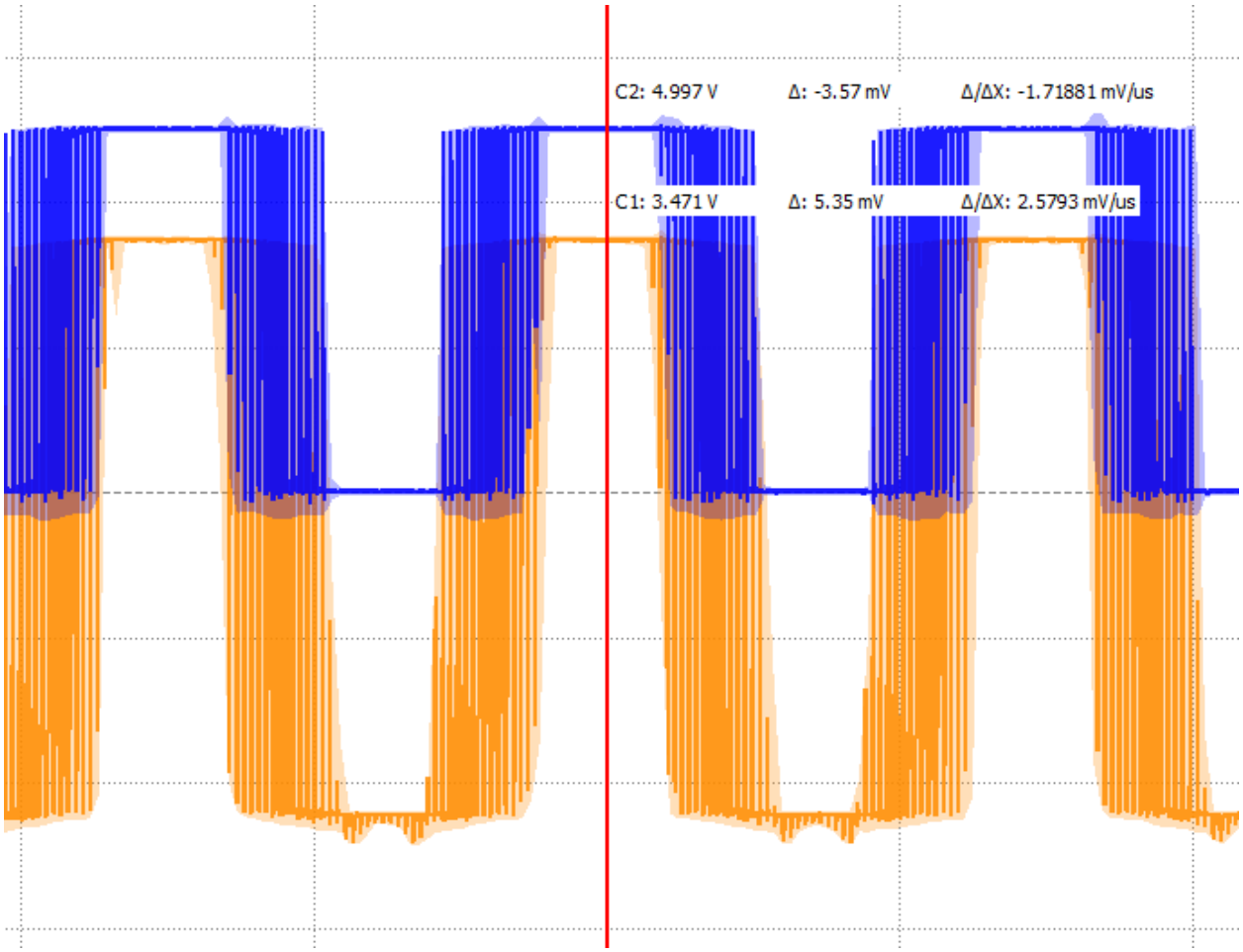

The Gate Driver and Output stage

We have two BJT push-pulls for the PMOS and NMOS of the output stage. We need an inverter before the BJT configurations due to how the output stage works. When the PWM signal is high, we want the PMOS mosfet to be on while the NMOS is off. When the PWM signal is low, we want the PMOS mosfet to be on while the NMOS is on. Although we can operate the output stage without the NMOS inverter, as the audio signal will only be inverted from the original, we wanted to add some sort of delay between the turn on and turn off of the output stage to avoid massive shoot-through current, and the NMOS inverter can help with this. The current gain from the BJT push-pull is then fed into the output stage to drive the switching. Below is the simulated output compared to the PWM waveform by passing a 50 kHz square wave into the class D amplifier (50% duty cycle)



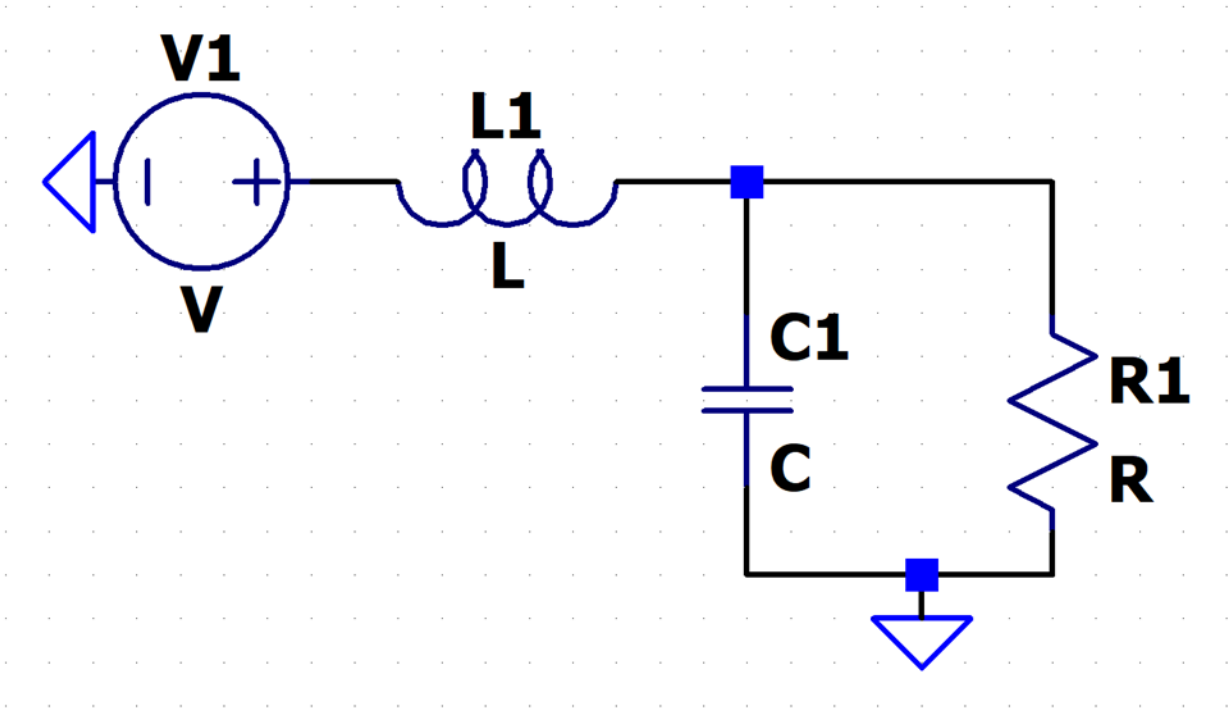

Simulated Output of Switching Output Stage

Our measured results are shown below.




Measured Output of Switching Output Stage

Our design for the filter stage of the class D is a standard LC low-pass filter with a load resistance pictured below.




LC Low Pass Filter Schematic

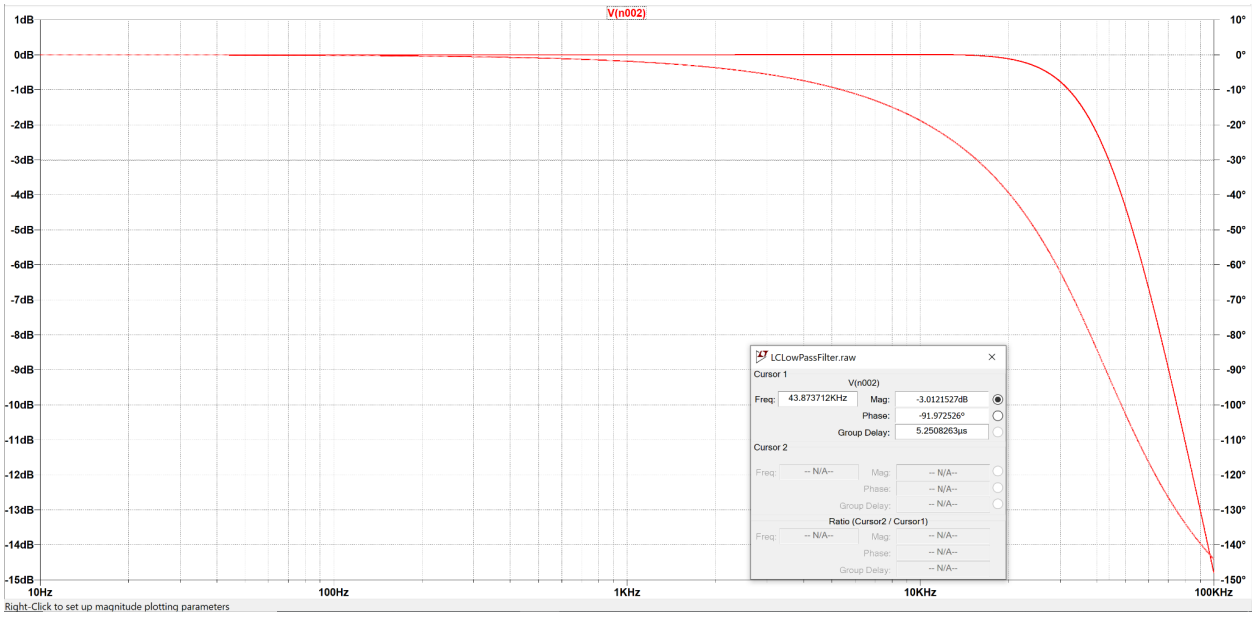
Next we derived equations for our LC lowpass-filter, and picked the cutoff frequency.

$$f_c = \frac{1}{2\pi\sqrt{LC}}$$

$$L = \frac{R_L \times \sqrt{2}}{2\pi f_c}$$

$$C = \frac{1}{2\pi f_c \times R_L \times \sqrt{2}}$$

The cutoff frequency had to be in the range of 40-60kHz to preserve the audio frequencies. Next we picked components from our parts kit to achieve the correct cutoff frequency. In our parts kit we only have 1mH and 100mH inductors. So to get in the 40-60kHz range we need a value of 2mH. From this value we can calculate the capacitance and the closet value we can achieve from our parts kit is 6.9nF. From the values we picked, and using a 390Ω load resistor to match the specification, we get the following frequency response.




LC Low Pass Filter Simulated Frequency Response

Using the values we have in our kit we got the below frequency response.

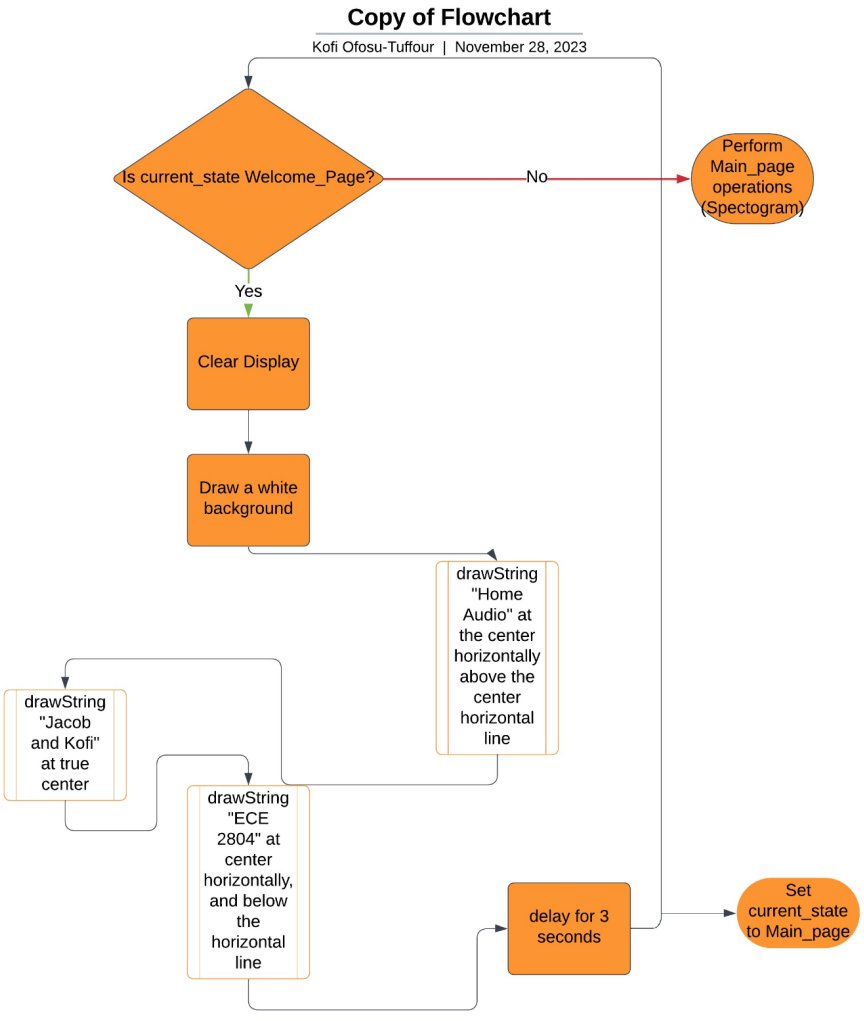

LC Low Pass Filter Measured Frequency Response

Our simulation matches the frequency response of our physical circuit.

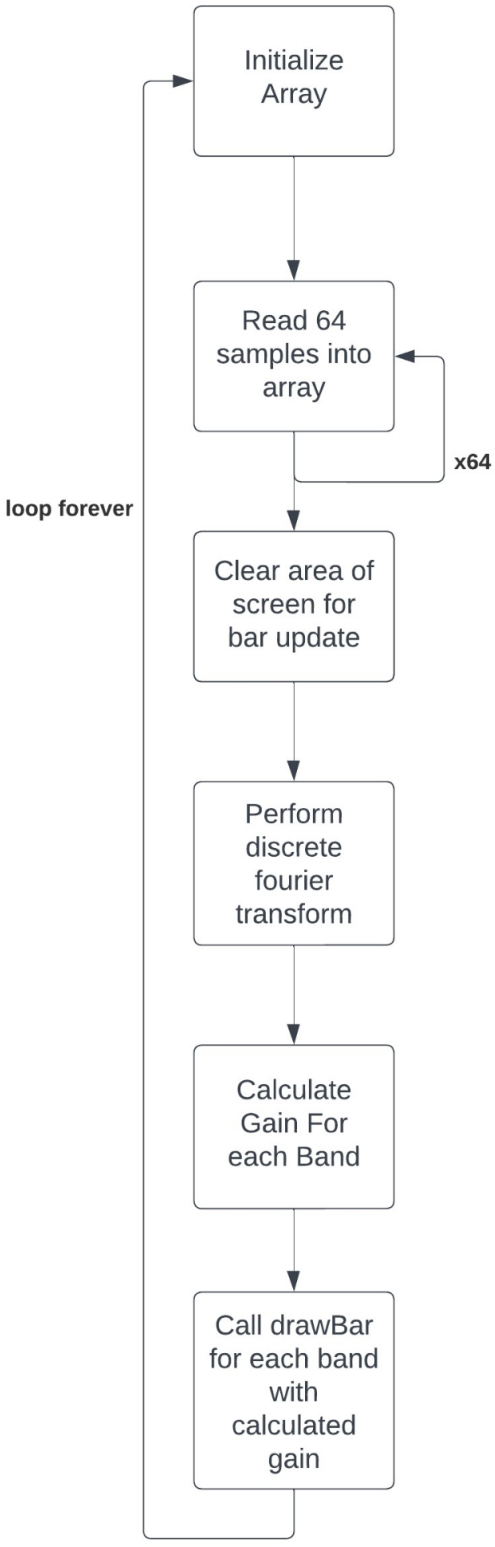
Passing a 1 kHz sinusoidal signal, 1.6 peak-to-peak voltage, as the input for the class D our LTspice simulation below shows how the amplifier amplifies the signal to with a peak-to-peak voltage of around 5V.

## Spectrogram

The spectrogram is a program run on an Arduino Uno V3, using a 128x64 LCD display communicating through i2c. This communication is already facilitated by a library that is not restricted for use. The spectrogram program first displays a welcome screen to the user before entering the spectrogram screen. The spectrogram loop of the program takes 64 samples of the amplified audio signal and decomposes the audio signal into its frequency components using discrete fourier transform, and displays a bar for each band showing whether the signal is currently gained, attenuated, or at 0 dB. To validate this approach, we passed a 500 Hz sinusoidal wave to test how the bass band responds to user-input from the equalizer.



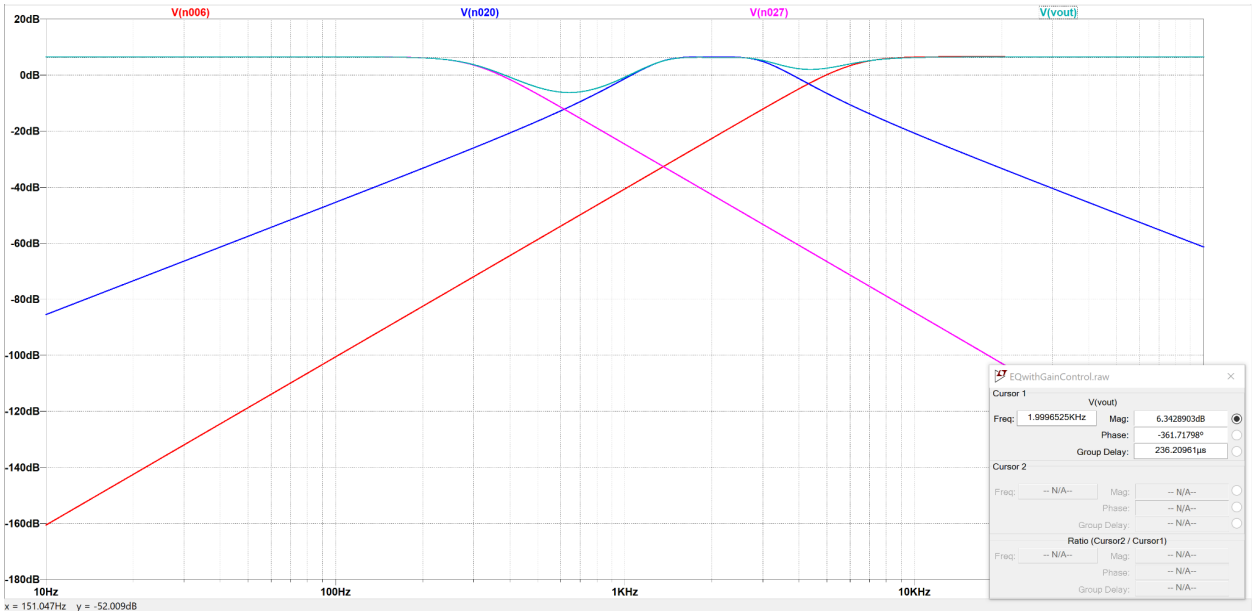

Welcome Page Loop



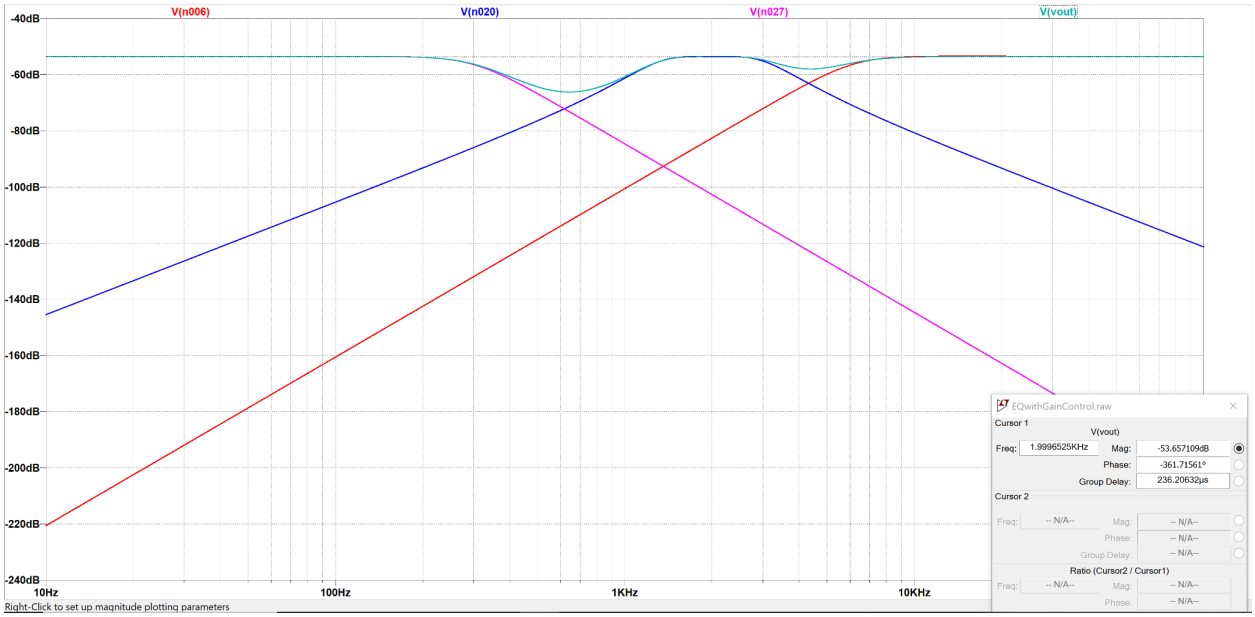

Spectrogram Loop

## Validation

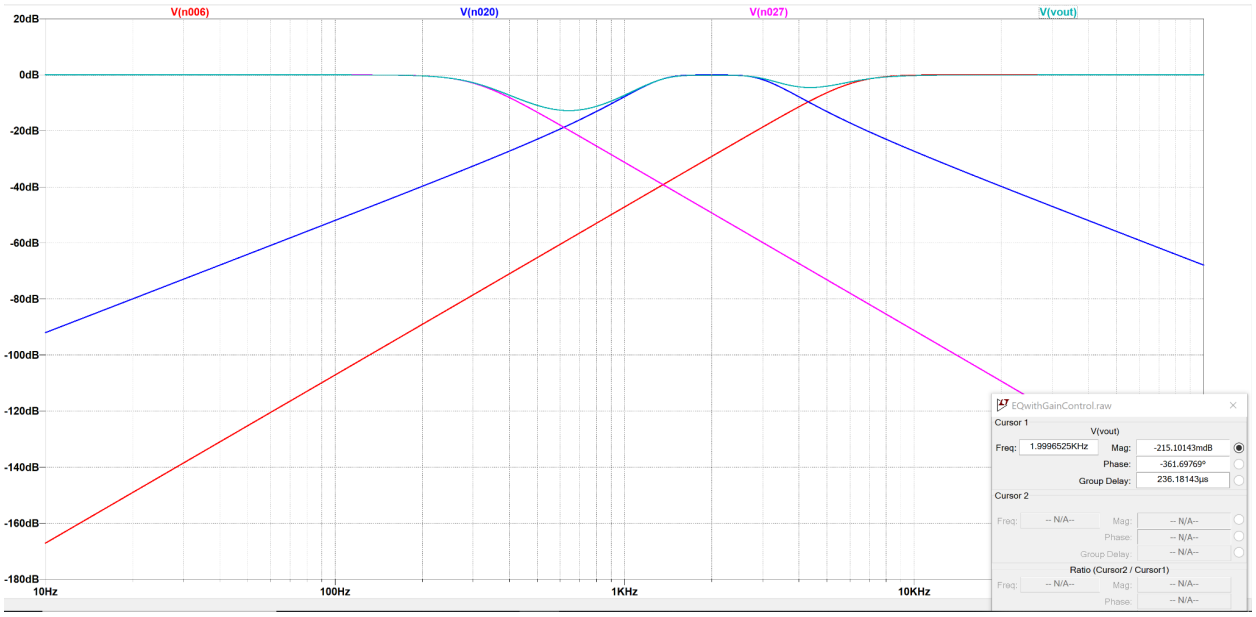
We will now go over how we validated our designs showing how they meet the specifications for any we haven't already gone over. First we show that our filters can all achieve gains greater than 0 dB, less than 0 dB and 0dB. When going over the design of our filters we showed the simulated and measured frequency responses. Below are the simulations and measured results of our graphic equalizer, showing that all three filters are capable of achieving the different gains.



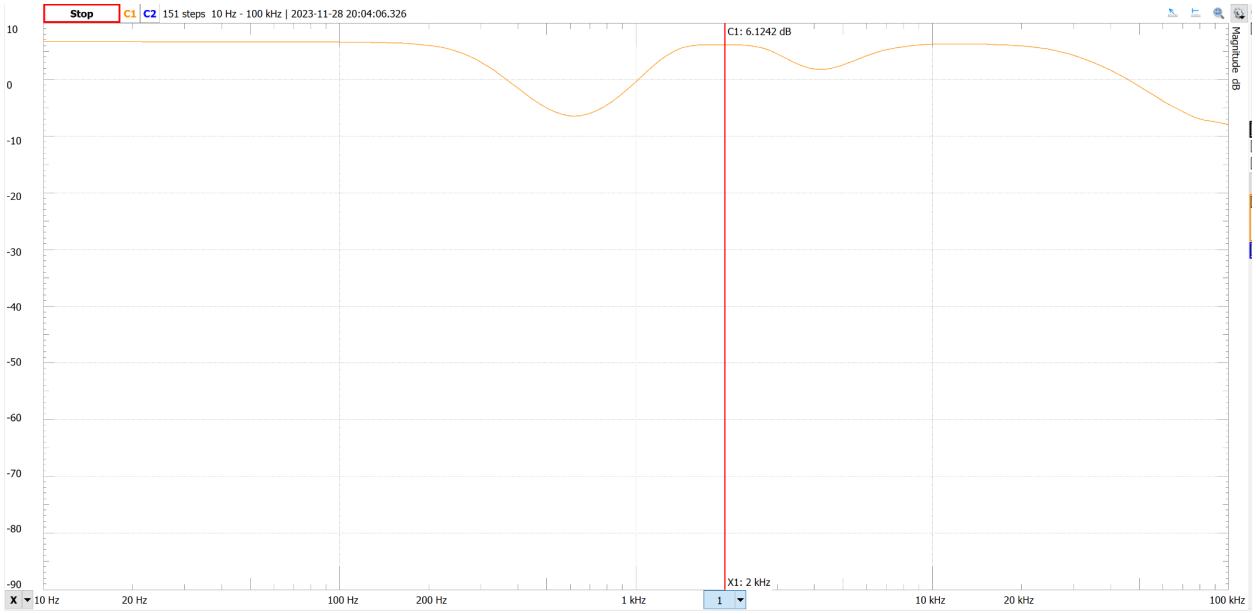

EQ Simulated Frequency Response Max Gain



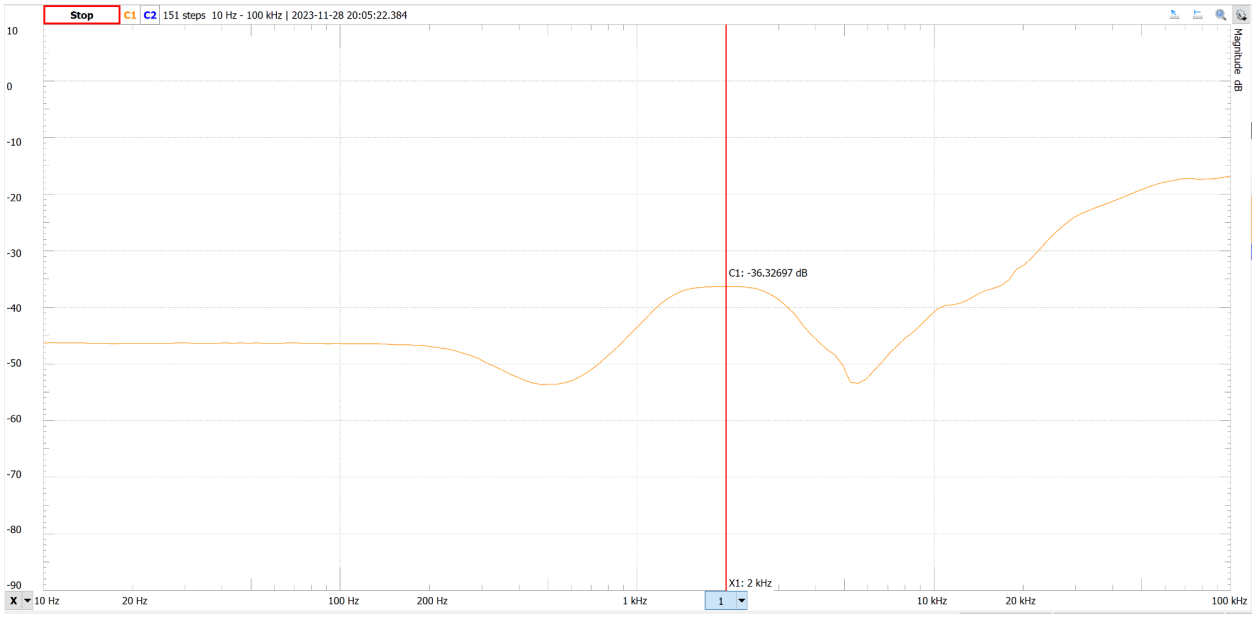

EQ Simulated Frequency Response Min Gain



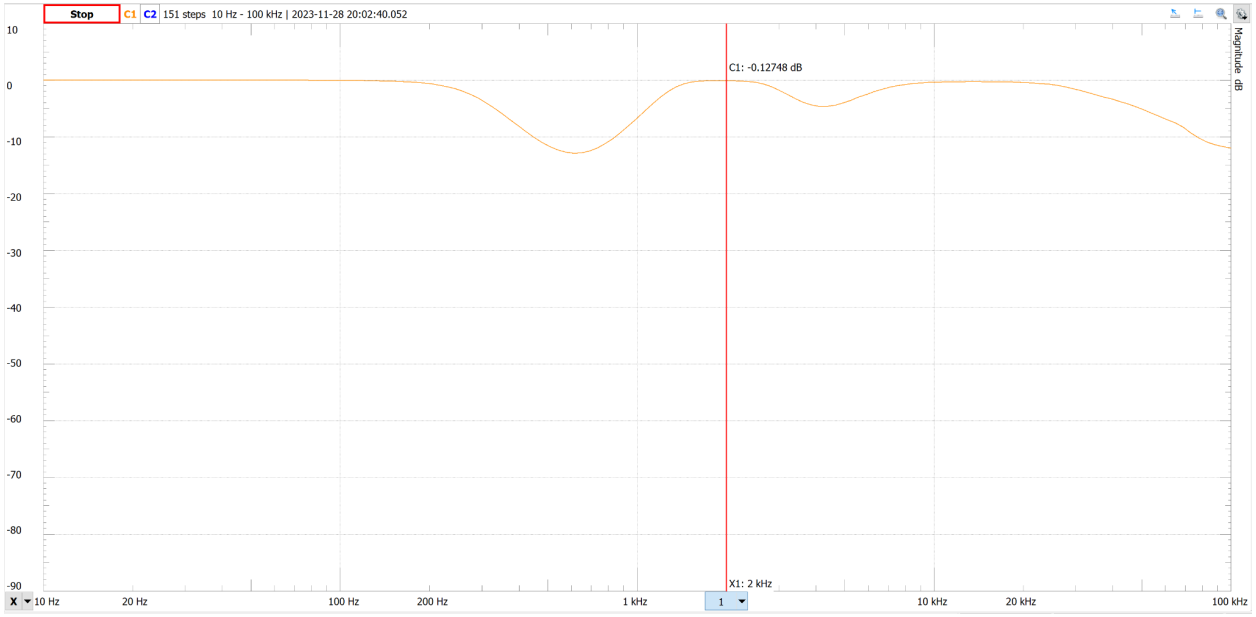

EQ Simulated Frequency Response No Gain




EQ Measured Frequency Response Max Gain



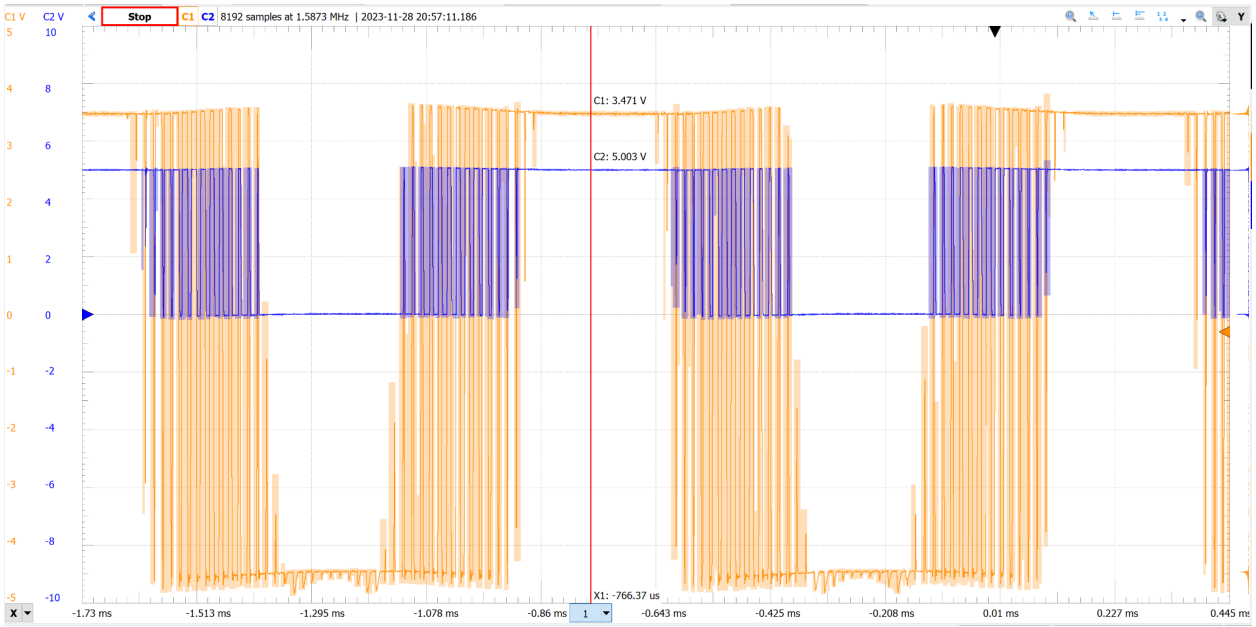

EQ Measured Frequency Response Min Gain




EQ Measured Frequency Response No Gain

From the graphs above you can see that our circuit meets the design requirements.

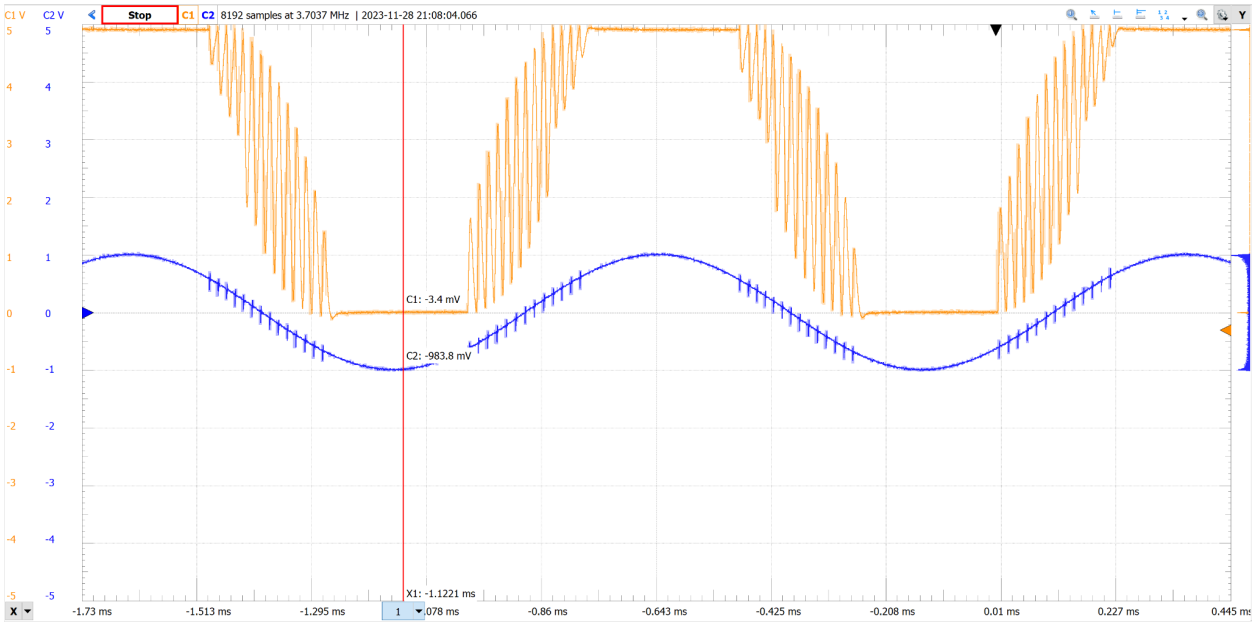
Below is the measured output of the switching output stage compared to the PWM waveform. The switching output is the blue waveform and the PWM is the orange waveform. They share the same shape however the output waveform for the switching stage is only positive voltages.




Measured Waveforms of PWM and Switching Output Stage

The shoot through current that we measured is pictured below. The value is in microamps which is way below the requirement of 100mA.


Below are the waveforms for the output of the class D amp and the input sine wave. You can see that the output of the amp is high when the voltage of the sine wave is larger than 0 and is low when the voltage of the sine wave is negative. You can also see how the carrier PWM modifies the sine wave.




Below is an image of the welcome page on the OLED screen. The images that show how the bars on the OLED screen change as we adjust the potentiometers are in the appendix.

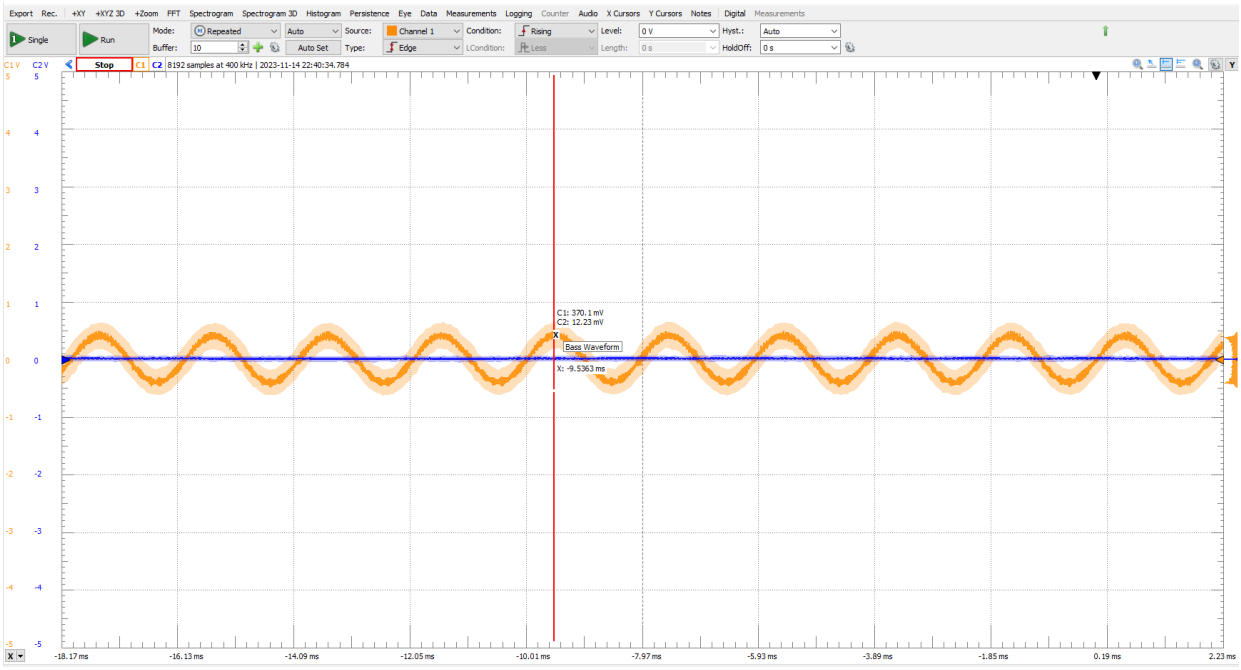
## Conclusion

Overall, we think our system turned out pretty well. We were able to meet all of the specifications and the coolest part of all we are actually able to play a song out of our circuit. Working on this project has been a really eye opening experience, it showed us how much work goes into designing, building, testing, and reworking a product. We got to experience the engineering process instead of just hearing about it. We had to go seek out information for what we didn't know how to do, and mix these new ideas with what we already knew. We experienced the best and worst of this process too. There were times we designed something, simulated it, but then when we actually built it, it didn't work. Then there were other times where we did the same process but it worked perfectly. It helped us understand what we did and didn't know, and that there is always more to learn.

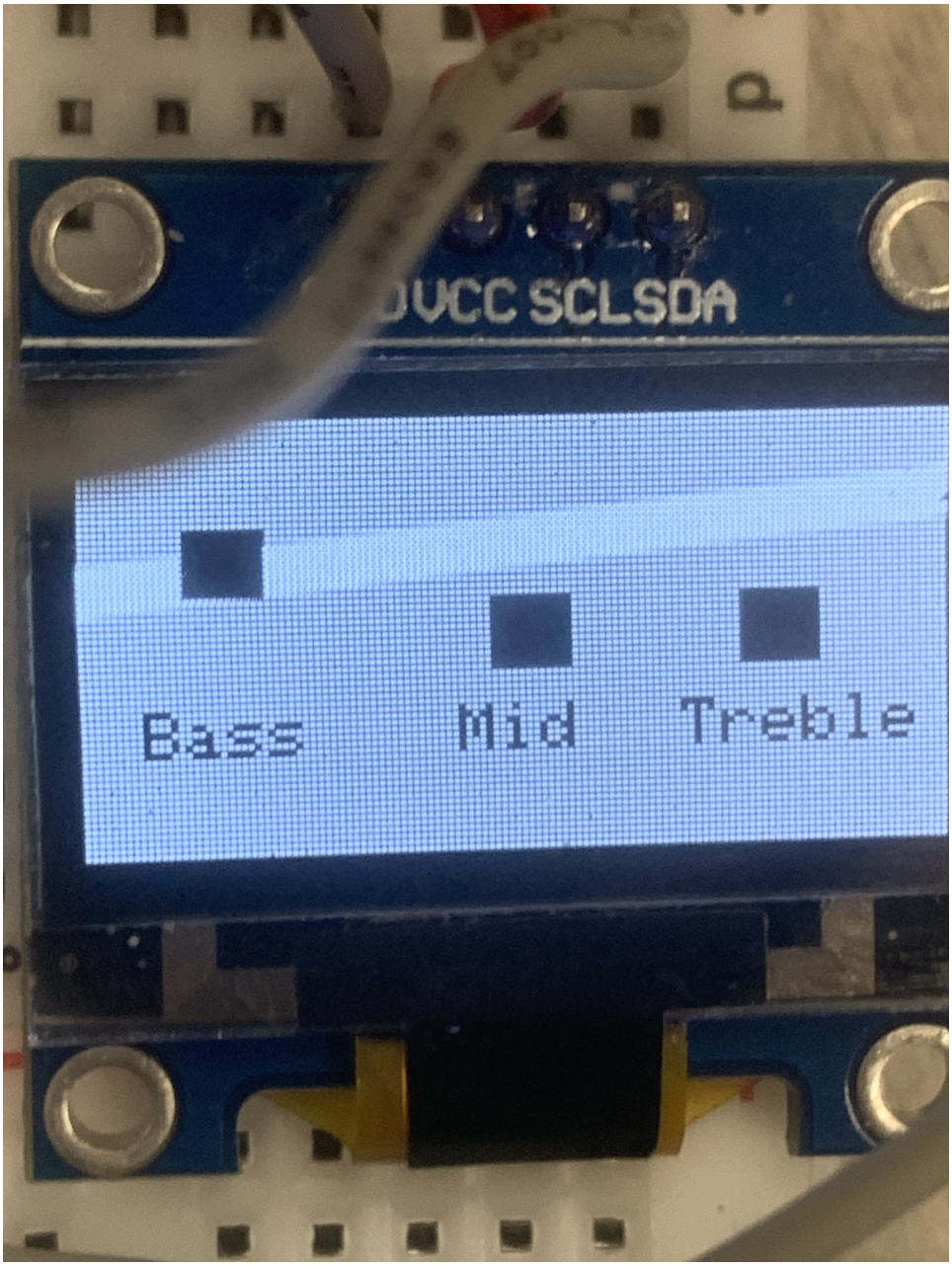
If we were to do this project again, I think we would spend more time searching for information, and going over different designs. Because, if you invest the time early on researching different designs and then picking the right one for your purposes, the simulation and building will be easier and quicker.

If we were to continue working on our project, the first thing we would work on is reducing noise. While our current design does produce an audio signal that you can manipulate, it is not the highest quality. Some of this noise could be from EMI as our graphic equalizer is built only on one board so the components are pretty close together. Another aspect we would look into improving would be how fast our switching output stage switches. Our current design is limited in how fast the carrier frequency for the switching output stage can be. If we could increase the frequency the switching stage could handle we would be able to produce a better output waveform.

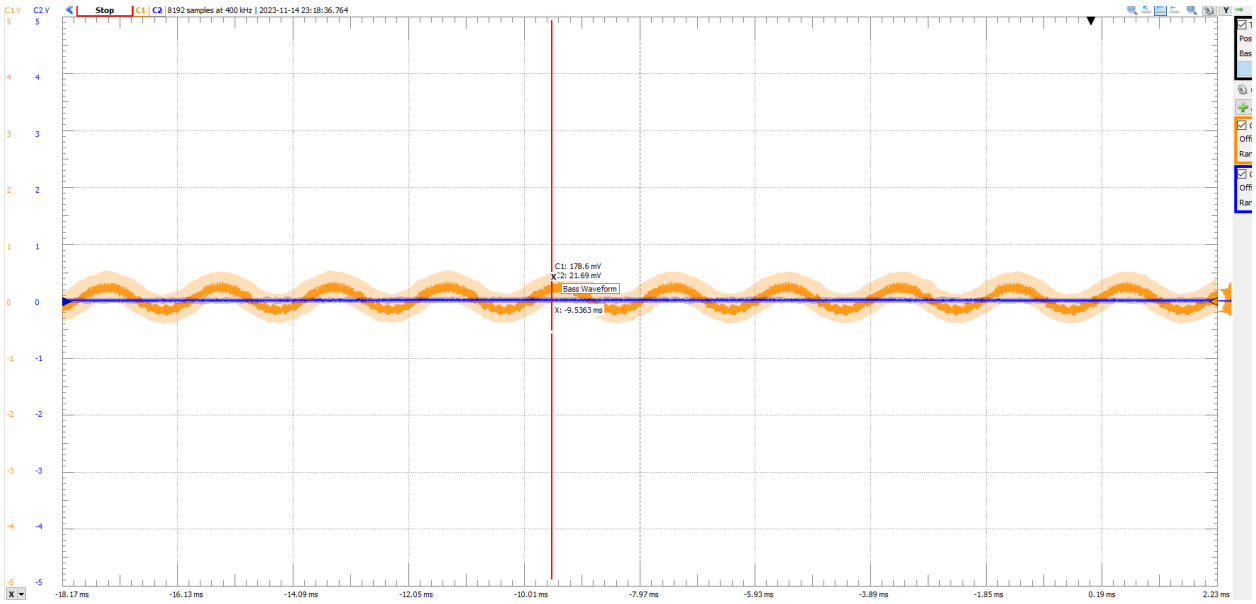
# Appendix



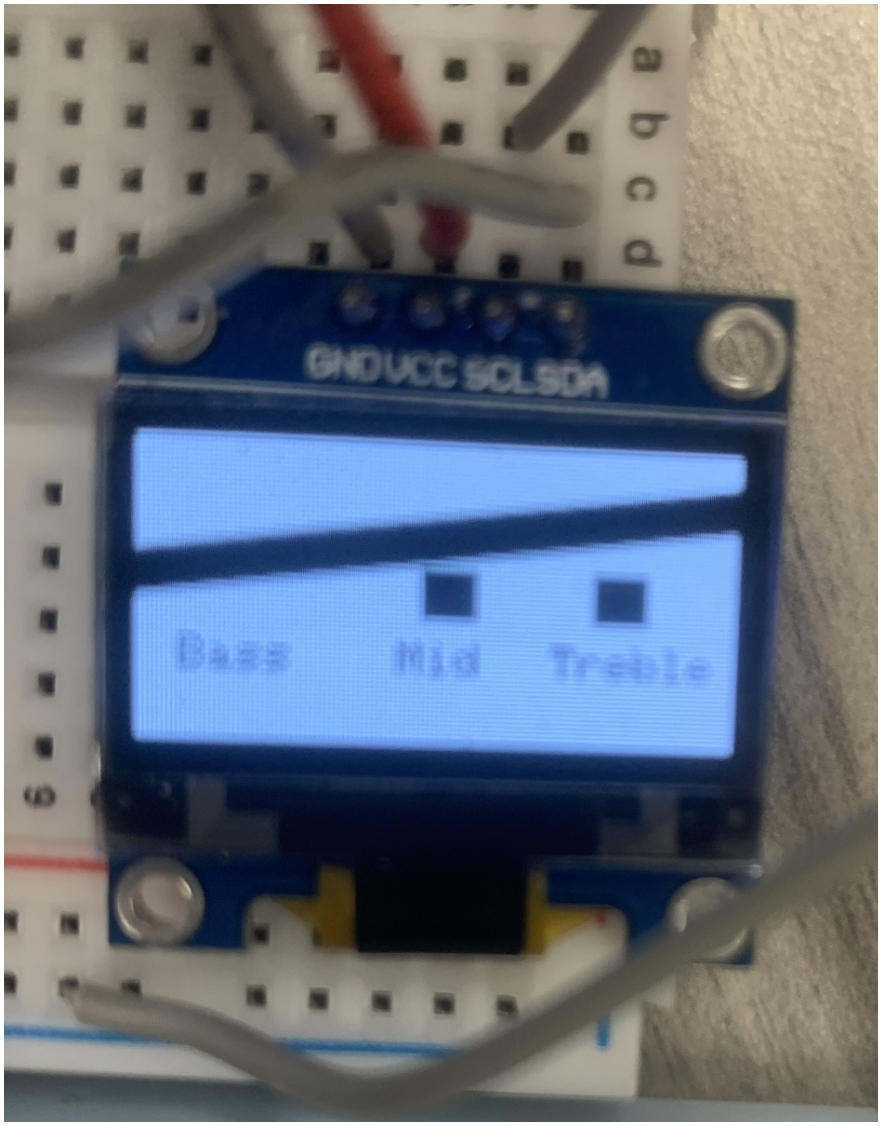

Max Bass Input



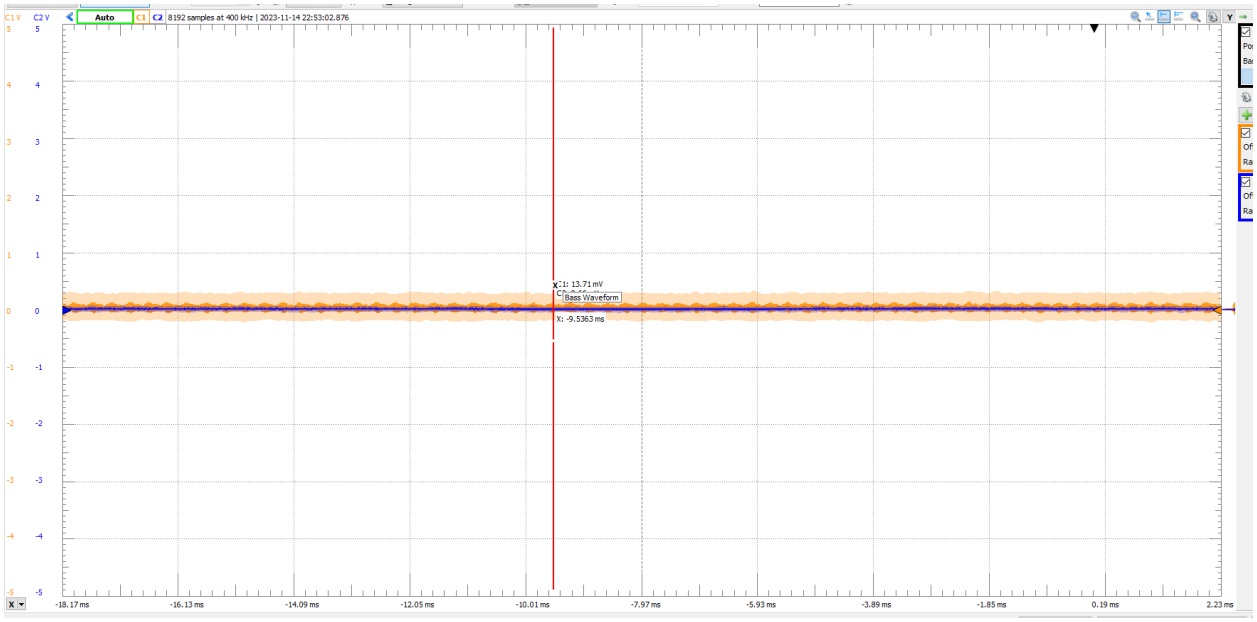
Spectrogram Response to Max Gain Bass Input




Neutral Bass Input

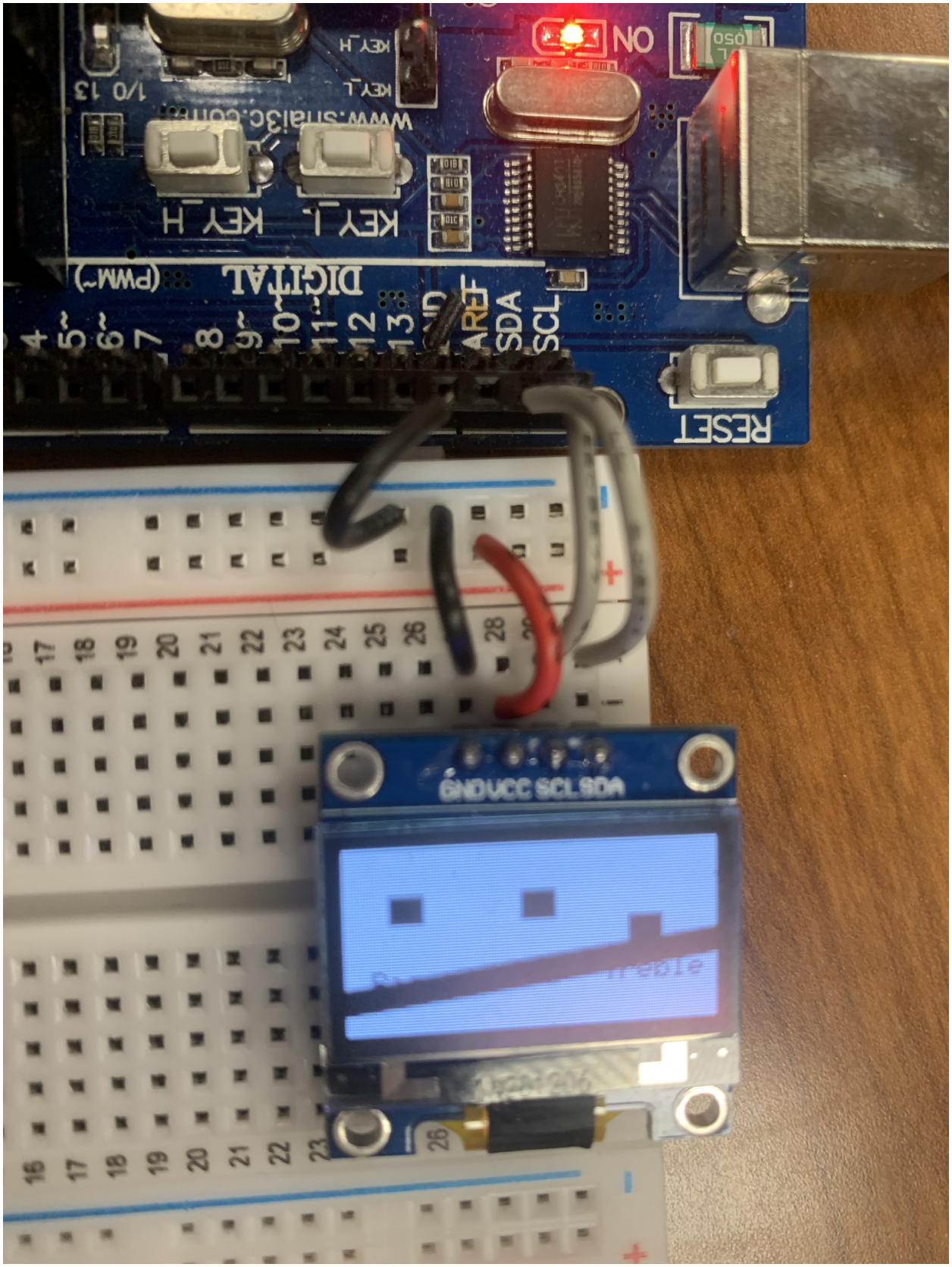



Spectrogram Response to Neutral Gain Bass Input

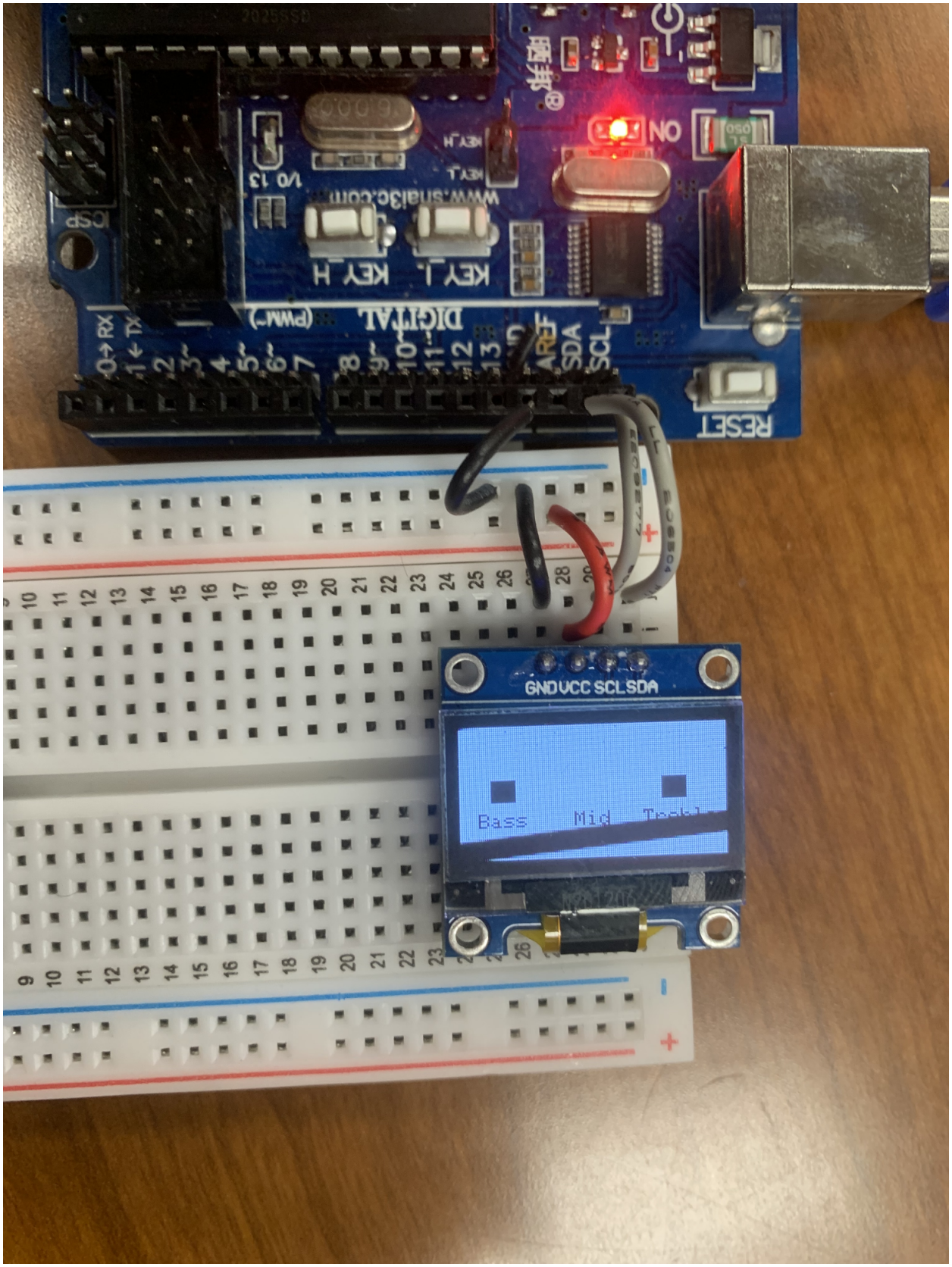



Attenuated Bass Input

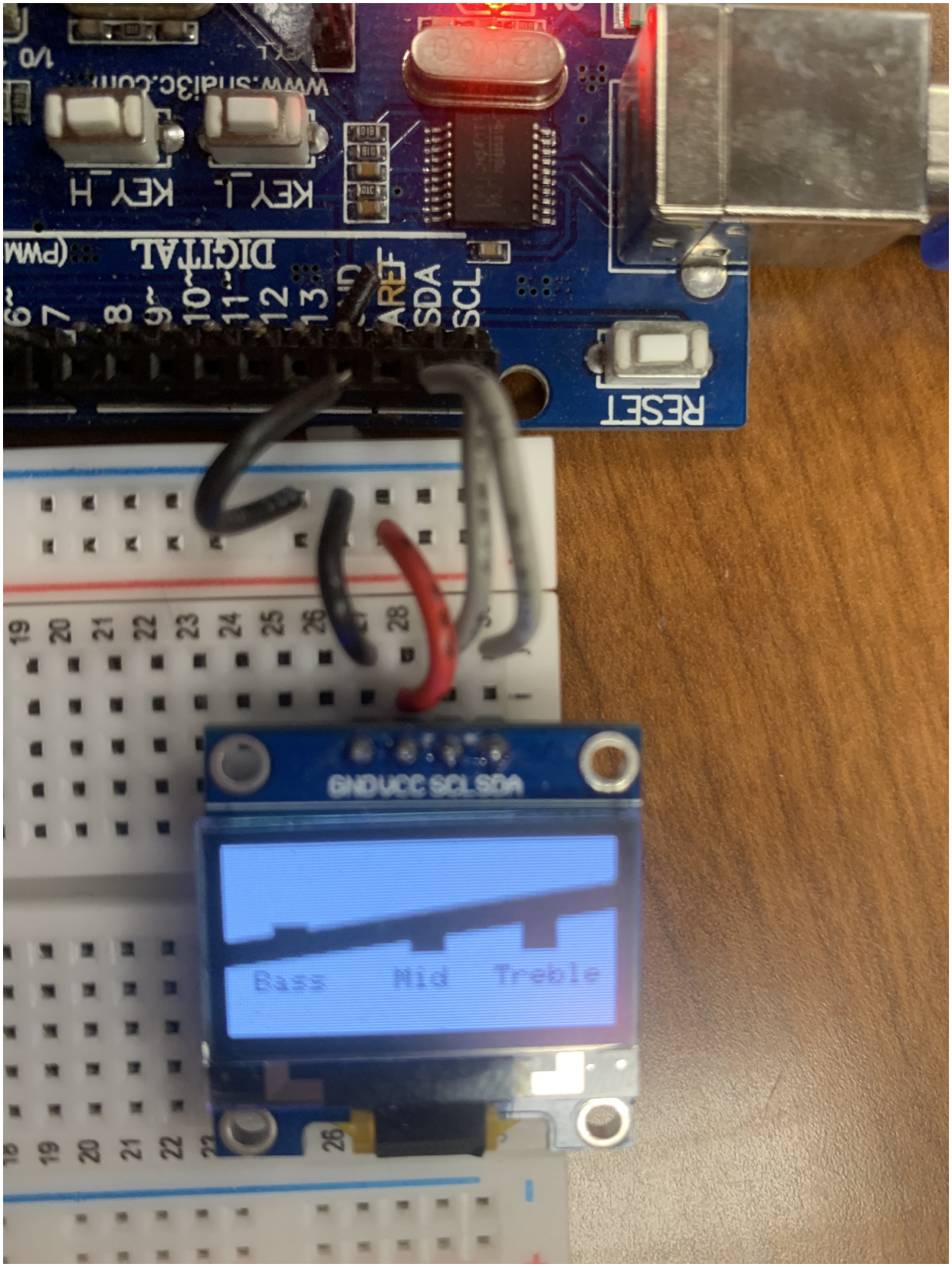

Spectrogram Response to Min Gain Bass Input



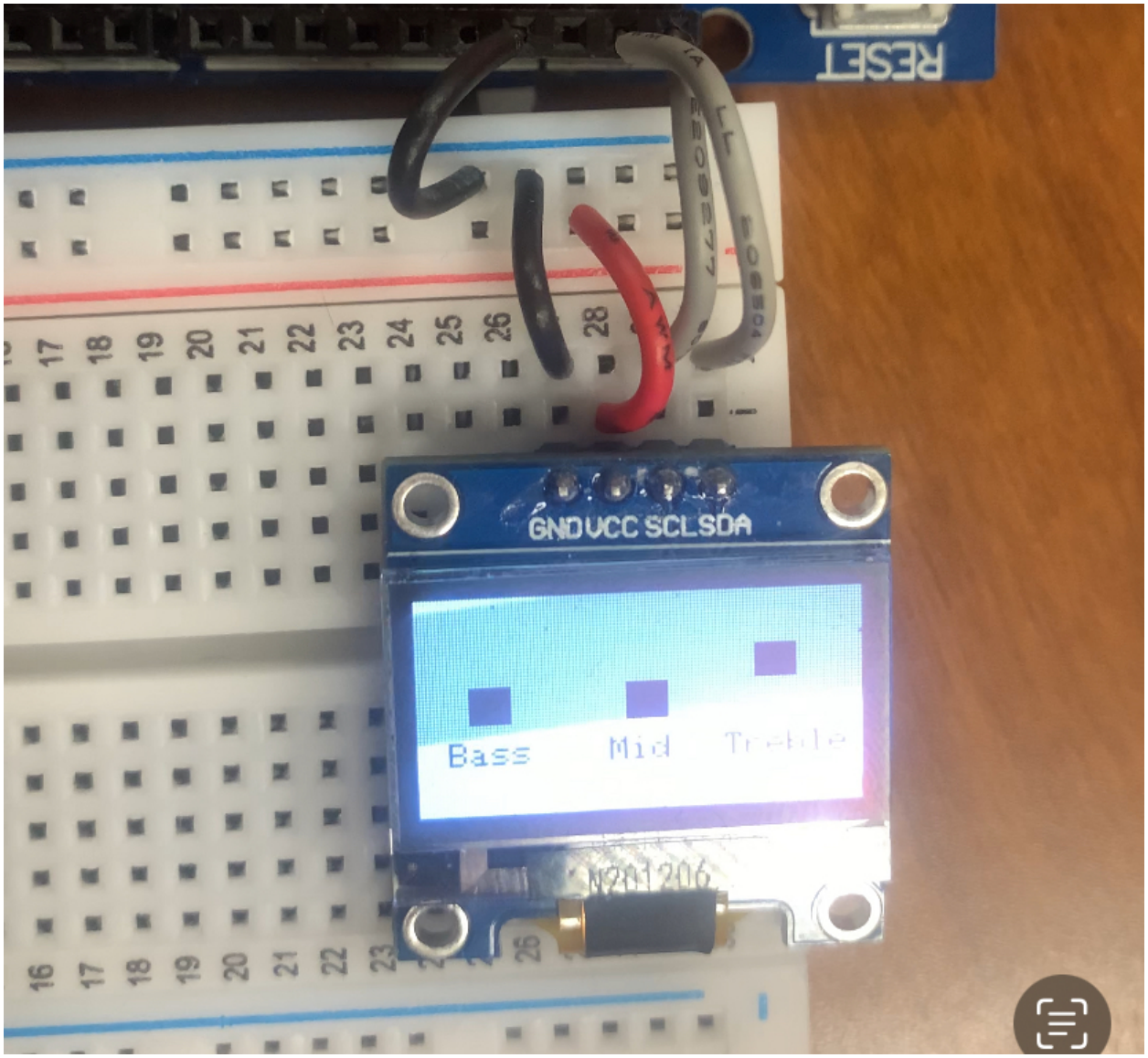

Spectrogram Response to Max Gain Mid Input



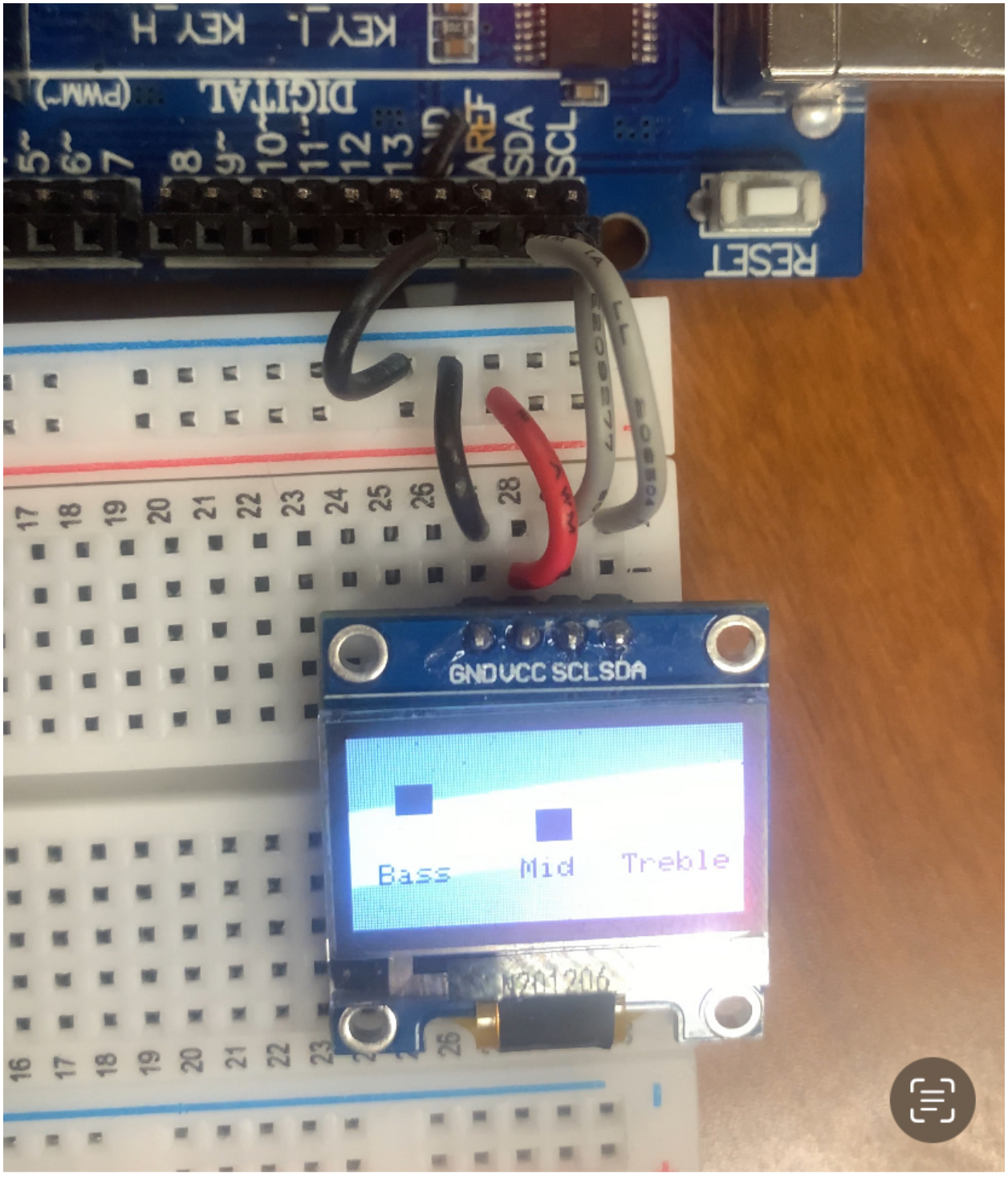

Spectrogram Response to Neutral Gain Mid Input



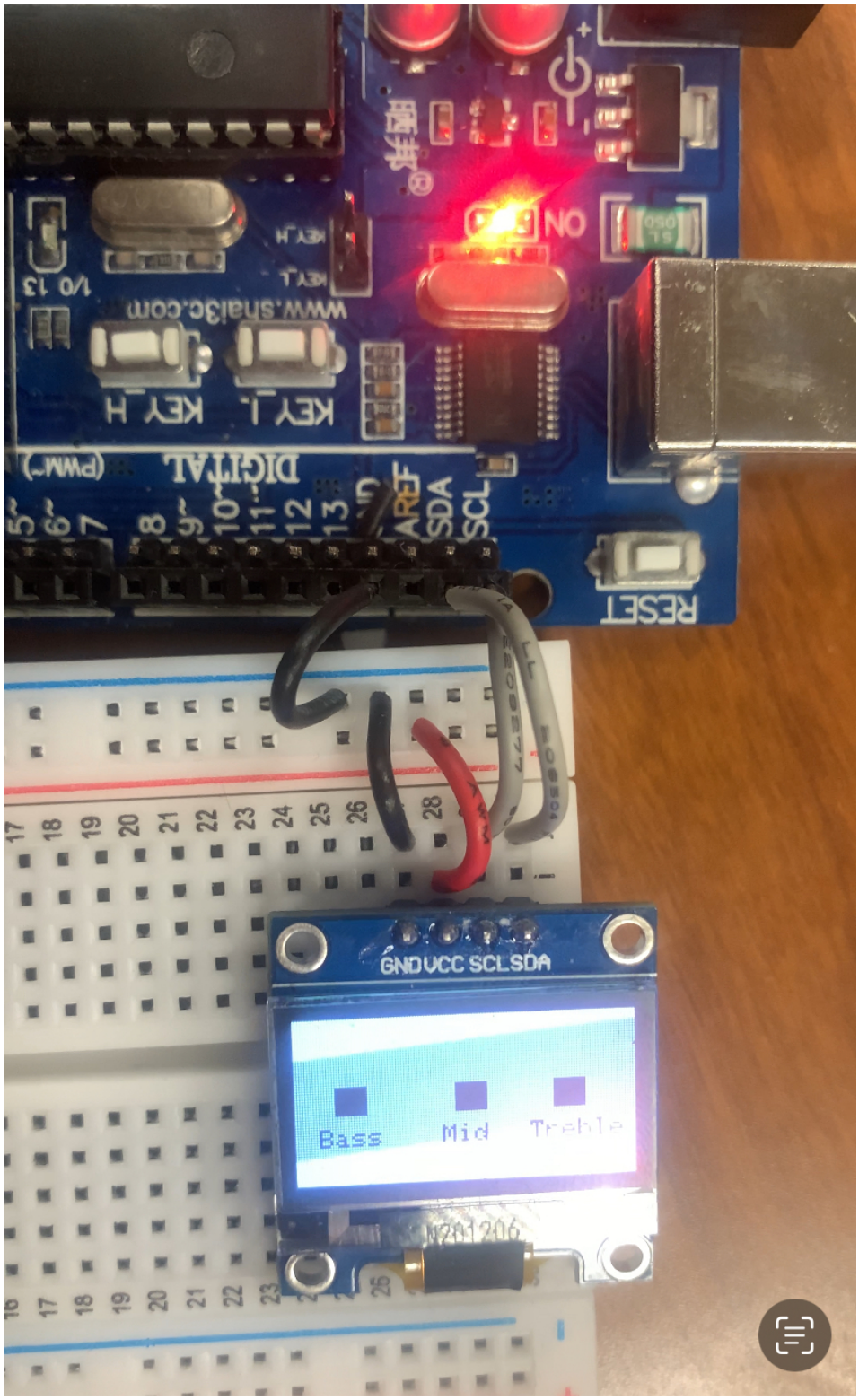

Spectrogram Response to Attenuated Gain Mid Input




Spectrogram Response to Max Gain Treble Input




Spectrogram Response to Neutral Gain Treble Input




Spectrogram Response to Attenuated Gain Treble Input

Arduino Code:

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// defines for setting and clearing register bits
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif


//Constants used in the program
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define SCREEN_I2C_ADDR 0x3C
#define INTRO_DELAY 3000
#define REFRESH_DELAY 150
#define SAMPLING_FREQ 38461.5385
#define SAMPLES 64

//Frequencies bin harmonics
#define BASS_K 1
#define MID_K  6
#define TRE_K  14

//The valid states of the program
enum class SpectogramSTATES : uint8_t {
  WELCOME_PAGE,
  MAIN_PAGE
};

//Welcome state strings and band frequencies labels
const char* DisplaySTRS[] = {"Home Audio", "Jacob and Kofi", "ECE 2804",
"Bass", "Mid", "Treble"};
Adafruit_SSD1306 spectogramDisplay(SCREEN_WIDTH, SCREEN_HEIGHT);
SpectogramSTATES currentState;

const int AudioPIN     = A0;
```

```
//const int MidPIN      = A1;
//const int TreblePIN   = A2

//Precalculated coordinates used the most
const int CommonDisplayPOS[] = {(SCREEN_WIDTH / 2), (SCREEN_HEIGHT / 2),
(SCREEN_WIDTH / 6), (2 * SCREEN_HEIGHT) / 3};
void setup() {
  //Open up serial port for debugging purposes
  Serial.begin(115200);

  //Checks if the display init was successful
  if (!spectogramDisplay.begin(SSD1306_SWITCHCAPVCC, SCREEN_I2C_ADDR)) {
    Serial.println("Unable to use display");
    while (1);
  }
  // set prescale to 16 for sampling rate for ADC
  sbi(ADCSRA,ADPS2);
  cbi(ADCSRA,ADPS1);
  cbi(ADCSRA,ADPS0) ;

  //Clear whole screen
  spectogramDisplay.clearDisplay();
  spectogramDisplay.display();

  //Set entry state to welcome screen
  currentState = SpectogramSTATES::WELCOME_PAGE;
}

void loop() {
  //Initialize data array in loop
  //Important in order to avoid out of memory situation occuring
  //when placed in global scope (Display constructor allocates for
screen ~1000 bytes)
  double data[SAMPLES] = {0.0};
  if (currentState == SpectogramSTATES::WELCOME_PAGE) {
    //On a white background, display the welcome page strings at
calculated positions and display
    spectogramDisplay.fillRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT,
0xFFFF);
```

```
    drawString(CommonDisplayPOS[0] - 30, CommonDisplayPOS[1] - 8,
DisplaySTRS[0], 0, 0);
    drawString(CommonDisplayPOS[0] - 42, CommonDisplayPOS[1],
DisplaySTRS[1], 0, 0);
    drawString(CommonDisplayPOS[0] - 24, CommonDisplayPOS[1] + 8,
DisplaySTRS[2], 0, 0);
    spectogramDisplay.display();

    //Delay for user to read
    delay(INTRO_DELAY);

    //Clear display, and draw the band frequency labels at calculated
positions for
    //one-time displaying. Avoids the need to refresh whole display
    spectogramDisplay.clearDisplay();
    spectogramDisplay.fillRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT,
0xFFFF);
    drawString(CommonDisplayPOS[2] - 12, CommonDisplayPOS[3],
DisplaySTRS[3], 0, 0);
    drawString(CommonDisplayPOS[0] - 9, CommonDisplayPOS[3],
DisplaySTRS[4], 0, 0);
    drawString(((5 * SCREEN_WIDTH) / 6) - 18, CommonDisplayPOS[3],
DisplaySTRS[5], 0, 0);

    //Set current state to spectogram screen
    currentState = SpectogramSTATES::MAIN_PAGE;
    //currentStamp = micros() / 1000000.0;
  }
  else if (currentState == SpectogramSTATES::MAIN_PAGE) {

    //Serial.print("Index: ");
    //Serial.println(index);

    //Collect <SAMPLES> amount of ADC samples into data array, converted
to voltage from [0-5]
    for (int i = 0;i < SAMPLES; i++) {
      data[i] = (((double)analogRead(AudioPIN)) * (5.0 / 1023.0));
    }
```

```
    //Initialize variables used to calculate the real and imaginary
components of
    //each discrete fourier series
    double rB = 0.0, iB = 0.0;
    double rM = 0.0, iM = 0.0;
    double rT = 0.0, iT = 0.0;

    //Perform the discrete fourier transform on each frequency
    //Cannot do the fourier transform once and grab results,
    //Due to memory constraints
    calculateFourierSeries(data, rB, iB, SAMPLES, BASS_K);
    calculateFourierSeries(data, rM, iM, SAMPLES, MID_K);
    calculateFourierSeries(data, rT, iT, SAMPLES, TRE_K);

    //Calculate the magnitude
    double bassMag    = sqrt((rB * rB) + (iB * iB));
    double midMag     = sqrt((rM * rM) + (iM * iM));
    double trebleMag = sqrt((rT * rT) + (iT * iT));

    //Draw the bars at calculated positions based on the magnitude
    spectogramDisplay.fillRect(0, 0, SCREEN_WIDTH, CommonDisplayPOS[3],
0xFFFF);
    drawBar(CommonDisplayPOS[2] - 6, CommonDisplayPOS[3] - 5, bassMag);
    drawBar(CommonDisplayPOS[0] - 4, CommonDisplayPOS[3] - 5, midMag);
    drawBar(((5 * SCREEN_WIDTH) / 6) - 9, CommonDisplayPOS[3] - 5,
trebleMag);
    spectogramDisplay.display();


    delay(REFRESH_DELAY);
  }
  else {
    //unknown state, reset
    currentState = SpectogramSTATES::WELCOME_PAGE;
  }

}

/**
```

```
 * Performs the discrete fourier transform and returns the bin frequency
 *
 * @param data:     The sampled data to perform the DFT on
 * @param real:     The real component of the bin harmonic frequency k
to store results in
 * @param imag:     The imaginary component of the bin harmonic
frequency k to store results in
 * @param samples:  The number of samples taken
 * @param k:        The bin harmonic frequency to use for the fourier
series
*/
void calculateFourierSeries(double data[], double& real, double& imag,
int samples, int k)
{

  double m = 2.0 * PI * (1.0 / samples);

  double r = 0.0;
  double i = 0.0;
  double w = (2.0  * PI * (double)k) / (double)samples;
  for (int n = 0; n < samples; n++)
  {
    real = real + data[n] * cos(w * n);
    imag = imag - data[n] * sin(w * n);
  }
}

/**
 * Draws a string at a location
 *
 * @param x:      The x coordinate to draw the string at
 * @param y:      The y coordinate to draw the string at
 * @param string: The null-terminated string to draw
 * @param col:    The foreground color of the string
 * @param bg:     The background color of the string
*/
void drawString(int16_t x, int16_t y, const char* string, uint16_t col,
uint16_t bg) {
  unsigned char c = *string;
```

```
  unsigned int counter = 0;
  while (c != 0) {
    spectogramDisplay.drawChar(x + counter, y, c, col, bg, 1);
    c = *(++string);
    counter = counter + 6;
  }
}

/**
 * Draws a magnitude bar based on tested values.
 * [0, 7] - attenuated
 * [14, 22] - normal
 * [22, double max] - gain
 *
 * @param x: The x coordinate to center the bar
 * @param y: The y coordinate to center the bar
 * @param mag: The magnitude to base the bar off of
*/
void drawBar(int16_t x, int16_t y, double mag) {
  int y_pos = 0;
  int height = 0;
  int midHeight = ((y - (SCREEN_HEIGHT / 4)) / 2);
  int midPos = (SCREEN_HEIGHT / 4) + midHeight;
  if (0.0 < mag && mag < 7.0)
  {
    y_pos = midPos;
    height = y - y_pos;
  }
  else if (14.0 < mag && mag < 22.0)
  {
    y_pos = midPos;
    height = 0;
  }
  else
  {
    y_pos = SCREEN_HEIGHT / 4;
    height = midHeight;
  }
  /*
```

```
  int height_pos = map(pinValue, 0, 1023, y, (SCREEN_HEIGHT / 4));
  int height = (y - height_pos);*/
  spectogramDisplay.fillRect(x, y_pos, (SCREEN_WIDTH / 10), height, 0);
}
```

**Authorship**

| Sections | Jacob (%) | Kofi (%) |
|---|---|---|
| Introduction | 20 | 80 |
| Block Diagram | 0 | 100 |
| Graphic Equalizer | 100 | 0 |
| Class D Amplifier | 70 | 30 |
| Spectrogram | 10 | 90 |
| Validation | 80 | 20 |
| Conclusion | 100 | 0 |
| Reviewed Doc | Yes | Yes |