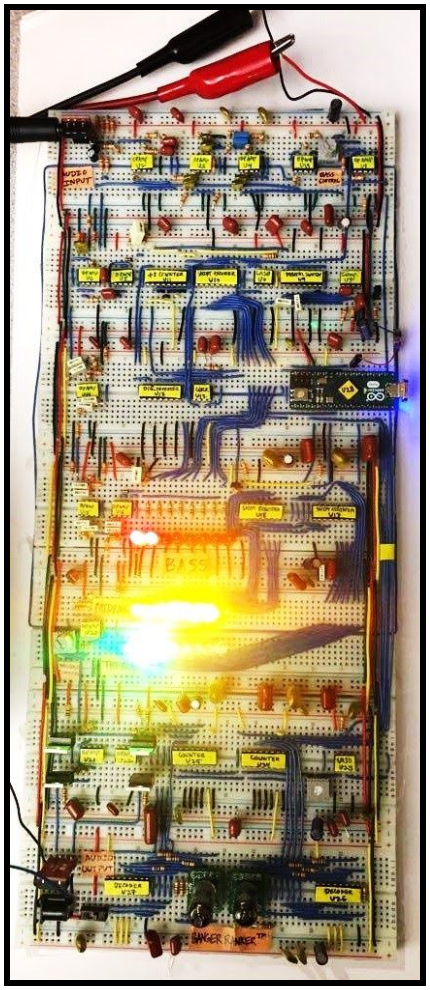
Bass-Boosted LED Music Visualizer with BangerRanker™

Billy Koech
Bryan Hu

Abstract: An LED music visualizer, using a single 9V battery, that displays the real-time amplitudes of the bass, midrange, and treble frequencies of any song plugged into the board. Includes bass-boosting control, an input 3.5mm audio jack, output speakers, an output 3.5mm audio jack, volume control, haptic board vibration, and a bass counter Numitron display.



1. Introduction

I. Overview

Personalized music is an auditory medium that only rarely employs visual or tactile elements. With our bass-boosted LED music visualizer, however, our project brings music to life by combining real-time audio, visuals, and haptic feedback. After all, there's a reason big concerts employ large, flashy light shows and bass drops you can feel in your bones—to get a multidimensional user experience out of an art form that doesn't have to be limited to the ears. We're trying to deliver that experience to the personalized, portable level.

II. Functionality

The visualizer needs 1) a song plugged into the board (through an aux cable) and 2) a 9V power source (e.g. a 9V battery).

There are six features of our project: 1) speakers, 2) three 8-LED arrays that light up in coordination with the song's bass, midrange, and treble, 3) a user-controlled bass-boosting knob, 4) a user-controlled total volume knob, 5) a motor that vibrates the board with every bass hit, and finally, 6) a BangerRanker™ Numitron display that counts and displays the number of bass hits—the higher the number, the more of a "banger" (party song) your song is.

III. Core Components

The design for the project can be split into six different functional sections:

1. Pseudo-ground (VREF) generator for single-supply design
2. Signal processing and visualization of the bass level
3. Signal processing and visualization of the midrange level
4. Signal processing and visualization of the treble level
5. Bass hit counter, Numitron display, and haptic motor vibration
6. Combining the bass and original audio signals to produce bass-boosted audio output

2. The Design

I. Overview

The Design section will be organized as follows: first, we will present the block diagrams for each of the aforementioned six core functional components, showing the relevant subsections and their relationships to each other. Then we will dedicate a chapter to understanding how we translate from audio through an aux cable to an initial analog voltage signal suitable for manipulation in our circuit. Finally, we will go through each of the six core functional components in detail, following the given structure of its corresponding block diagram.

II. Block Diagrams

Below are the block diagrams for each core functional section.

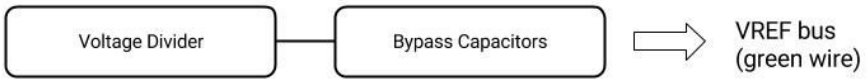


Figure 2.1: Pseudo-ground generator

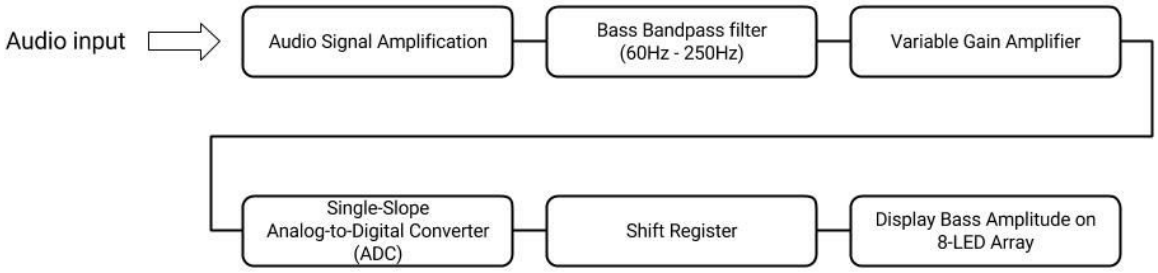


Figure 2.2: Signal processing and visualization of the bass level

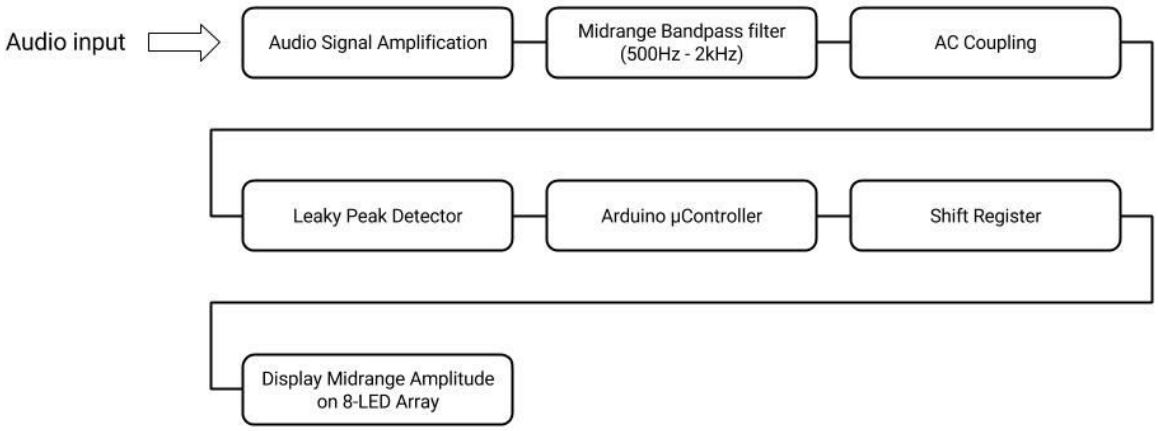


Figure 2.3: Signal processing and visualization of the midrange level

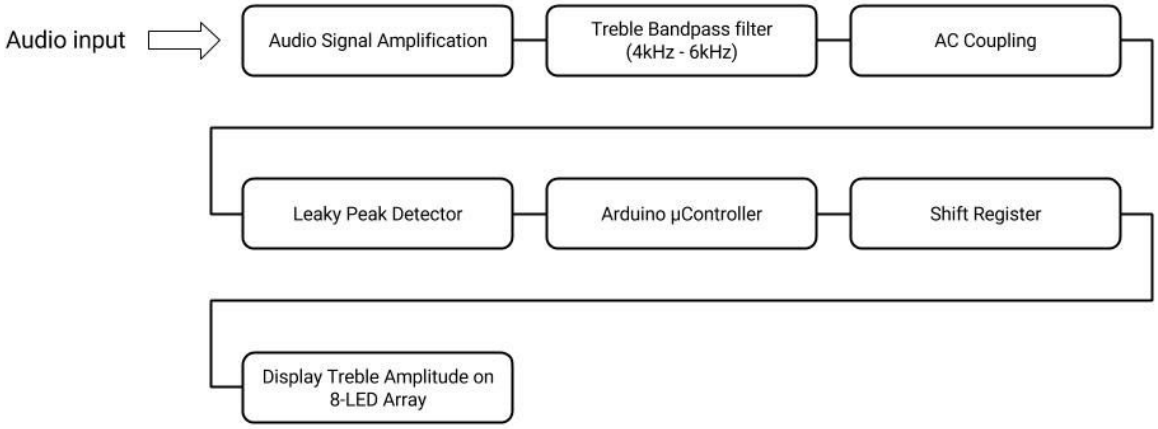


Figure 2.4: Signal processing and visualization of the treble level (same as midrange)

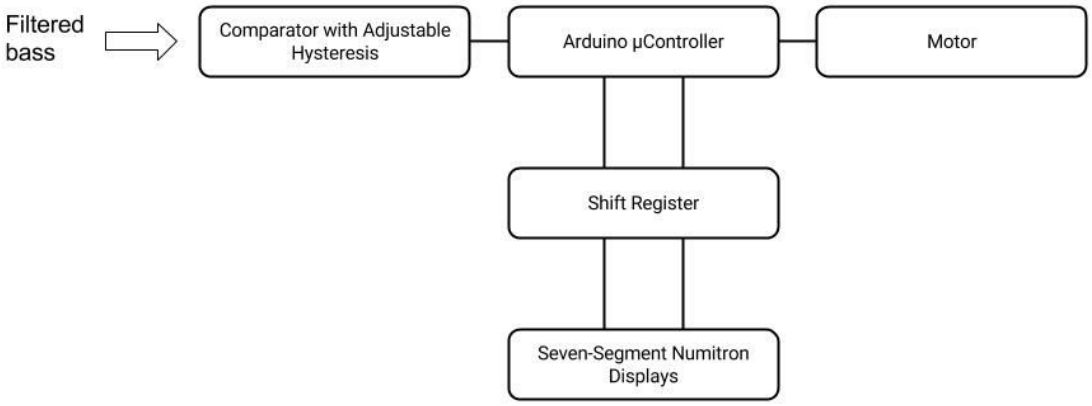


Figure 2.5: Bass hit counter, Numitron display, and haptic motor vibration

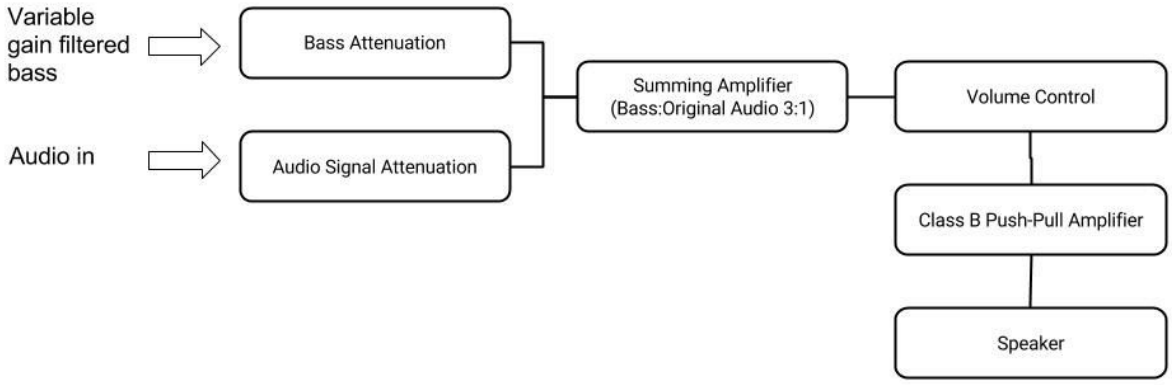


Figure 2.6: Combining the variable gain bass and original audio signals to produce bass-boosted output audio

III. Understanding the Music Input

We planned to have our music visualizer take in music playing from a phone, a laptop, or any other device through an auxiliary cable. For this reason, before designing the visualizer, we first had to understand the typical frequency range of music, the typical voltage output range coming from the headphone jack of phones and laptops, and how the channels (left and right) of audio coming from the input audio jack work.

A. Typical Frequency of Music

The frequency of audible music ranges from 20Hz to 20KHz.[1] For this project we chose to use three ranges of frequencies for our display:

I. Bass : 60Hz - 250Hz
II. Midrange: 500Hz - 2kHz
III. Treble: 4kHz - 6kHz

We chose these frequencies because they are likely to occur in songs of different genres.

B. Typical Voltage Output of Audio Devices

Various online forums and websites suggested that the typical output voltage of 3.5mm headphone jacks is between 1.5 V peak-to-peak[2] and 3V peak-to-peak[3]. Because of the variation in values provided online, we decided to measure the voltage coming out of our own phones

[1] https://www.teachmeaudio.com/mixing/techniques/audio-spectrum
[2] https://www.ifixit.com/Answers/View/97418/what%27s+the+iPhone+3.5+mm+output+jack+power+supply+output
[3] https://electronics.stackexchange.com/questions/37926/what-is-the-typical-max-voltage-out-of-a-pc-speaker-jack

when playing music at ½ volume, ¾ volume, and maximum volume, so as to obtain accurate values for our design. We measured the left channel of a music signal from a phone. Below is a table with of the recorded data from the measurement:

Volume level	Peak-to-peak voltage
½	160 mV
¾	480 mV
Max volume	3.04 V

Figure 3.1: Measured voltage values from a 3.5mm jack of an Iphone 6S at different volume levels. *Song playing: WizKid - Sound It*

We also noticed that the voltage rarely reached or exceeded 3V peak-to-peak. This made 3V a good theoretical choice for us to use as a maximum peak-to-peak input voltage in our subsequent calculations.

Having defined 3V as our maximum peak-to-peak input voltage, we designed all subsequent amplifiers within our circuit such that any no signal would exceed a maximum output of 9V peak-to-peak. This is because our circuit runs off of 9V, single-supply, and any music above 9V peak-to-peak would be clipped, which we want to avoid. The implementation of this design choice applies to all op amps used in this project.

C. L+R Audio Channels

Smartphones and laptops have the capability of playing audio on two separate channels—left and right. We had no need for distinction of music by channels, but we did want to include all audio and treat it as if it were mono audio. Therefore, we chose to combine the left and right channels using a non-inverting summing op-amp configuration. Below is a schematic of the non-inverting summing op-amp configuration:

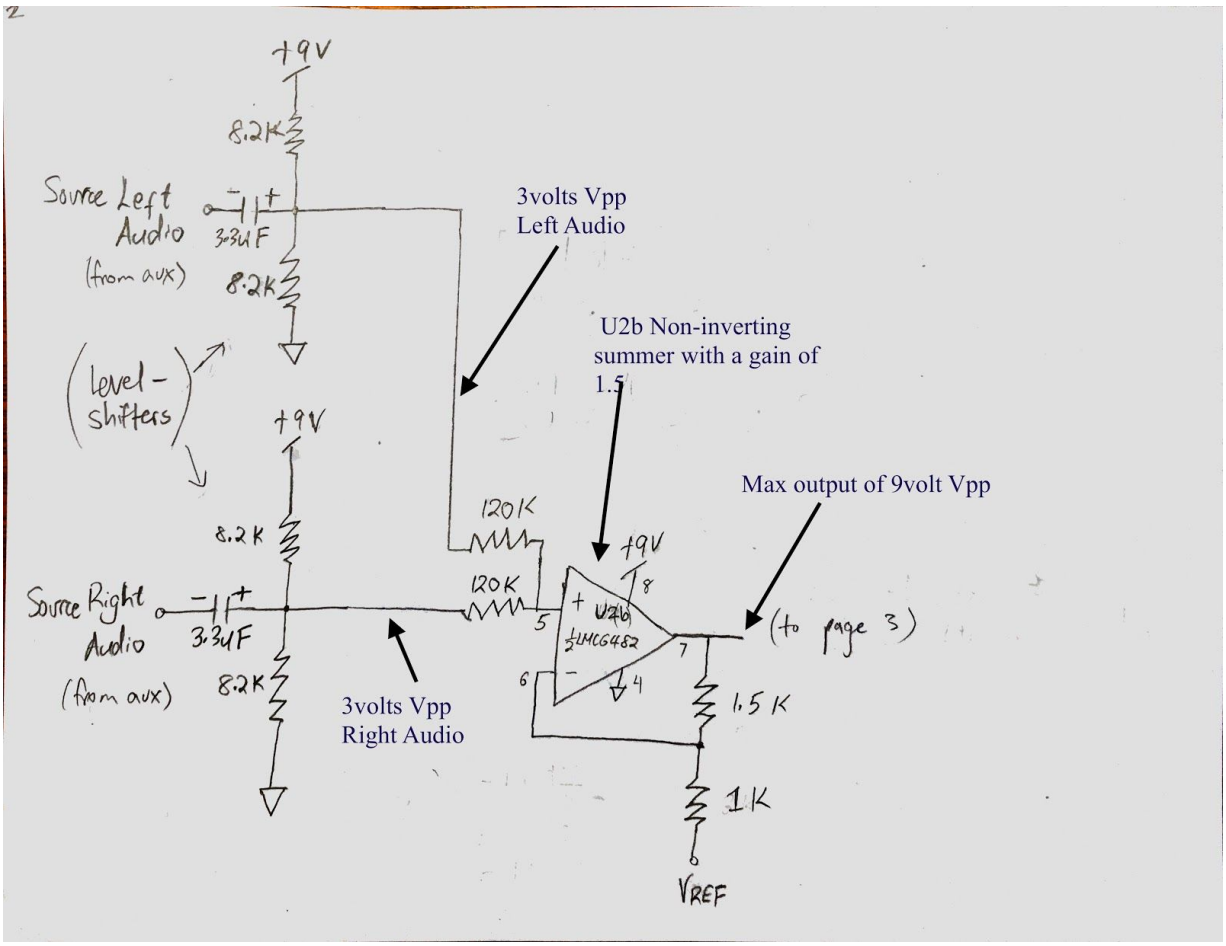


Figure 3.2: Conversion of stereo to mono using a non-inverting summing op-amp configuration to combine L+R audio with a gain of 1.5

The use of a summing op-amp also gave us the opportunity to amplify our audio signal at the same time. We wanted to amplify our audio signal to the full range for the most accurate measurements in later stages. We chose to amplify at this stage, rather than later, for two reasons:

i. To reduce the number of op amps used.
ii. To prevent amplification of noise after the signal is later filtered. It is undesirable to amplify after filtering because that would not only amplify the passed signal but also amplify the noise from the filtered signal.

The configuration in Figure 3.2 has a gain of 1.5 and meets the 9V peak-to-peak constraint stated in part B of section III above as per the calculations below:

Maximum Sum of left and right = 3V + 3V = 6V
*Amplification of the maximum sum of left and right by 1.5 = 6V * 1.5 = 9V*

Now that we understand how music is inputted into our board, we move on to examining each functional section in more depth.

IV. Single-Supply Design

Since we plan to run our circuit on a 9V battery, we choose to design our project in single supply. In other words, instead of having a range of -9V to +9V, for example, to work with, we are limited to 0V to +9V. This means that we can never have negative voltages and that most analog signals should be now centered around +4.5V, the new midpoint. We now call +4.5V "pseudo-ground" or VREF in the schematic. Below is a circuit that produces +4.5V for the pseudo-ground (VREF) bus:

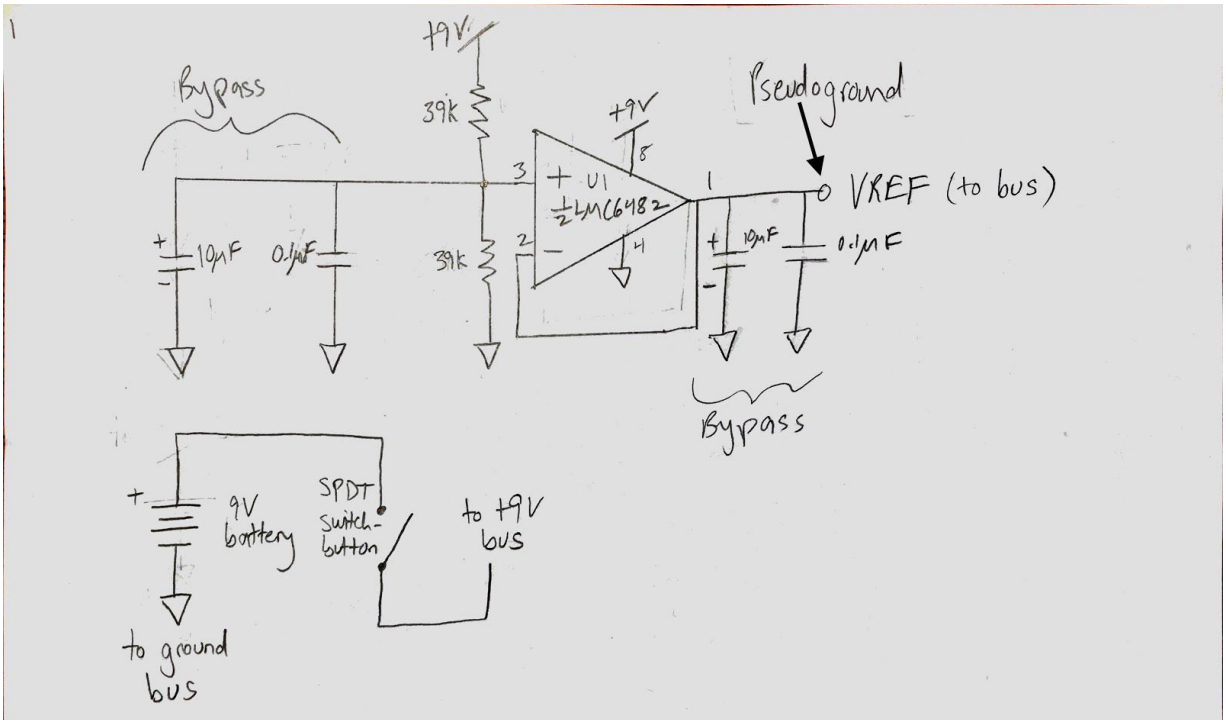


Figure 4.1: Pseudo-ground (VREF) generator 9V battery connected to SPDT switch

Single supply also meant that we had to level-shift the DC levels of all of our audio signals to +4.5V before sending it into our op amps so as to prevent clipping. Below are the schematics of two level shifters—one for the left and the other for right channels of audio: (Level shifting was done before the stereo to mono conversion that we have illustrated in Figure 3.2 of section III above)

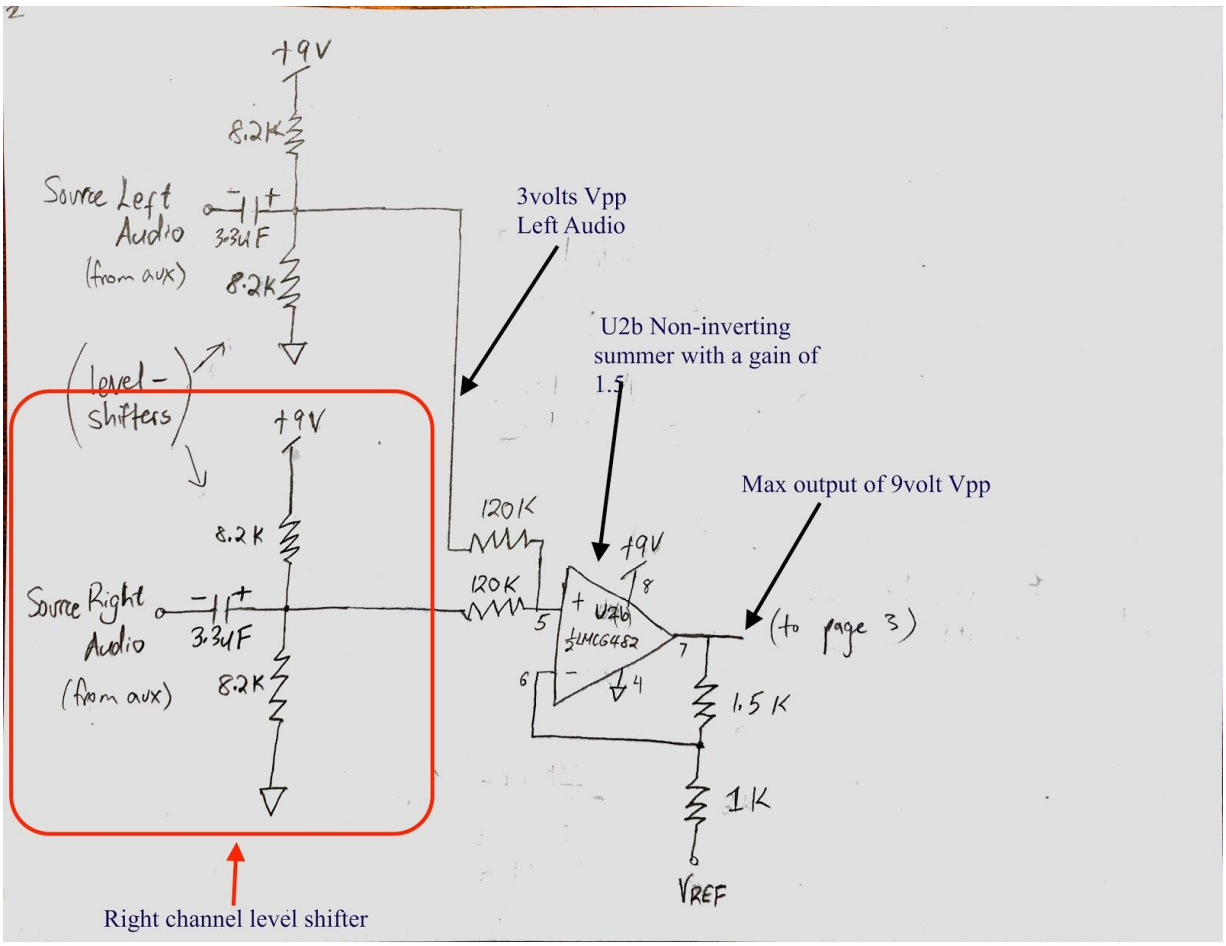


Figure 4.2: Level shifting the DC level of the right channel of our audio signal to 4.5V

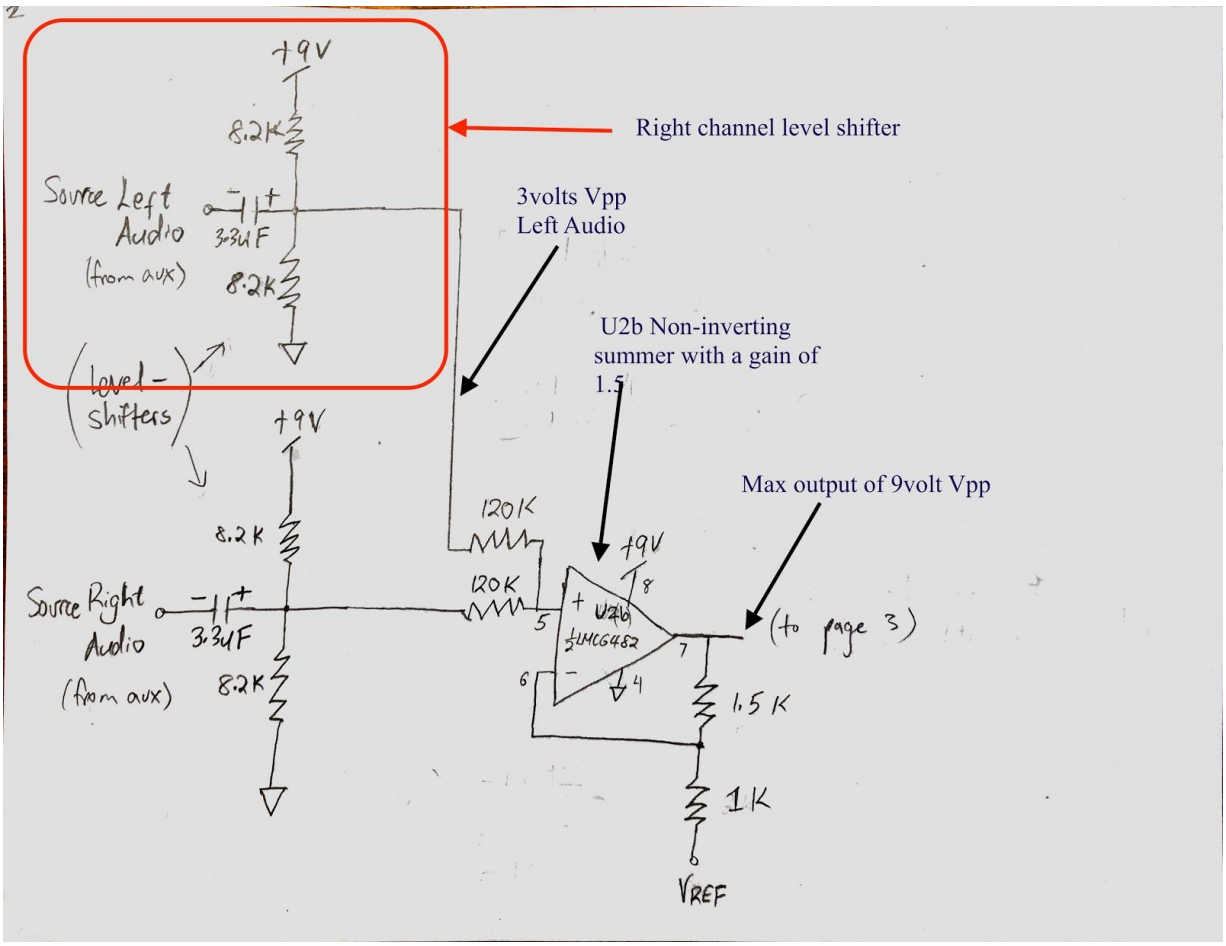


Figure 4.3: Level shifting the DC level of the left channel of our audio signal to 4.5V

V. Bass Signal Processing & Visualization

A. 60Hz - 250Hz Bass Bandpass Filter

We need to take in the source audio signal (already combined and amplified, as in part III C) and filter out everything but the bass component. To do this, we need a strong filter that declines steeply outside the passband. The following bandpass filter was designed using analog.com's Filter Wizard.[4] Below is a schematic produced by analog.com for an active five-stage 60Hz - 250Hz bandpass filter:

[4] http://www.analog.com/designtools/en/filterwizard/

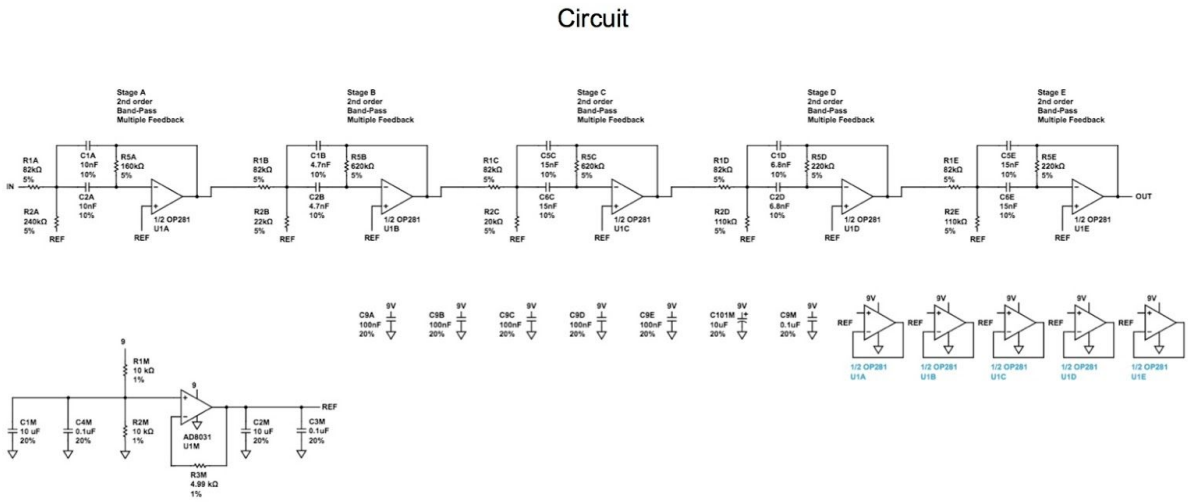


Figure 5.1: Schematic of a 60Hz - 250Hz bandpass filter from analog.com

The filter passed signals between 60Hz and 250Hz and had an attenuation of about -100 db per decade beyond 250Hz and one of 100db per decade below 60Hz. We adapted the design above to suit our parts by replacing the OP281 op amps with LMC6482 op amps, as we had more experience with the LMC6482's, they were readily available in lab, and they had a sufficient rail-to-rail input range. Below is a schematic of the bass bandpass filter that we built:

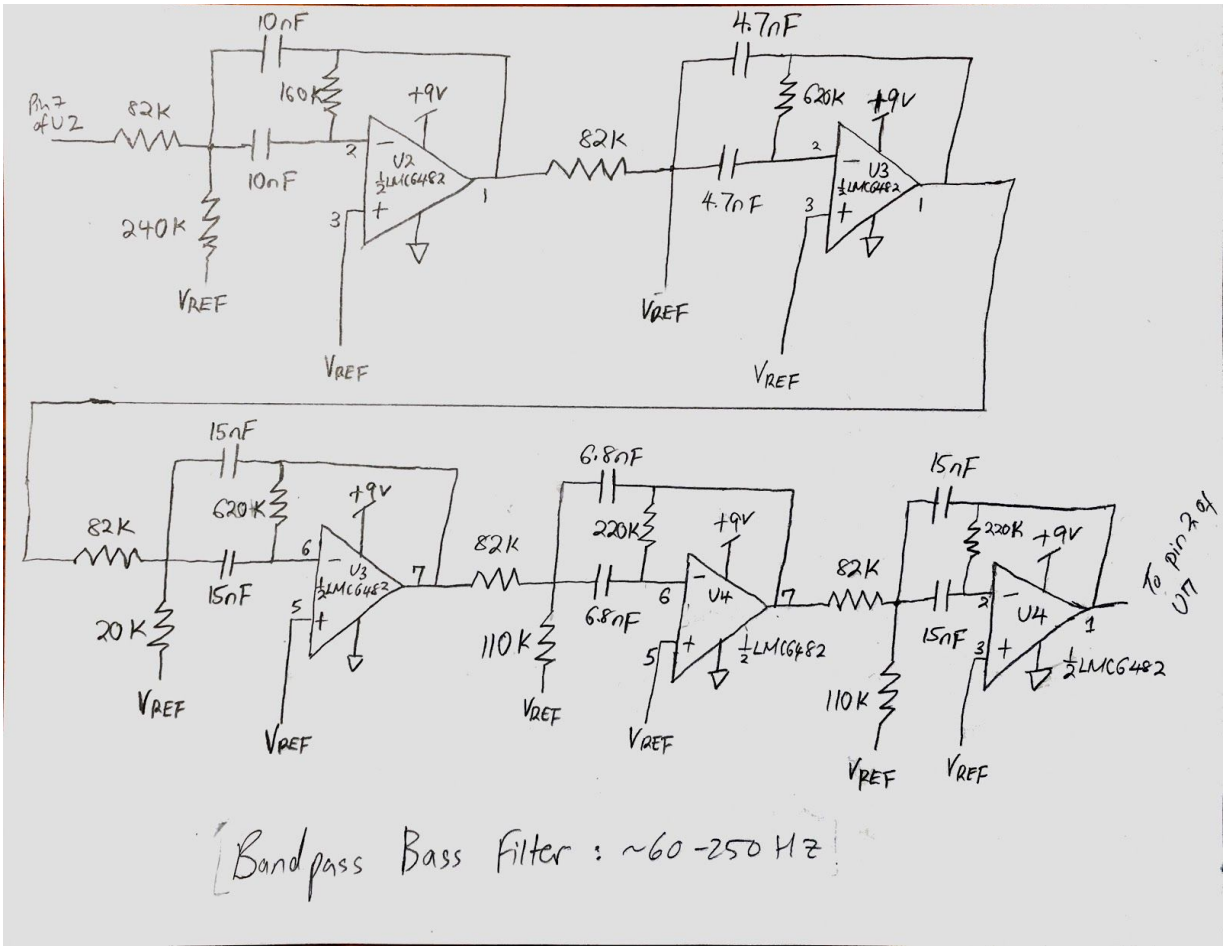


Figure 5.2: 60Hz - 250Hz bandpass filter based off of a design from analog.com's Filter Wizard.

B. Variable Gain Amplifier for Bass Boosting

We wanted to be able to control the gain of our bass signal, so that the user can "boost" the bass whenever they wanted. Therefore, we designed a variable gain amplifier using an op amp with a maximum gain of 1 and a minimum gain of 0. This enabled us to control the amplitude of the bass signal going beyond this point. Below is a schematic of a variable gain amplifier with an inverting op amp configuration:

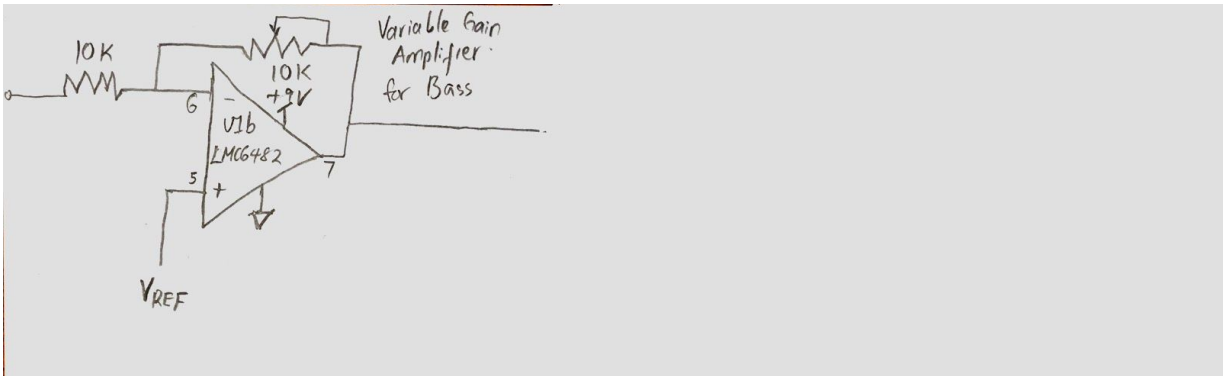


Figure 5.3: Variable gain amplifier with an inverting op amp configuration (range of gain: 0 to 1)

C. Single-Slope Analog-to-Digital Converter (ADC)

To display the amplitude of the bass on an array of LEDs, we converted our analog signal into digital. To do this, we designed a single-slope analog-to-digital converter (ADC). The basic idea behind a single-slope ADC is that you compare an analog input waveform that you want to know the decimal level of against a linearly-sloped known reference voltage, and the time it takes for the input signal to cross that slope is proportional to its decimal level at that moment. You compare the input waveform with that slope repeatedly (resetting the sloped line at a fixed speed) to get a rough estimate of its digital level over time.

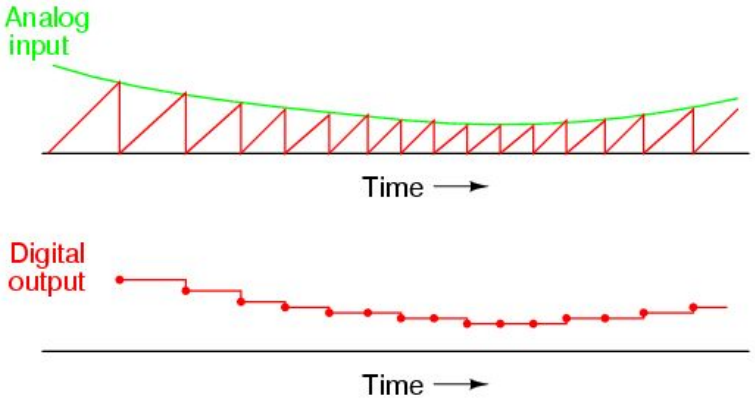


Figure 5.4: Converting an analog waveform to digital outputs using a single-slope ADC. Image courtesy of sub.allaboutcircuits.com/images/04268.png

First, we needed to create our known, repeated slope. We did this by using a resettable non-inverting integrator to integrate a constant voltage, generating a linear slope that rises from

4.5V to 9V. We chose to integrate in this range because we only intended to sample just the top half of our audio signal (which occurs between 4.5V and 9V), rendering only the top half of the slope useful to us. Only the top half of the signal is necessary to digitize because we want to know the max level of the bass signal anyway, which always occurs in the top half, never in the bottom half. In addition, we chose the non-inverting configuration, rather than the inverting configuration we've seen in lab, because we needed a positive slope.

We intended to acquire at least 3 samples per wave for any frequency of bass, as this would provide us with sufficient resolution to ensure we weren't missing any waves. Since the highest theoretical passed frequency of our bass signal is 250 Hz as per the specification of our bass passband filter (Section V part A), our sampling frequency was thus chosen to be 3*250Hz = 750Hz. From our chosen sampling frequency, we were able to determine the RC integration time for our integrator:

*1/750hz = 0.00133 sec = 1.33 milliseconds = R*C*

This means that it will take our integrator 1.33 milliseconds to rise from 4.5V to 9V in a linear fashion .Below is a schematic of the non-inverting resettable integrator:

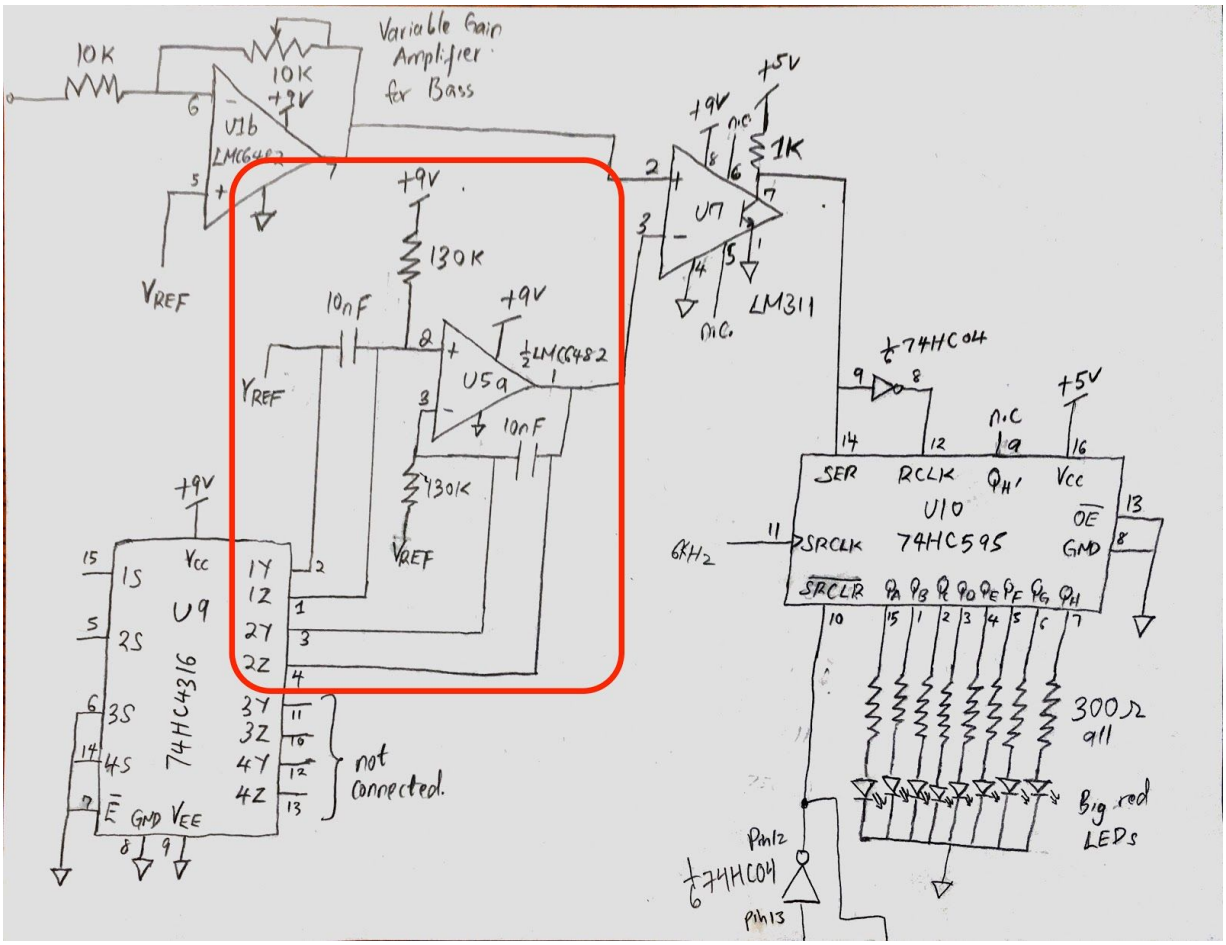


Figure 5.5: non-inverting resettable integrator that integrates from 4.5V to 9V in 1.33mSec

The slope from the integrator was then compared to the output of the variable gain amplifier that we have talked about in section V part B above using a LM311 comparator. We chose the LM311 over the LMC6482 because we need to set the output at 0V (low) and +5V (high) for the subsequent digital circuit whilst still taking in a signal between 0 and 9V, which the LM311 can do, but the LMC6482 cannot.

Using the ADC in the above manner also means we needed to reset the non-inverting integrator every 1.33mSec. To do this, we built a 6kHz clock using an LMC555 timer and used a 4 bit counter to divide that frequency by 8 so as to achieve a 750Hz clock, which we then hooked up to a digital switch that resets the necessary capacitors at the desired rate. The capacitors, when shorted, reset essentially instantaneously, so the ramp would be reset back to 4.5V and the integration would repeat. Note: We decided to build a 6kHz clock and not just a 750Hz clock right off the bat because we'll need the 6kHz clock later anyway. (In particular, the 6kHz clock shifts in data to light the 8 LEDs through a shift register, as we'll discuss later.) Below is a schematic of the LMC555 timer and the divide-by-8 counter:

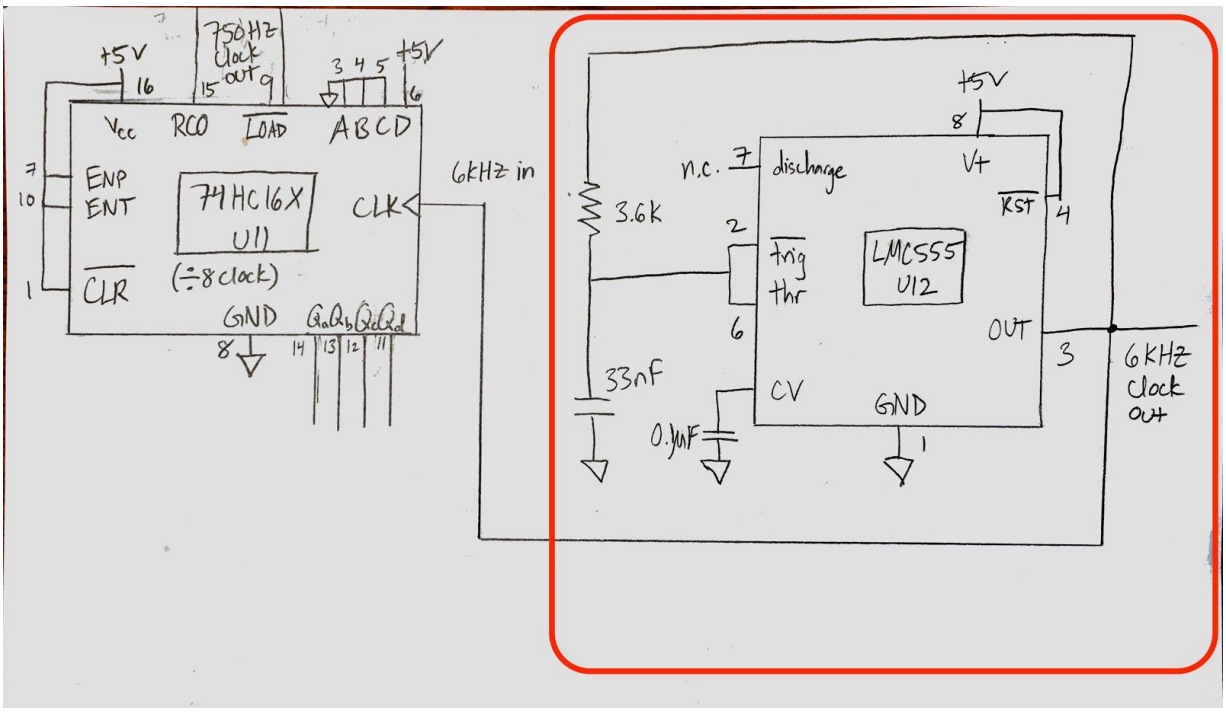


Figure 5.6: LMC555 timer that produces a 6kHz clock and a divide-by-8 counter that produces a 750Hz clock

The 750Hz clock was then connected to a digital switch (74HC4316) that allowed us to short the two capacitors in the integrator at the same time. Below is a schematic of the non-inverting integrator with the 74HC4316 digital switch:

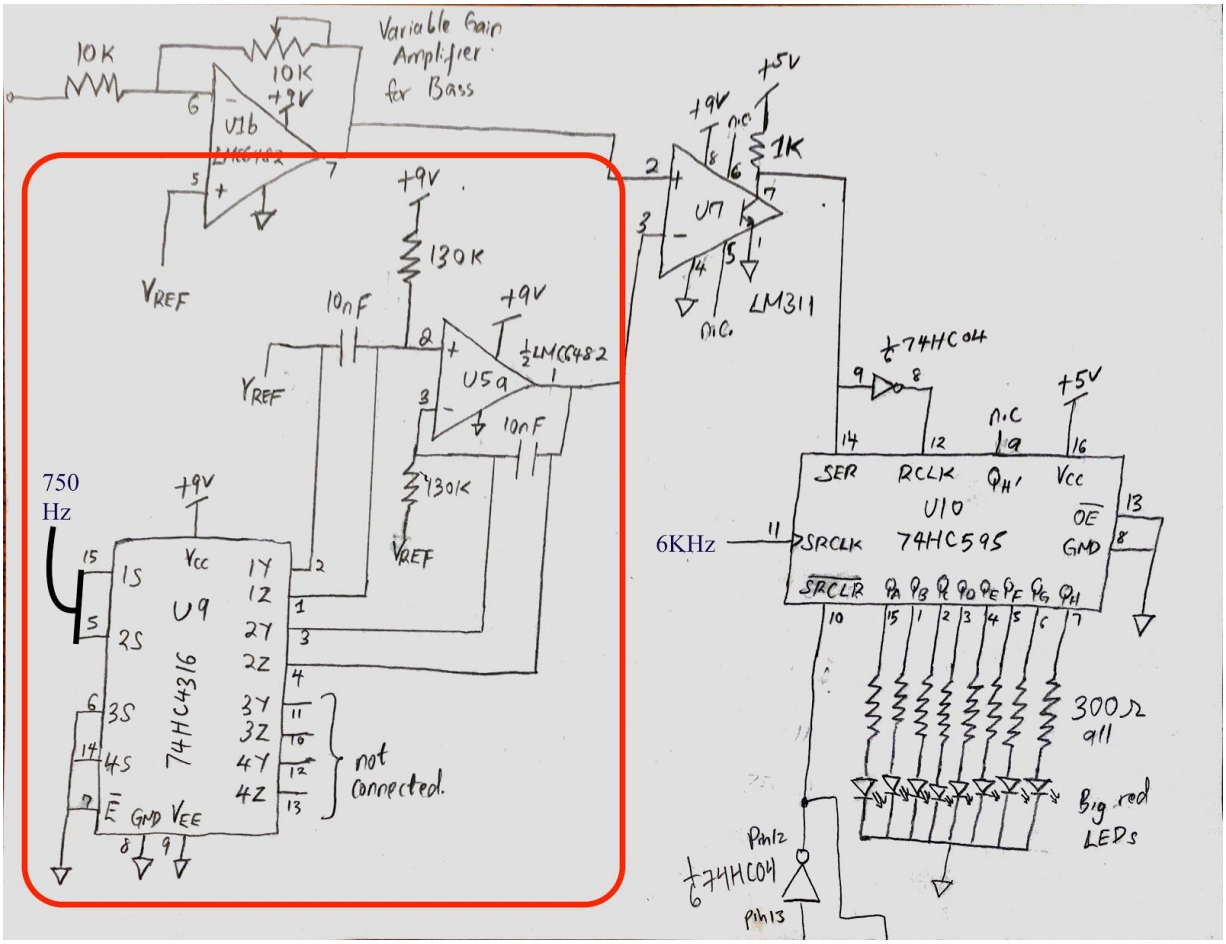


Figure 5.7: non-inverting integrator connected to a 74HC4316 digital switch

At first, we designed the ADC to produce a 3-bit encoded output using a 4-bit counter. The output runs from 000_2 to 111_2 in binary (i.e. 0 to 7 in decimal). But this posed the problem of decoding the output, i.e. putting it in a displayable form, because for a given number, we want that particular LED to light up *and light up all the LEDs below it*. For example, if the output is zero (the lowest), we want just the first LED lit up, but if the output is 7 (the highest), we want all eight LEDs lit up. (This will create the audio visualizer "wave", which is more visually pleasing than a single dot moving around.)

Still trying to implement a 4-bit counter this way, we first tried using a logic table, but that quickly got too complicated (an inordinate amount of NAND gates would have to be used). Then, we tried using a 3-bit-to-8-line converter available in lab, which would easily turn our 3-bit binary outputs into the corresponding decimal position (i.e. 000_2 to 10000000 and 111_2 to

00000001), but again, there was no clear way to light up more than a single LED in the way that we wanted, even with the converted 8-line parallel output.

D. Shift Register

Fortunately, after discussing with our ES52 instructor, we adopted a simpler design without any binary counting that allowed us to achieve our desire of lighting up more than a single LED at a time. Our selected design involved the use of a shift register in place of a 4-bit counter (i.e. no more counter and no more 3-to-8 converter). We fed the 6kHz clock into the clock pin of the 74HC595 shift register and fed the output of the comparator into the data pin. This allowed us to display shifted data on multiple LEDs if the input data of the shift register had been HIGH for more than one rising edge of the 6kHz clock.

In this design, we shifted in HIGH when the comparator output was +5V, then latched the output of the shift register when the output of the comparator changed to LOW, i.e. when the input waveform crossed the integrated slope. For example, if the output of the comparator was HIGH for 6 clock cycles, then instead of having a counter count up to 6 or 110_2 in binary as originally intended, we simply shifted in 6 HIGHs through the shift register, turning on only the first 6 LEDs. As soon as the output of the comparator went LOW, we latched that value so that the LEDs would stay in the first six slots, otherwise zeros would be shifted in and the 6 LEDs would be pushed to the other side of the array.

Then, everything, including both the integrator and shift register, is reset at a 750Hz rate. The cycle then begins anew after the next time that the input waveform goes higher than the integrated slope, as new HIGHs are shifted in for another set amount of time. We used the 750Hz clock to thus reset the shift register at the same time that we reset the integrator (as above). Below is a schematic of the combination of the non-inverting integrator, the 74HC4316 digital reset switch, the LM311 comparator and the 74HC595 shift register that make up the single-slope ADC:

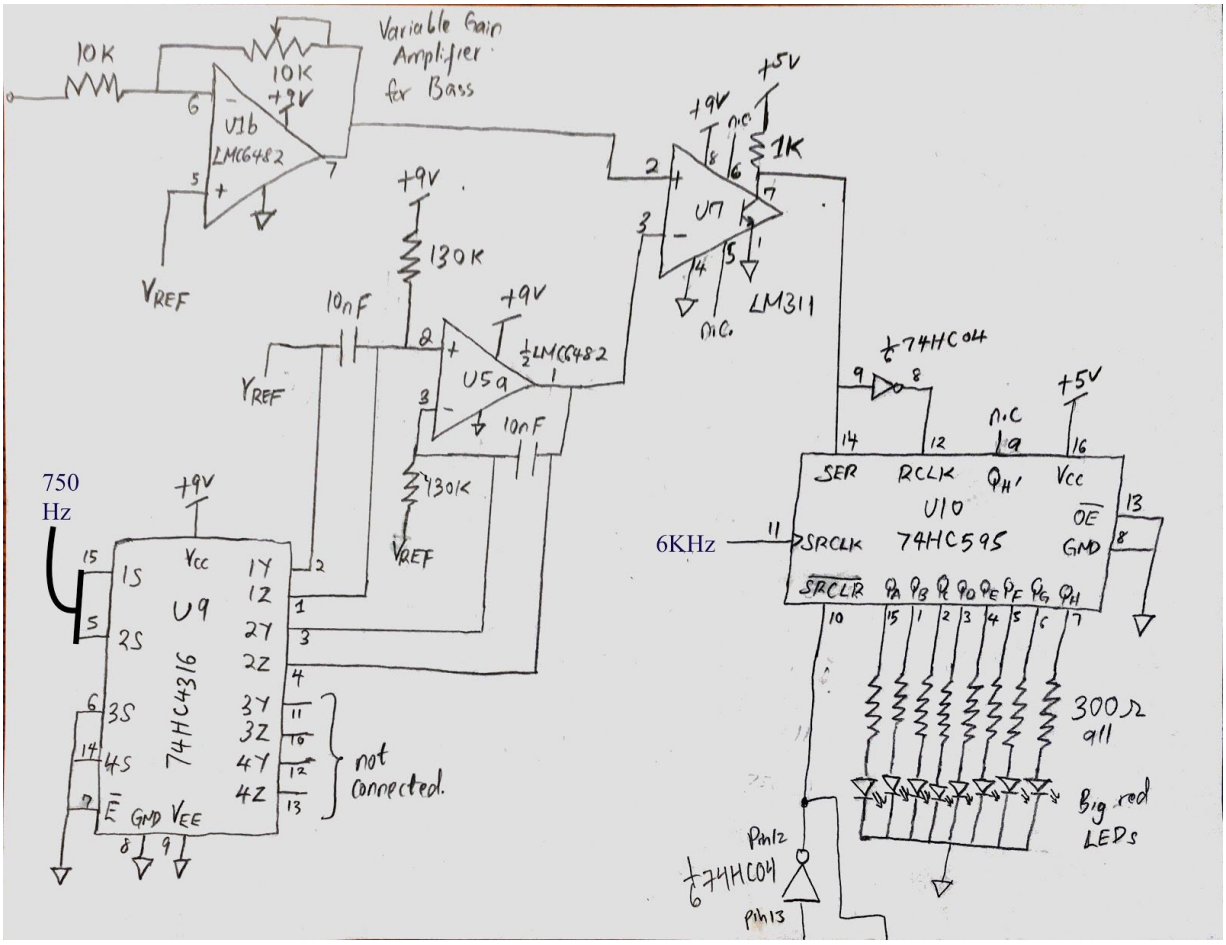


Figure 5.8: single slope ADC (74HC4316 digital reset switch, LM311 comparator and the 74HC595 shift register)

E. Display Bass Amplitude on an 8-LED Array

The output pins of the shift register were then connected to 8 big red LEDs with forward voltages of about 2V. We then chose to drive 10mA through them because 10mA provided sufficient brightness for our purposes. This current was set by a current limiting resistor of 300Ω. Figure 5.8 above shows the resistors and the LEDs connected to the output of the 74HC595 shift register.

VI. Midrange Signal Processing & Visualization

A. 500Hz - 2kHz Midrange Bandpass Filter

Just as in the bass section, we now go back to the start and take the L+R combined and amplified source audio signal (from Section III C) and take it down another path. This time, we filter out everything but the midrange frequencies with the aim of eventually displaying their levels. To do

this, we designed the midrange bandpass filter using one low pass filter that passes frequencies lower than 2KHz and one high pass filter that passes frequencies higher than 500Hz. Below is the schematic for the bandpass filter:

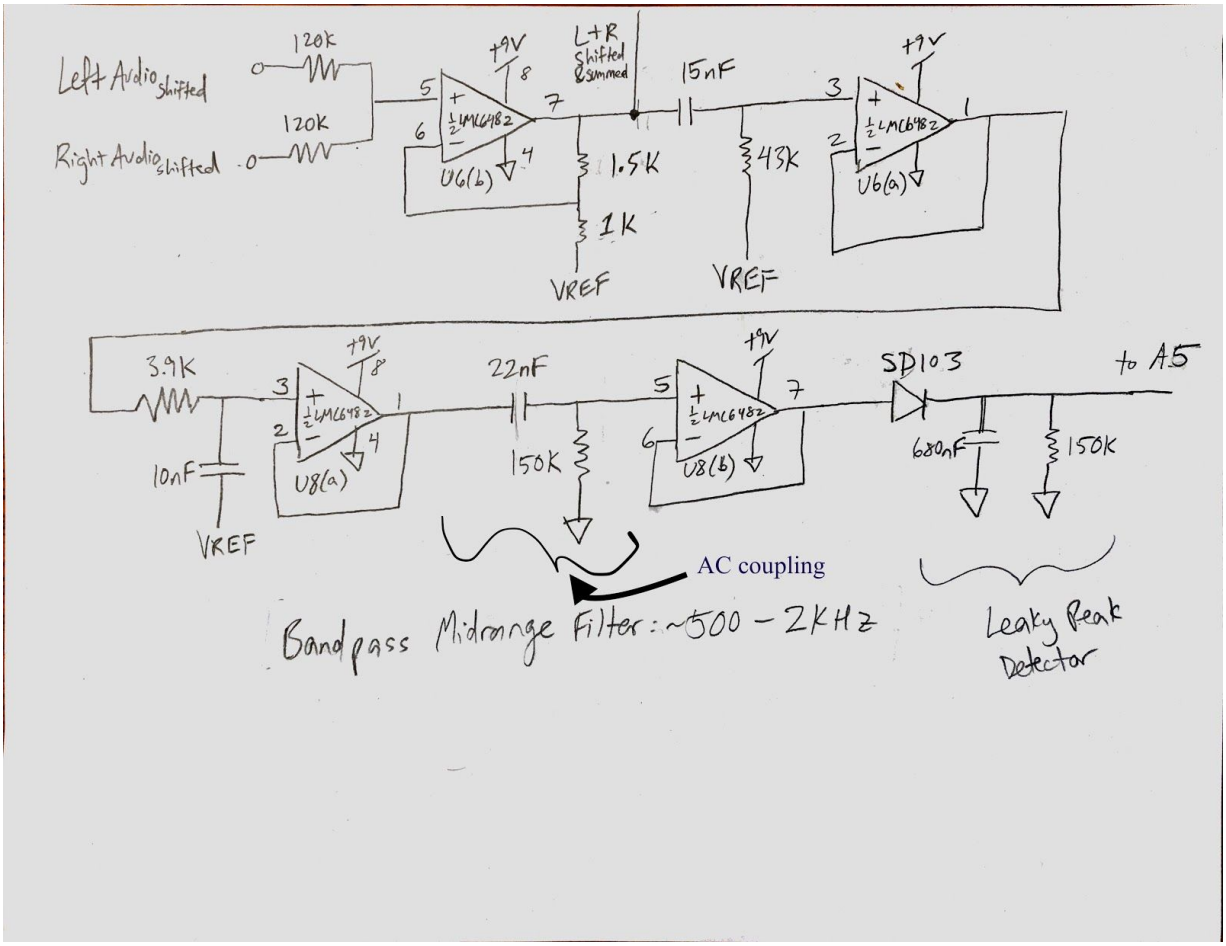


Figure 6.2: Schematic of the 500Hz - 2kHz midrange bandpass filter

B. AC Coupling

Similar to how we handled the passed bass signal, we now want to convert our analog midrange signal to a digital signal to display on another 8-LED array. Whereas we did this previously in hardware using a single-slope ADC, we now handle the midrange signal in software, e.g. by connecting the output of our midrange signal to an analog input pin of an Ardunio µController. From there, we can use the µController to do whatever we want—in this case, mapping the analog level to a series of digital outputs.

However, there's one thing we need to do first. An analog input pin of an Arduino µController only reads voltages between 0 and +5V, yet the voltage of our analog signal ranges between 0V and +9V. For this reason, in order to fit our signal into this range without using a +2.5V secondary pseudo-ground (which would get unnecessarily complicated), we used an AC coupling circuit so as to remove the DC levels of the audio signal. This centered the midrange signal back around 0V, cutting off signals below 0V. This is fine, because as always, we only care about the top half of the signal, as that is where the maximum will always lie. Then, the new "grounded" signal will range from 0 to +4.5V (imagine the top half of the original signal—4.5V to 9V—sitting on ground), which is usable by the µController.

Furthermore, because an AC coupling circuit acts like a high pass filter, we chose a critical frequency f_c of 50Hz because it is ten times less than the lowest frequency of our passed midrange, 500Hz, as specified in the midrange filter (section IV part A) above. We chose this value to be away from our critical frequency because the point of this circuit is to strip DC, not attenuate the signal. Below is the schematic of the AC coupling circuit:

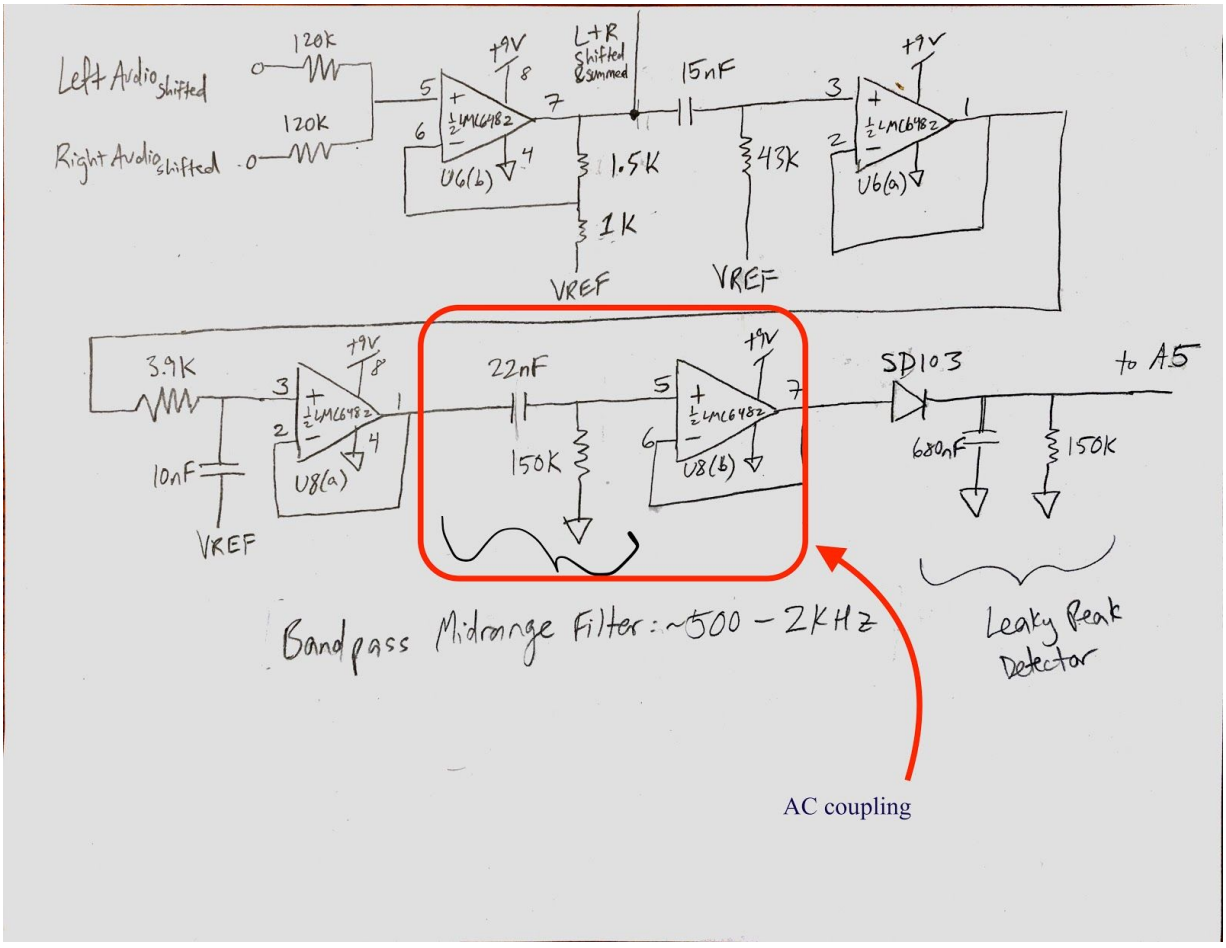


Figure 6.3: AC coupling circuit for the midrange with f_c of 50Hz

C. Leaky Peak Detector

We then decided to average the values of peaks of the midrange signal using a leaky peak detector. To choose the RC values for the leaky peak detector we considered the period of the lowest frequency of the midrange, 500Hz.

$$Period = 1/500Hz = 2\ milliseconds$$

From the above value, we understood that our RC time constant needed to be much greater than 2 milliseconds yet less than the time it takes for the amplitude to decrease to a lower peak value. Based on this constraint, we arbitrarily chose 100 milliseconds. Finally, we used an SD103 diode due to its low forward voltage, minimizing voltage loss. Below is the schematic of the leaky peak detector for the midrange:

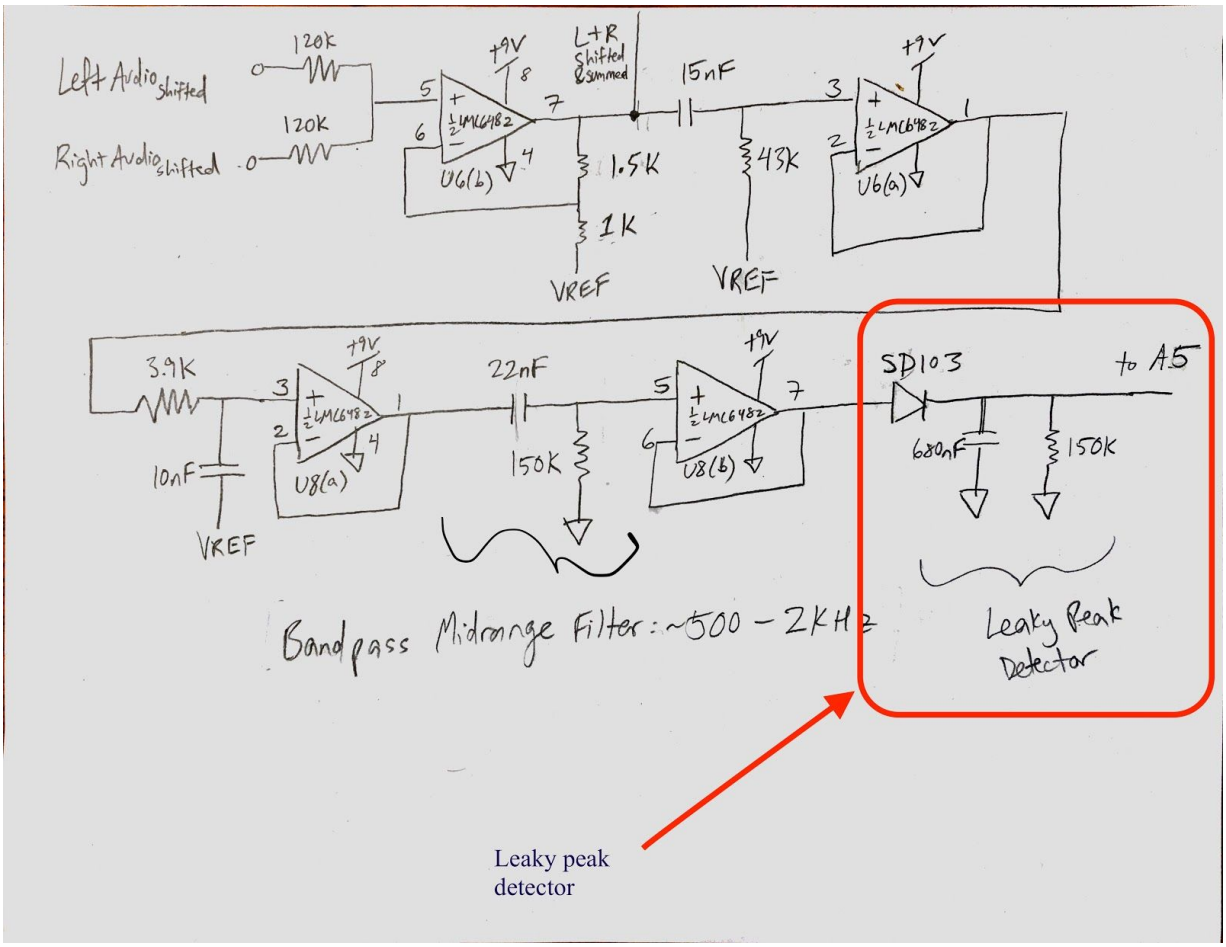


Figure 6.4: Leaky peak detector circuit for midrange with RC time constant of 100mSec

The output of the leaky peak detector is then connected to an analog input pin (pin A5, as denoted in our code) of an Arduino µController.

D. Arduino µController & Shift Register

We are now ready to display the analog level of the midrange on an array of 8 LEDs by using the Arduino µController. We already know that we will be using one input pin (A5). To minimize the number of Arduino output pins that we use, we chose to use a shift register and just one digital output pin. We chose the TLC5916 shift register because it is a constant current driver whose output current can be defined by a resistor connected to its R-EXT pin.

We determined the desired current level by connecting one of the yellow LEDs to the power supply and incrementally adjusting the amount of current provided across the LED, which demonstrated that a current of about 10mA provided sufficient brightness for our purposes. Therefore, our choice of resistor value for R-EXT could be calculated from this I_{out} value and from the formula provided by the datasheet of the TLC5916:

> *R-EXT = (1.25/I_{OUT}) * 15*
> *R-EXT = (1.25/[10E-3 A]) * 15*
> *R-EXT = 1875Ω*
> *Value chosen for R-EXT= 1.8kΩ*
>
> *Note: a slightly smaller R-EXT value than theoretically desired means a slightly higher current than desired, which is fine; 10mA is not an absolute maximum value.*

The TLC5916 also has a serial data out (SDO) pin, which is convenient to our design, because we still need another shift register for the treble display, as we will discuss below in section VII. Below is the schematic of the Arduino µController connected to two TLC5916 shift registers:

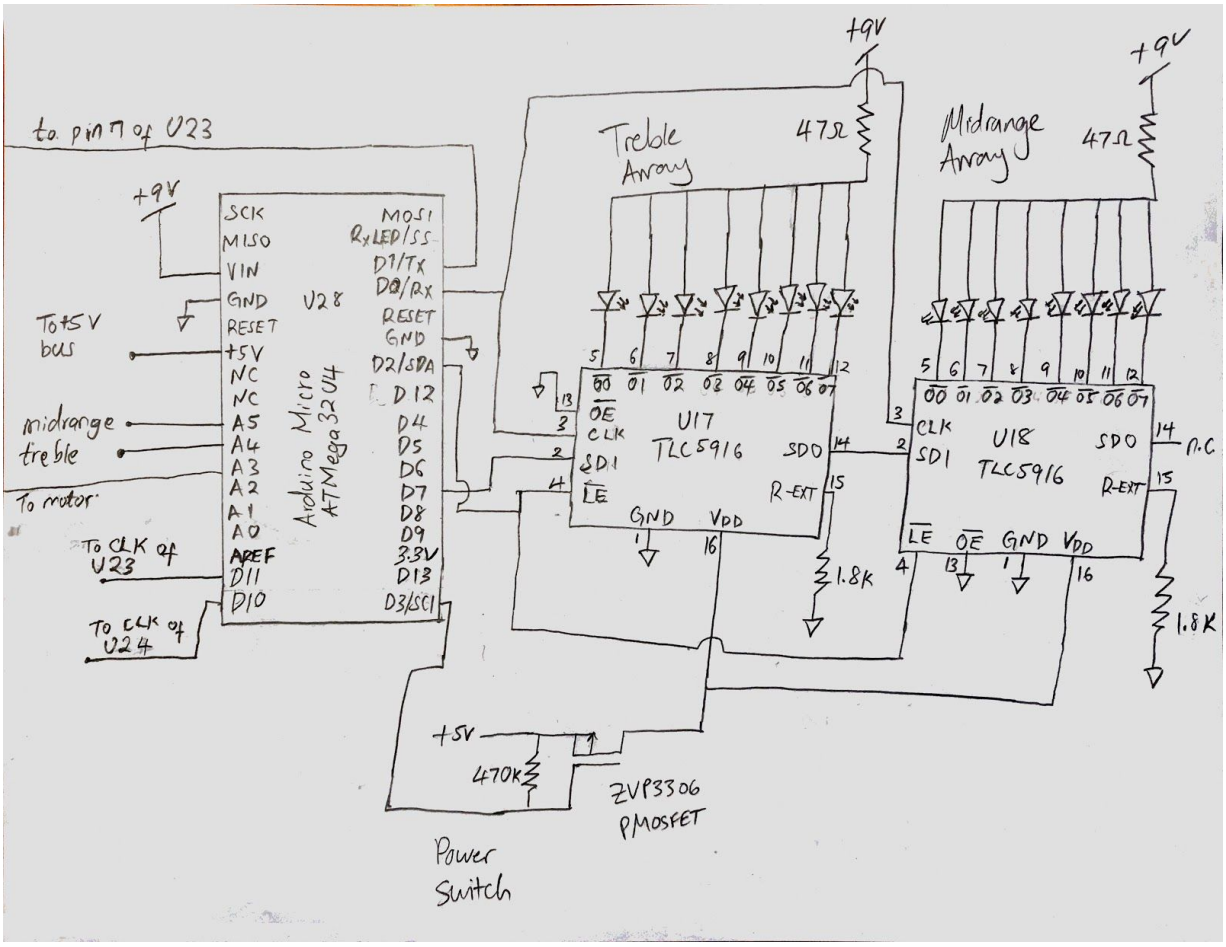


Figure 6.5: Arduino µController connected to two TLC5916 shift registers for the midrange and treble

E. Display Midrange Amplitude on an 8-LED Array

To minimize redundancy, we will postpone discussion of the specific code within the Arduino and the discussion of the midrange 8-LED array until the next section. This is because much of the code involved in the display for the midrange was abstracted and combined with that of the treble level. In addition, the midrange and treble displays are closely linked and require a discussion of code and therefore will be described all together in the next section.

VII. Treble Signal Processing & Visualization

Please note that many of the early subsections of the Treble section are similar to their counterparts in the Midrange section.

A. 4kHz - 6kHz Treble Bandpass Filter

We designed the treble filter with the help of analog.com's Filter Wizard.[5] The filter is designed to take in the original combined L+R signal from way back in section III C and pass frequencies between 4kHz and 6kHz. It should attenuate frequencies higher than 6kHz by about ~dB and attenuate frequencies lower than 4KHz by about ~dB. Below is the schematic generated by analog.com:

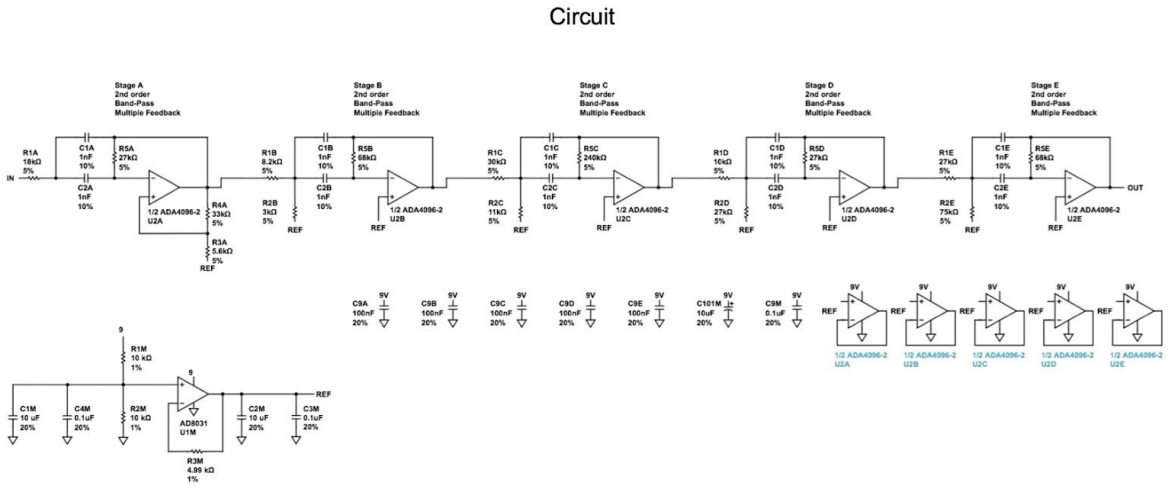


Figure 7.1: Schematic of a 4kHz - 6kHz treble bandpass filter generated by analog.com's Filter Wizard.

Just as before, we adapted the design from analog.com to suit our parts by replacing all the op amps with LMC6482 op amps and connecting the REF terminals to our pseudo-ground (VREF).

[5] http://www.analog.com/designtools/en/filterwizard/

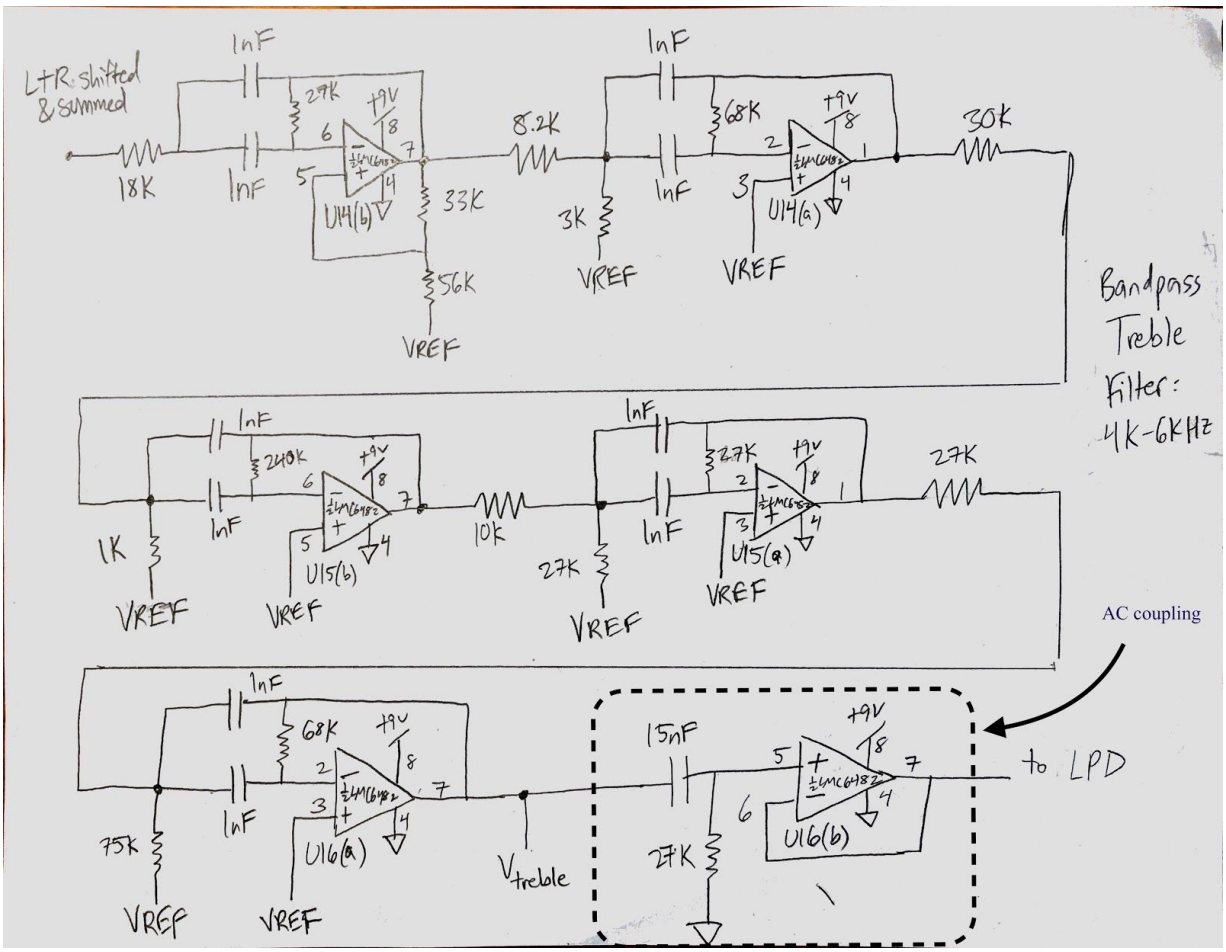


Figure 7.2: Schematic of a 4kHz - 6kHz treble bandpass filter based off of a design from analog.com's filter wizard.

B. AC Coupling

Just like with the midrange, we need to AC couple of the output of the treble filter. Please refer back to the AC Coupling section of the Midrange section (Section V B) for a more detailed description of why we need AC coupling.

In this case, we choose our f_c to be 10 times less than the frequency of the lowest treble frequency (4kHz). Thus our f_c becomes 400Hz to ensure that none of the frequencies between 4kHz and 6kHz are attenuated. Below is the schematic of the AC coupling for the treble:

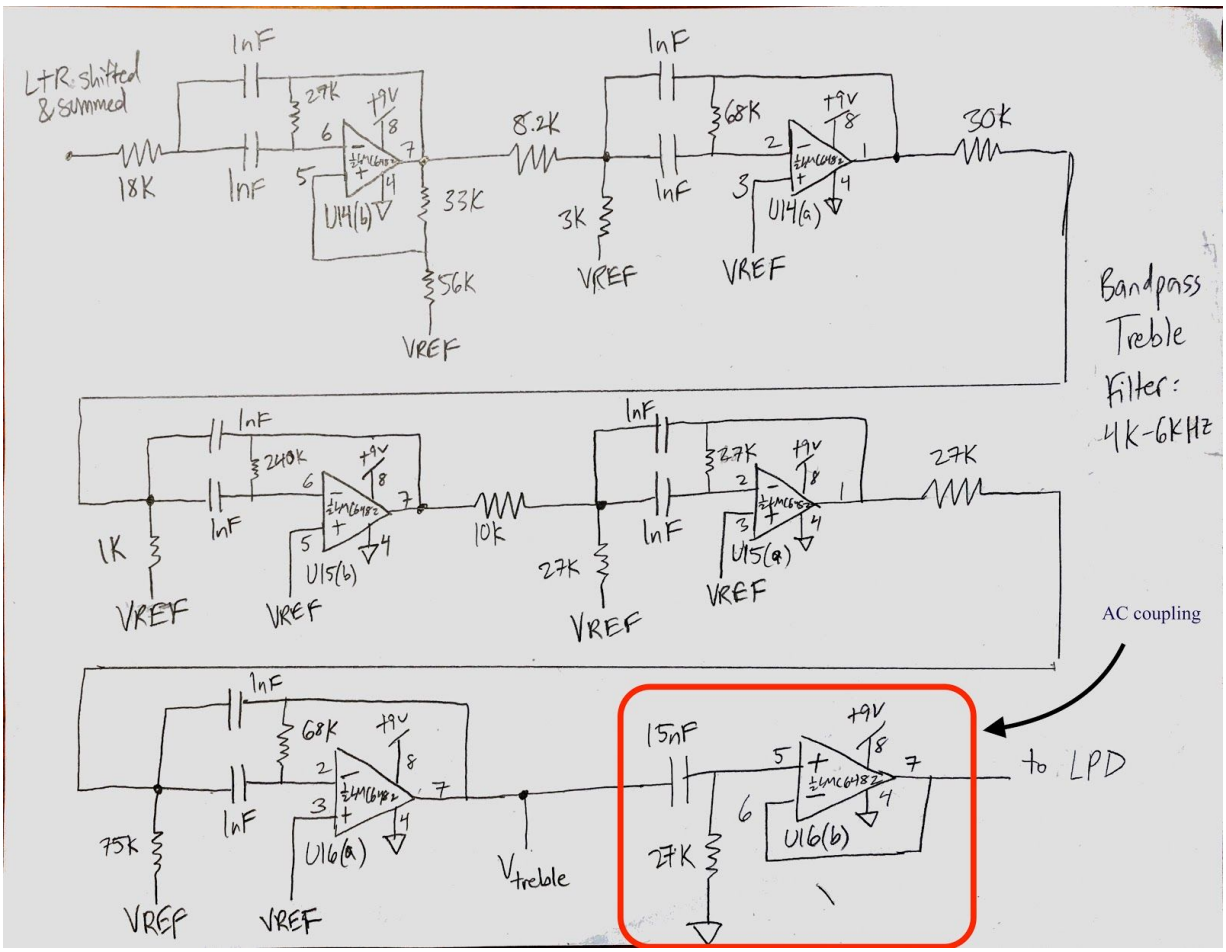


Figure 7.3: AC coupling for the treble (4kHz - 6kHz) with f_c of 400Hz

C. Leaky Peak Detector

Just like with the midrange, we now use a leaky peak detector to average the peak values before connecting the circuit to an input pin of the Arduino µController. We used the same RC time constant of 100 milliseconds because the period of the lowest passed treble frequency (4kHz) is 0.025 milliseconds and much less than the 100 millisecond time constant. Below is the schematic for the treble leaky peak detector:

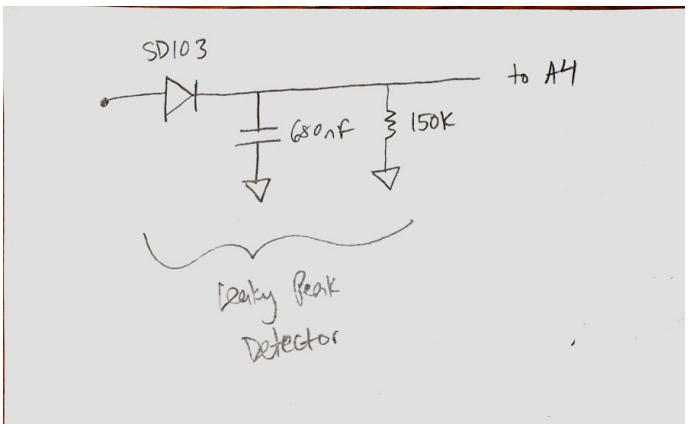


Figure 7.4: Treble leaky peak detector

D. Arduino µController & Shift Register

To minimize the number output pins needed to drive 8 LEDs, the treble display was also implemented using the TLC5916 shift registers. This second shift register was connected to the serial data out (SDO) of the midrange shift register and it shared the same clock pin and latch pin as the first shift register; thus, no additional Arduino output pins were required outside of those attached to the first shift register.

The only difference between the midrange circuit and the treble circuit was the color of the LEDs. Treble used green LEDs while midrange used yellow LEDs. Nonetheless, both TLC5916 used the same 1.8kΩ resistors to drive a constant current of about 10mA.

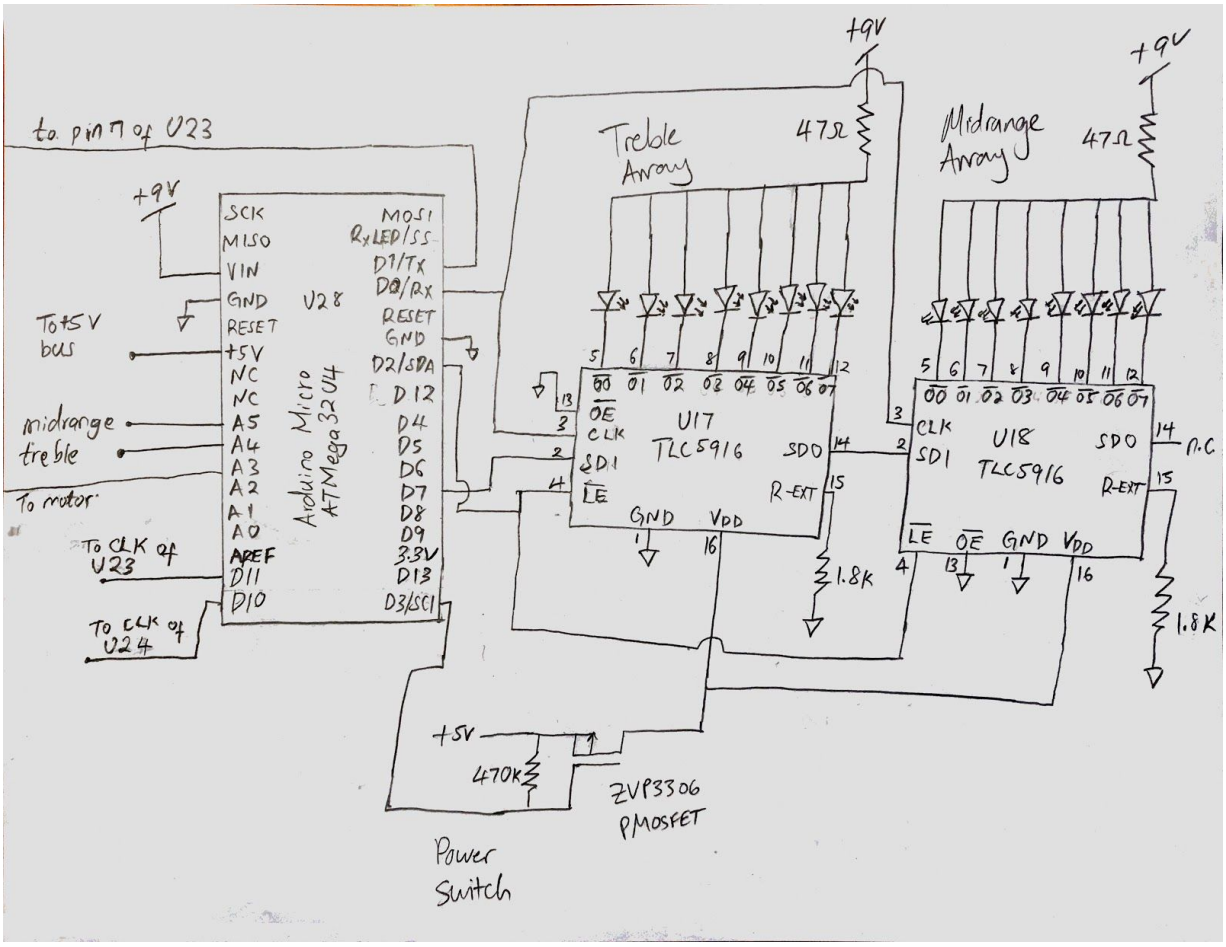


Figure 7.5: Arduino µController connected to TLC5916 shift registers for both treble and midrange. *Note: this is the same as figure 6.5*

E. Display Midrange & Treble Amplitude on an 8-LED Array using Arduino Code

We now dissect, piece by piece, the code in the Arduino that will drive both shift registers and light up both LED arrays.

Note: Before implementing our actual LED-lighting function, we created several smaller global functions, with useful names, for our final function inside `loop()` to use.

Step 1. Mapping the analog inputs from the input pins into digital levels.

Converting analog to digital in Ardunio code is nice because the Arduino software basically does it all for us. This process requires two functions: analogRead() and map(). Thus, we read both our analog midrange signal, which was connected to pin A5, and our analog treble signal, which was connected to pin A4, and pass those into map() separately, which will return a digital value we can use.

First up is using the analogRead() function, which will "read" our midrange and treble signals and return a value between 0 and 1023 corresponding linearly with analog values between 0V and +5V, as per the online Ardunio Reference page[6]. Then, we map that number to a range of 1 to 8, corresponding to how many LEDs we want lit up. For example, 0V would be read as 0, which would be mapped to 1, and only the bottom-most LED would be lit up. Conversely, +5V would be read as 1023, which would be mapped to 8, and all of the LEDs would be lit up.

However, there's one small quick fix to make before moving on: the output signal of the AC coupling for both midrange and treble only ranges from 0 to +4.5V instead of the expected 0 to +5V. To ensure that we're including the highest LED, we tweak our mapping values. [0 - 5] → [0 - 1023] → [1 - 8] now becomes [0 - 4.5] → [0 - ?] → [1 - 8], where we solve for the question mark by proportionally reducing the highest input mapping value:

$$5/1023 = 4.5/?$$
$$? = (4.5/5) * 1023 \approx 920$$

Now an input voltage of +4.5V, corresponding to 920, maps out to 8 and lights up all the LEDs, as desired.

We made two separate small global functions to do the above:

```
//function that maps midrange input
int midrangeLevel(){
  int amplitude = analogRead(MIDRANGE_INPUT);
  amplitude = map(amplitude, 0, 920, 1, 8);
  return amplitude;
}

//function that maps treble input
int trebleLevel(){
  int amplitude = analogRead(TREBLE_INPUT);
  amplitude = map(amplitude, 0, 920, 1, 8);
  return amplitude;
```

We now create another small function called `risingEdge()` that just takes in a clock pin number and simply creates a rising edge on that clock pin. We will later use this to seamlessly shift in data into the shift registers by calling `risingEdge()` rather than coding two `digitalWrite()` lines every single time.

```
//Rising edge for for any clock pin
```

[6] https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/

```
void risingEdge(int clockPin){
    digitalWrite(clockPin, LOW);     // falling edge
    digitalWrite(clockPin, HIGH);    // rising edge
}
```

Step 2. Writing a function to shift out the proper data to the corresponding 8-LED arrays.

We then leveraged `risingEdge()` to create a function called `turnLEDOn()` that takes two integer parameters—the midrange 1-8 level and the treble 1-8 level—and then lights up *both arrays of LEDs through the shift registers at once*.

For example, if `LEDNumMidrange` is 5 and `LEDNumTreble` is 2, we pass those numbers into `turnLEDOn()`, which shifts 5 HIGH (on LEDs) and then 3 LOW (off LEDs) into the midrange array, and then shifts 2 HIGH (on LEDs) and then 6 LOW (off LEDs) into the treble array, all in one fell swoop:

```
//function that lights up 8 midrange LEDs and 8 treble LEDs
void turnLEDOn(int LEDNumMidrange, int LEDNumTreble){


      //shift in value of midrange
      digitalWrite(DATA_PIN, HIGH);    //data high
      for(int i = 0; i<LEDNumMidrange; i++){
        risingEdge(CLK_PIN);
      }

      //shift in zeros for unreached level of midrange
      digitalWrite(DATA_PIN, LOW);    //data low
      for(int i = 0; i<(8-LEDNumMidrange); i++){
        risingEdge(CLK_PIN);
      }

      //shift in value of treble
      digitalWrite(DATA_PIN, HIGH);     //data high
      for(int i = 0; i<LEDNumTreble; i++){
        risingEdge(CLK_PIN);
      }

      //shift in zeros for unreached level of treble
      digitalWrite(DATA_PIN, LOW);    //data low
      for(int i = 0; i<(8-LEDNumTreble); i++){
        risingEdge(CLK_PIN);
      }
}
```

We then called `turnLEDOn()` inside `loop()` and passed in the corresponding amplitude levels of midrange and treble from the `midrangeLevel()` function and the `trebleLevel()` function, which constantly monitor the A4 and A5 analog input pins:

```
//display midrange and treble level on array of LEDs
   turnLEDOn(midrangeLevel(), trebleLevel());
```

The midrange and treble arrays of 8 LEDs each then successfully simulated the variation in amplitude of the midrange and treble frequencies of our music signal by "waving" up and down in real-time along with our music.

iii. Challenge faced: rapid flickering of LEDs due to data being shifted in rapidly

We noticed our unlit LEDs would flicker rapidly. We figured this was due to data being shifted into the shift registers at a rapid rate. Therefore, we incorporated a latch inside `loop()` to slow down the rate of flickering. In the code below `LEDDisplayTime` is declared as `const unsigned long HapticFeedbackTime = 200;`

```
//decrease amount of flickering by latching value for LEDDisplayTime
milliseconds
  if(digitalRead(LATCH_PIN) == HIGH){  //if not latched
    LatchStart = millis();              //set time when latched
  }

  digitalWrite(LATCH_PIN, LOW); //activate latch(active low)

//only deactivate latch when LEDDisplayTime has elapsed
  if(millis() > LatchStart + LEDDisplayTime){
    digitalWrite(LATCH_PIN, HIGH); //deactivate latch
  }
```

We implemented the above code in this manner instead of using delay() so as to to keep other processes from being halted.

VIII. Bass Hit Counter, Numitron Display, & Haptic Feedback

The next part of our circuit is entirely based (pun intended) off the bass signal. Using the bass signal alone, we will drive a counter that counts how many times the bass goes beyond a certain threshold, display that number on a Numitron, and drive a motor everytime the bass goes above that threshold.

A. Comparator with Adjustable Hysteresis

To detect the bass beats, we built a comparator to compare the amplitude of the bass against a constant threshold voltage with adjustable hysteresis. We used hysteresis so as to minimize the amount of signal bouncing that would occur as the input bass waveform approached the threshold voltage due to noise. We used a potentiometer to provide us with a range of adjustability to experimentally find a "sweet spot" of hysteresis that provided us with a reasonable amount of detection.

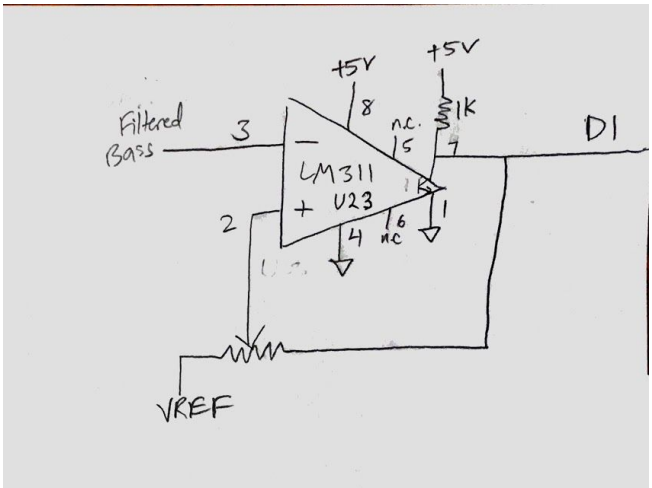


Figure 8.1: Schematic for bass hit detector with variable hysteresis

B. Ardunio µController, Shift Registers, & Seven-Segment Numitron Displays

Now that we have a digital signal (either 0V or +5V from the comparator) based on whenever the bass crosses a predefined threshold, we can now use the Arduino µController to 1) count the bass and 2) drive the Numitron. Before explaining how the bass counter was implemented in code, we first establish how the display circuit has been designed.

The display circuit involved two 4-bit counters. Each clock pin of the two counters was connected to a separate Arduino digital output pin. The reset pins of the two counters were then shorted and connected to a single Arduino digital output pin; this allowed the counts of the two counters to be controlled individually and reset at the same time. We wanted to be able to reset immediately; therefore, we used the 74HC161 counters, which have an asynchronous clear. These two counters were then connected to two separate 3-bit-to-7 segment decoders (LS247). These decoders were then connected to two separate Numitron displays. Below is a schematic of the display circuit:

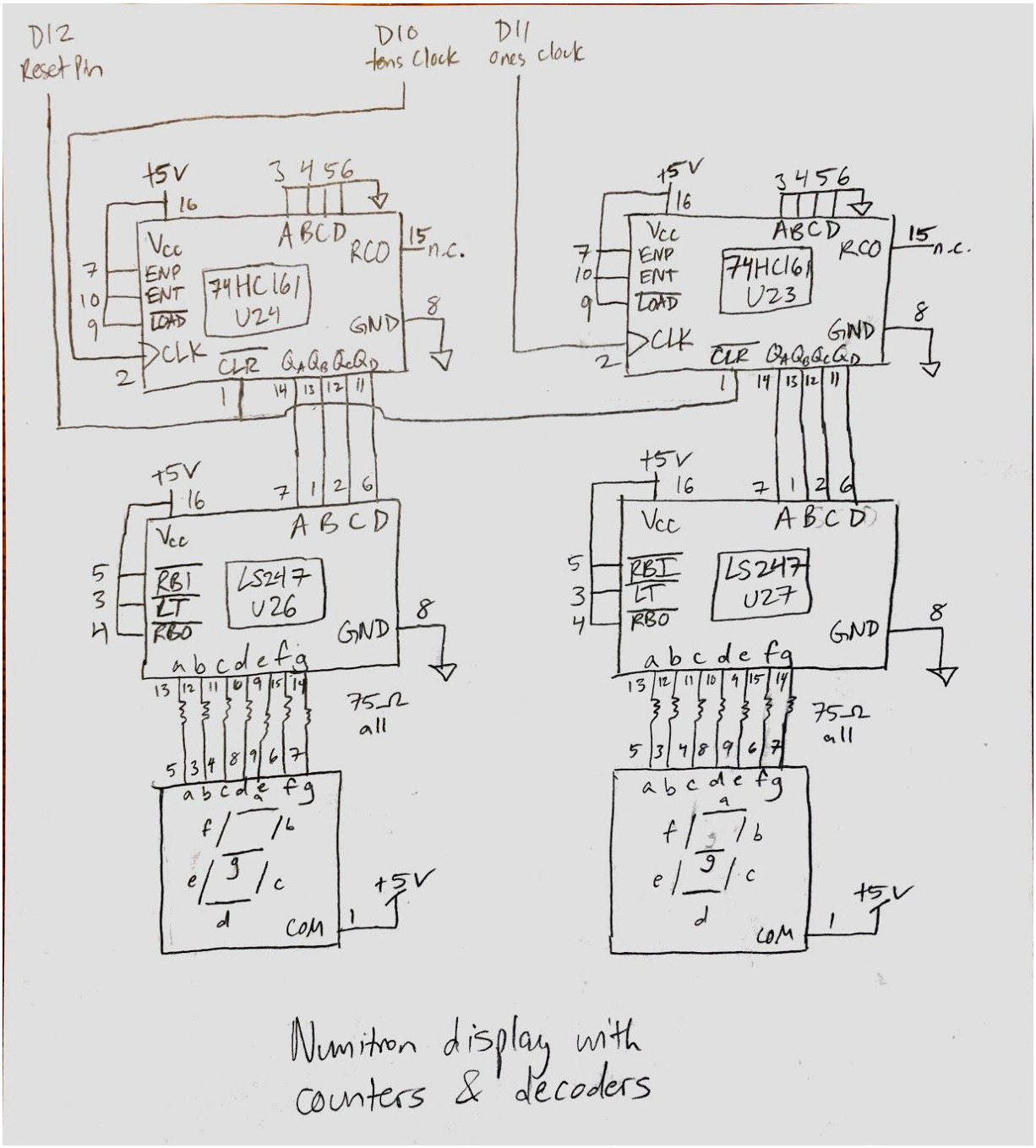


Figure 8.2: Schematic for the bass beat counter display

Now we examine the code used in the Arduino to count the bass beats:

Step 1. Counting beats using an Interrupt Service Routine

We implement our counter for the bass hits using an Interrupt Service Routine (ISR) because it is faster than `digitalWrite()`. We start by naming our boolean ISR flag `BeatDetected`, which is initially set to be `false`, and change its value from `false` to `true` whenever a rising edge is detected from the bass input pin (which we have called the `BASS_PIN`). We then initialize a new variable called `BeatCntr` to zero in `setup()` and constantly monitor the flag, `BeatDetected`, in `loop()`. Whenever the flag is set to true, i.e. a beat is detected, the beat counter `BeatCntr` increments by one in the loop. The flag is then reset to `false` and `loop()` goes back to constantly monitoring the `BeatDetected` flag. Below is the code showing how we attach the ISR and how we reset the beat counter in `setup()`:

```
//attach interrupt to Bass input pin
attachInterrupt(digitalPinToInterrupt(BASS_PIN), setISRFlag, RISING);

BeatCntr = 0; //initiate variable to hold number of counts
```

Function that sets ISR flag:

```
//ISR function
void setISRFlag(){
  BeatDetected = true;
}
```

Function that is inside `loop()` to increment `BeatCntr` and to clear the ISR flag:

```
//increment beat counter if beat is detected
  if(BeatDetected){
    BeatCntr = BeatCntr + 1;           //increment beat counter

    //clear ISR flag
    BeatDetected = false;
  }
```

Step 2. Displaying the beats on the Numitron display

After we've established our `BeatCntr` variable that tells us the number of bass hits since the start of the song, we now need to put that number onto the Numitron displays. As above, to simplify our final function, we first code several smaller global functions.

We will be sending rising edges to the counters, which then increase the values that we see displayed on the Numitron displays. Therefore, the same function `risingEdge()` that we used in the shift registers is applicable here:

```
//Rising edge for any clock pin
void risingEdge(int clockPin){
    digitalWrite(clockPin, LOW);      // falling edge
    digitalWrite(clockPin, HIGH);     // rising edge
}
```

To reset the displays, we implemented a function that asserts the clear pins of the two counters:

```
//function to reset the numitron displays
void resetNumitronDisplay(void){
  risingEdge(NUMITRON_RESET_PIN);
}
```

Our two Numitron displays will show the tens digit (on the left) and the ones digit (on the right). Therefore, we implement two separate functions that take in an integer and returns the ones place and the tens place, respectively. For example, given an integer of "38", the `onesDigitFunc()` will return "8" and the `tensDigitFunc()` will return "3":

```
//function that returns the the ones digits of a number
int onesDigitFunc(int x){
  return (x%10);
}

//function that returns the tens digit of a number
int tensDigitFunc(int y){
  int subtractOnesValue = (y - onesDigitFunc(y));
  int truncated = (subtractOnesValue)/10;
  return onesDigitFunc(truncated);
}
```

We then write a function that takes in two integers, which we call "`onesDigit`" and "`tensDigit`" for usability, and displays those digits on the corresponding Numitron displays. It does this by causing a particular number of rising edges on the corresponding clock pins:

```
//function that causes multiple clocks cycles to display a number on the
numitron displays
void displayDigit(int onesDigit, int tensDigit){
  //clear display
  resetNumitronDisplay();

  //display ones digit
  for(int i=0; i<onesDigit; i++){
    risingEdge(ONES_CLK_PIN);
```

```
  }
  //display tens digit
  for(int i=0; i<tensDigit; i++){
    risingEdge(TENS_CLK_PIN);
  }
```

With these functions, we have all the tools we need to display the counted beat hits. We do so with just one functional call to `displayDigit()`, which we pass the ones and tens digit of `BeatCntr` into as parameters:

```
//display beat counts on the numitron display
  displayDigit(onesDigitFunc(BeatCntr), tensDigitFunc(BeatCntr));
```

C. Motor Haptic Feedback

Along with counting every bass hit, we want to drive a motor that will vibrate the board every time the bass level exceeds a certain threshold. This is the tactile element of our project combining audio, visuals, and tactile elements. We do this by taking in the same signal that the bass counter circuit used and using it in a much simpler way.

The hardware part of the haptic feedback involves a 3V, 75mA DC motor connected between one analog output pin of the Arduino µController and a current-limiting resistor connected to ground. A flyback diode is connected across the motor and resistor to clamp negative voltages and protect the rest of the circuit:

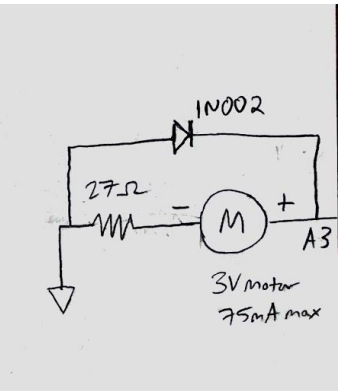


Figure 8.1: Schematic for the motor that provides haptic feedback.

The software part uses the ISR flag of the beat counter to initiate the haptic feedback. This is done by recording the time when the ISR flag is toggled before clearing the flag:

```
//increment beat counter if beat is detected
  if(BeatDetected){
    BeatCntr = BeatCntr + 1;           //increment beat counter

    //store time when haptic feedback starts
    StartHapticFeedback = millis();

    //clear ISR flag
    BeatDetected = false;
  }
```

The motor is then activated for a set time period defined by a user defined constant `HapticFeedbackTime`.

The motor stays activated until HapticFeedbackTime elapses:

```
//keep haptic feedback active until HapticFeedbackTime elapses
  if(StartHapticFeedback + HapticFeedbackTime > millis()){
     analogWrite(MOTOR_PIN, 255); //make motor vibrate at max
  }else{
    analogWrite(MOTOR_PIN, 0); //turn off motor
  }
```

Haptic feedback is implemented in the manner shown above to allow for the code to run without stopping to wait for a delay; thus, it does not prevent other lines of code from running.

IX. Bass-Boosted Audio Output

A. Attenuation of Variable-Gain Bass & Original Audio Signal

Because everyone loves bass, we want the user to be able to manually control the level of the song's bass they hear. The bass-boosting design of our project involves combining the amplified variable gain bass with the original audio signal such that at their highest amplitudes, their ratio should be 3:1 of bass:original audio.

Combining the two requires the use of a summing op amp. However, the voltage of the bass at its max volume is 9V peak-to-peak while the voltage of the original signal after being level shifted, summed, and amplified is also 9V peak-to-peak at its max value. If the two signals are summed as is, they will far exceed the output swing voltage of the LMC6482 at their max values and definitely be clipped.

Therefore, to keep them from exceeding our V_{cc} and V_{ee} of +9V and 0V, respectively, we have to attenuate both of them such that when summed, they still fit within the swing voltage of the LMC6482 while maintaining their ratio of 3:1 for bass:original audio.

Also, since we plan to drive the sum through a speaker, we need to ensure that one signal is not inverted with respect to the other. Adding the same signal to its inversion will cancel it out.

To begin, we attenuate the the bass by ¾ using an op amp inverting configuration. In section V B, we built a variable gain amplifier that already inverted the bass; thus, this operation inverts it back. We then proceed to attenuate the original signal by ¼ using an op amp in an inverting configuration. This inverts the original signal with respect to the bass. To fix this inversion, we invert the original signal back again using a unity gain inverting op amp configuration.

B. Summation of Bass & Original Audio with Volume Control

After this, we can proceed to sum the left and the right to produce a bass boosted output signal. We also decide to add a variable gain control at this point so as to control the volume of the output. For this reason we use an inverting op amp with a potentiometer, which allows us to vary the gain between 0 and 1:

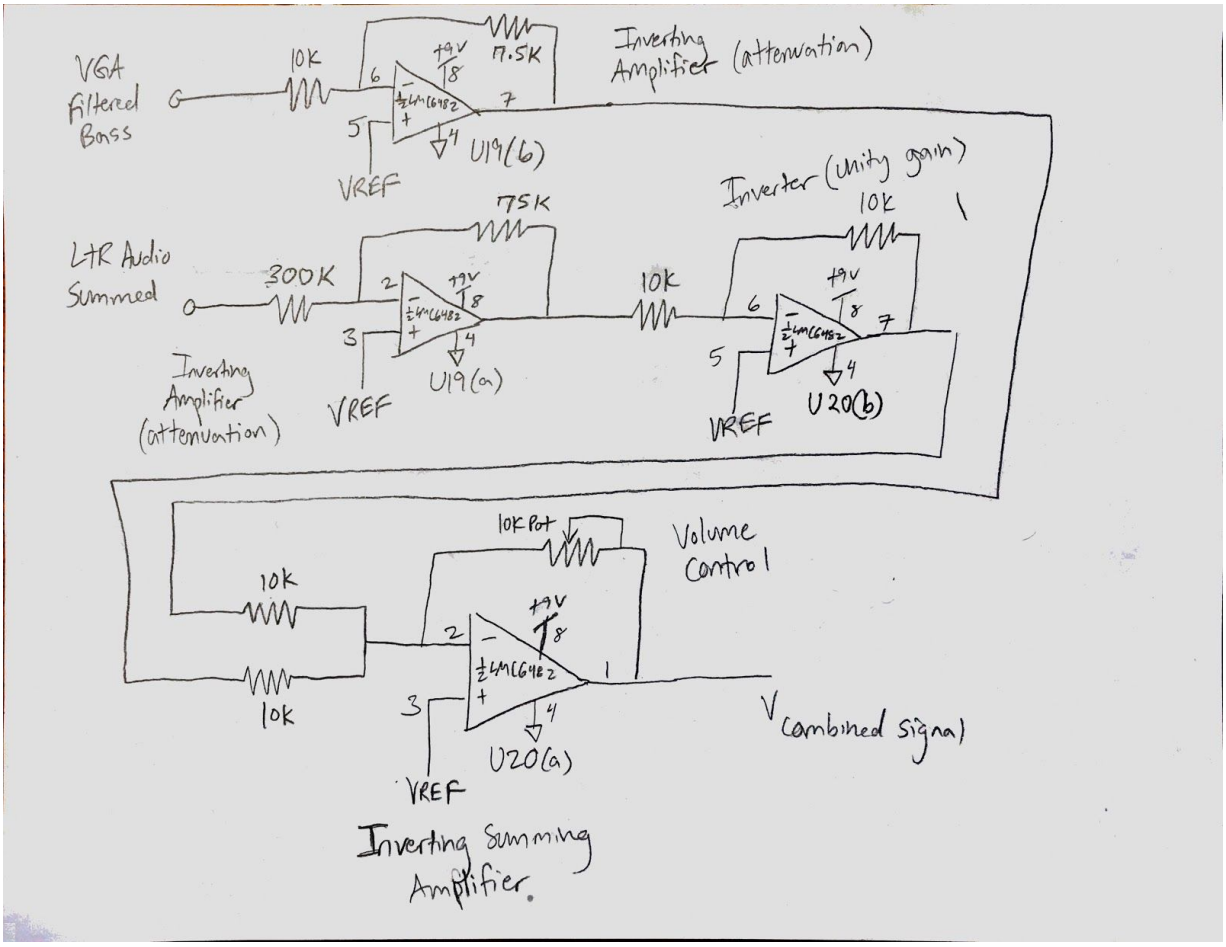


Figure 9.1: Attenuation and summation of the bass and and the original signal to arrive at a ratio of 3:1.

C. Class B Push-Pull Amplifier to Drive Output Speaker Audio

Finally, we want the user to be able to hear the song they plugged in with the new bass-boosted addition. To do this, we need to amplify the current so as to drive the speaker to a sufficient volume using a class B push-pull amplifier.

To ensure that the voltage across the speaker is zero when there is no music—the quiescent DC voltage would otherwise create a constant tone even when nothing is plugged in—we initially planned to connect the negative input of the speaker to pseudo-ground (VREF), but after talking to our ES52 instructor, we got a great suggestion to connect the negative input of the speaker to an the output of the same audio signal, *but inverted and amplified through another push-pull amplifier*. This configuration increases the output volume of the speaker even more. Below is a schematic of the configuration:

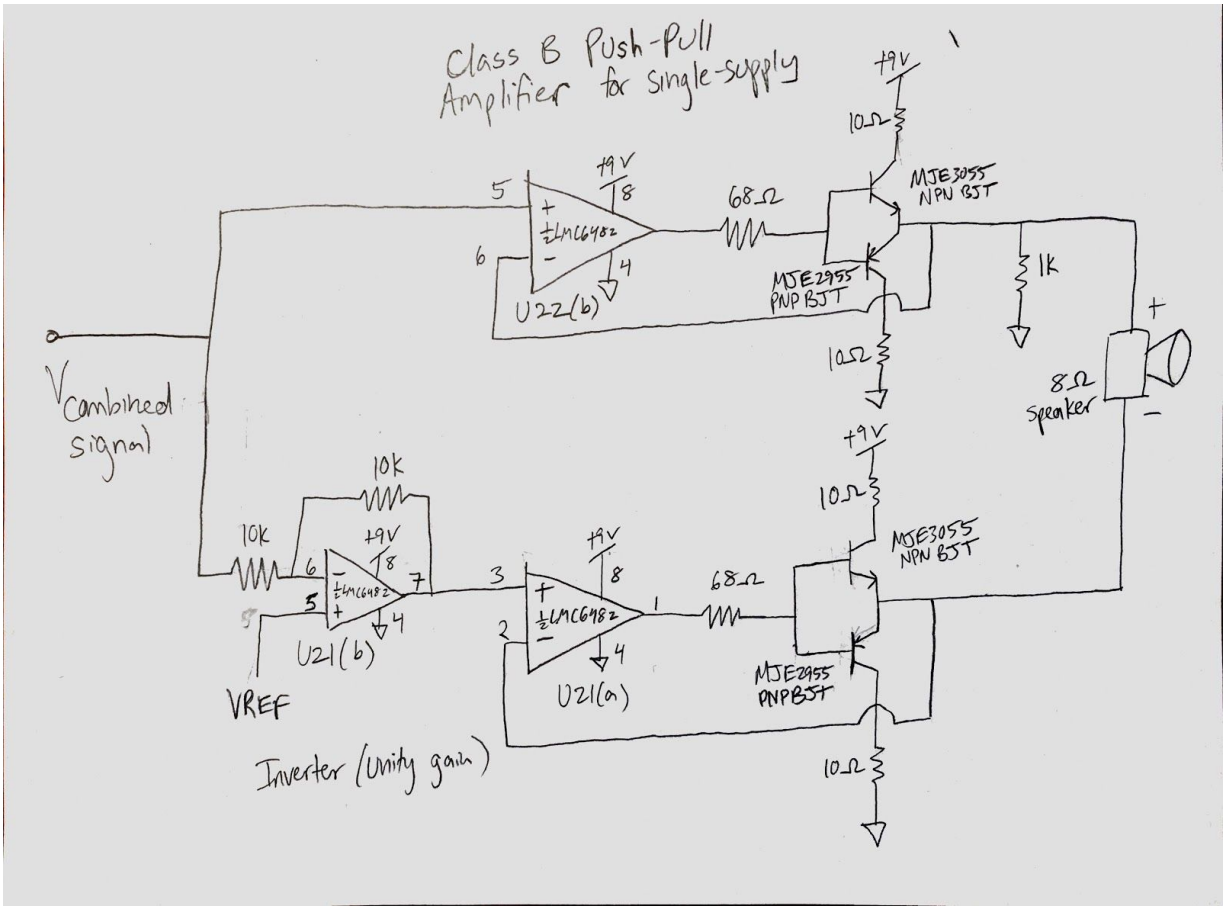


Figure 9.2: Driving a speaker using two class B push-pull amplifiers.

3. Conclusion

A. Our Results & Areas of Improvement

We enjoyed working on this project and believe the final result was a full achievement of what we set out to do. Though not perfect, we implemented every feature from our project proposal to varying extents, and were able to get working versions of all of them together. We made a few mistakes along the way and we learnt the following things:

1. When building a bandpass filter using analog.com, the values inserted should not correspond to the critical frequencies; rather, they should correspond to the pass frequencies. The quality of our midrange filter suffered due to this mistake—it was much too wide and did not attenuate quickly enough at the edges.

   For example, our midrange bandpass filter still couldn't even filter out a 16kHz frequency, which should've been nearly completely gone:

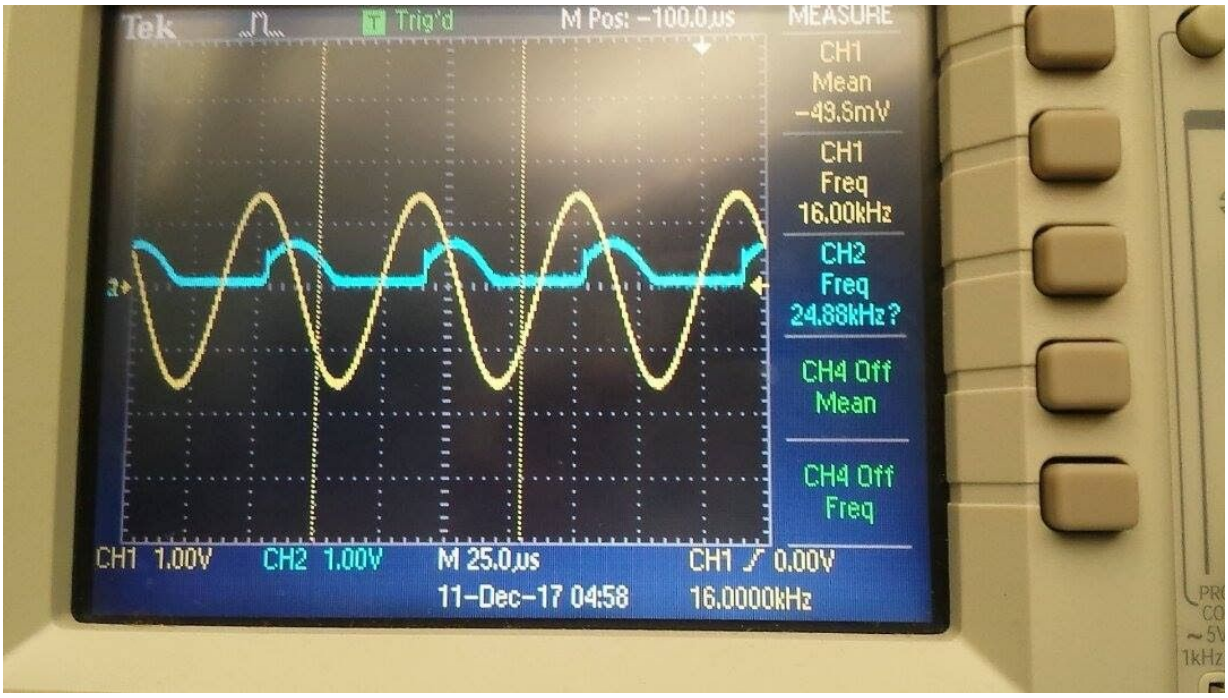


Figure A: A 16kHz sine wave signal still made it through our designed theoretical 2kHz - 4kHz midrange filter. Yellow is the input audio signal and blue is the output from our midrange filter.

2. Single-slope ADCs do not have the same accuracy as double-slope ADCs, which we did not explore in this project. Our single-slope ADC, which we used for our bass signal,

rarely ever reached the top two LEDs. Still, however, our single-slope ADC converted analog signals into digital levels of 1 through 6 quite nicely:

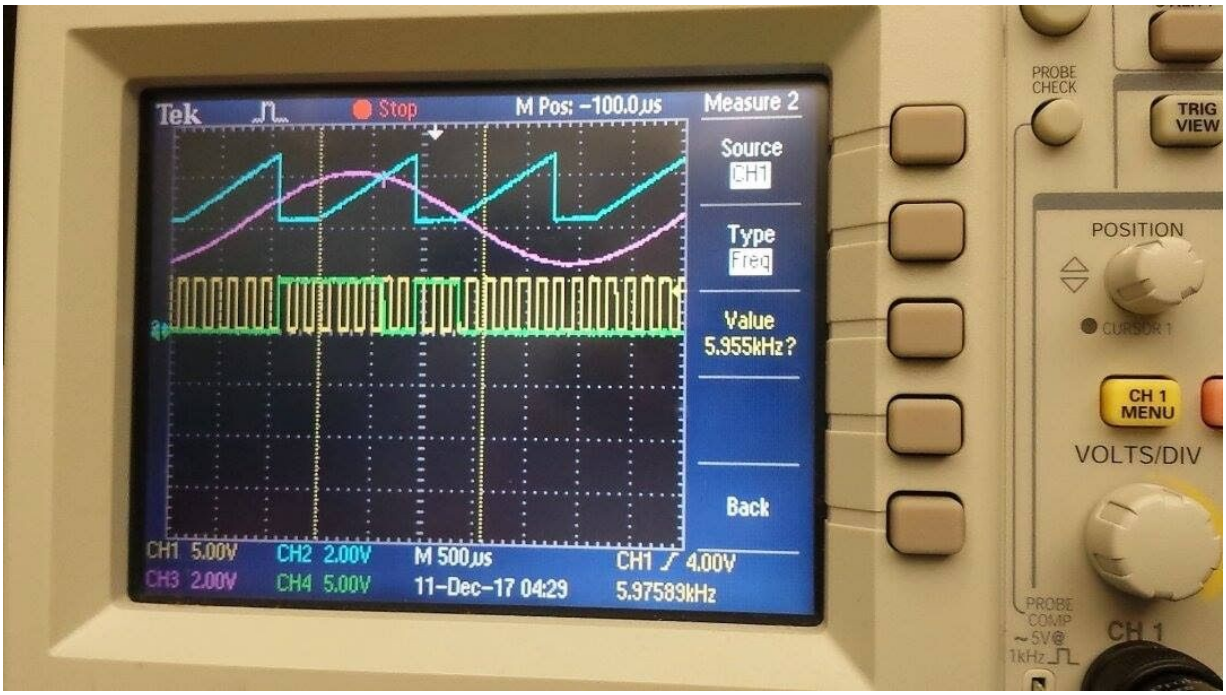


Figure B: Our single-slope ADC. Yellow is the 6kHz clock, blue is the integrator output, purple is the input bass waveform, and green is the comparator output. It is visible that the comparator output stays high for between 1 and 6 cycles, depending on how long the bass waveform stays higher than the integrated slope. That length of time is then fed into the shift register and goes into how many LEDs are lit, as per the report.

3. Separating the entire digital and analog parts of the project, and only having them connect at one point, will reduce the amount of noise leaking into one another. Our output audio constantly had a background signal of around ~750Hz that we believe to be a product of our 750Hz clock leaking into the analog audio output.

B. Things We'd Do Given More Time

1. We'd firstly like to isolate the source of the 750Hz leaking signal and then get rid of it. We would do this by either implementing a sharp filter that filtered out 750Hz (though this would affect the music too) or redesigning our entire project such that digital and analog only touch at one point.
2. We would like to explore using a double-slope ADC in place of our single-slope ADC to get a better digital version of our bass signal. In addition, instead of sampling the upper half of the bass signal as we did in the report, we could have AC coupled circuit to shift

the DC levels of the audio signal to 0V and then used a leaky peak detector to average the amplitudes like we did in the midrange/treble, which yielded good results.

3. Our Numitron display only counts up to 99 and then resets to zero after the "real" count goes past 100. We would like to implement, in code, a beats *per time* variable that average bass drops per minute to display. This would prevent the display from going "up" indefinitely and provide more intrigue as to how often your song has bass beats in it. We tried doing this, but the Numitrons couldn't display a stable number and we didn't have time to debug it, so we reverted to doing a simple counter.
4. We would like to redo our midrange filter so it passes the correct range of frequencies.

4. Bibliography

See all footnotes.

https://www.teachmeaudio.com/mixing/techniques/audio-spectrum
https://www.ifixit.com/Answers/View/97418/what%27s+the+iPhone+3.5+mm+output+jack+power+supply+output
https://electronics.stackexchange.com/questions/37926/what-is-the-typical-max-voltage-out-of-a-pc-speaker-jack
http://www.analog.com/designtools/en/filterwizard/
https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/
sub.allaboutcircuits.com/images/04268.png (single-slope ADC picture)

5. Appendix

I. Code for the entire program:

```
/************************************************
  Program that displays level of midrange and treble levels
  on two arrays of 8 LEDs using TLC5916 shift registers and
  constant current drivers.

  This program also uses an ISR to detect bass input
  and to increment the value on a numitron display. The
  program also causes haptic feedback when bass
  is detected


  created by Billy Koech and Bryan Hu
  creation date: 12/3/2017

  mru:12/4/2017 added midrange input
    12/4/2017 added treble input
    12/5/2017 added code for numitron display
    12/6/2017 added haptic feedback
    All changes by Billy Koech and Bryan Hu
*****************************************/

//Declaration of pin numbers
#define CLK_PIN 0
#define DATA_PIN 7
#define LATCH_PIN 2
#define DISPLAY_SWITCH 3
#define MIDRANGE_INPUT A5
#define TREBLE_INPUT A4
#define BASS_PIN 1
#define ONES_CLK_PIN 11
#define TENS_CLK_PIN 10
#define NUMITRON_RESET_PIN 12
#define MOTOR_PIN A3


//variables
int BeatCntr; //variable to hold number of counts
volatile boolean BeatDetected; //ISR flag
unsigned long LatchStart; //variable to keep time when latch is
initiatated
const int LEDDisplayTime = 50; //variable to delay midrange and treble
to prevent flickering
```

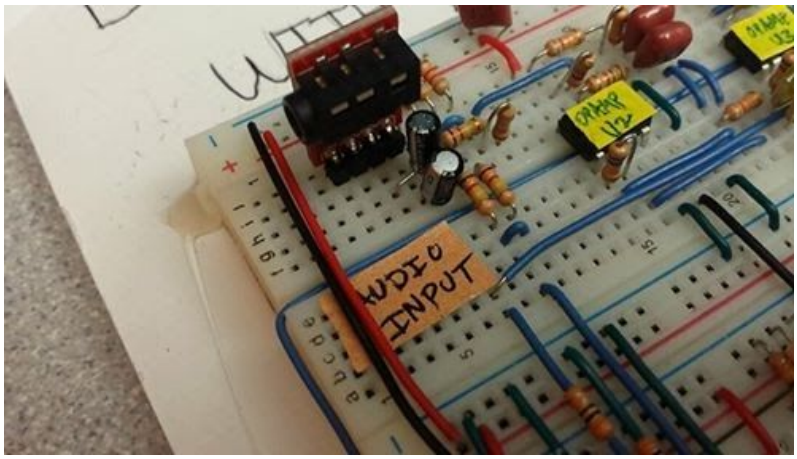
```
const bool debugMode = false;
unsigned long StartHapticFeedback;
const unsigned long HapticFeedbackTime = 200; //length of haptic
feedback


//Rising edge for any clock pin
void risingEdge(int clockPin) {
  digitalWrite(clockPin, LOW);     // falling edge
  digitalWrite(clockPin, HIGH);    // rising edge
}

//function that causes multiple clocks cycles to display a number on the
numitron display
void displayDigit(int onesDigit, int tensDigit) {
  //clear display
  resetNumitronDisplay();

  //display ones digit
  for (int i = 0; i < onesDigit; i++) {
    risingEdge(ONES_CLK_PIN);
  }
  //display tens digit
  for (int i = 0; i < tensDigit; i++) {
    risingEdge(TENS_CLK_PIN);
  }

}

//function to reset the numitron displays
void resetNumitronDisplay(void) {
  risingEdge(NUMITRON_RESET_PIN);
}

//function that returns the the ones digits of a number
int onesDigitFunc(int x) {
  return (x % 10);
}

//function that returns the tens digit of a number
int tensDigitFunc(int y) {
  int subtractOnesValue = (y - onesDigitFunc(y));
  int truncated = (subtractOnesValue) / 10;
  return onesDigitFunc(truncated);
}


//function that lights up 8 midrange LEDs and 8 treble LEDs
void turnLEDOn(int LEDNumMidrange, int LEDNumTreble) {
```

```
  //shift in value of midrange
  digitalWrite(DATA_PIN, HIGH);    //data high
  for (int i = 0; i < LEDNumMidrange; i++) {
    risingEdge(CLK_PIN);
  }

  //shift in zeros for unreached level of midrange
  digitalWrite(DATA_PIN, LOW);    //data low
  for (int i = 0; i < (8 - LEDNumMidrange); i++) {
    risingEdge(CLK_PIN);
  }

  //shift in value of treble
  digitalWrite(DATA_PIN, HIGH);     //data high
  for (int i = 0; i < LEDNumTreble; i++) {
    risingEdge(CLK_PIN);
  }

  //shift in zeros for unreached level of treble
  digitalWrite(DATA_PIN, LOW);    //data low
  for (int i = 0; i < (8 - LEDNumTreble); i++) {
    risingEdge(CLK_PIN);
  }
}


void turnAllLEDOff() {
  //turn everything else off
  digitalWrite(DATA_PIN, LOW);    //data high
  for (int i = 0; i < 16; i++) {
    risingEdge(CLK_PIN);
  }
}
//function that maps midrange input
int midrangeLevel() {
  int amplitude = analogRead(MIDRANGE_INPUT);
  amplitude = map(amplitude, 0, 920, 1, 8);
  return amplitude;
}

//function that maps treble input
int trebleLevel() {
  int amplitude = analogRead(TREBLE_INPUT);
  amplitude = map(amplitude, 0, 920, 1, 8);
  return amplitude;
}
```

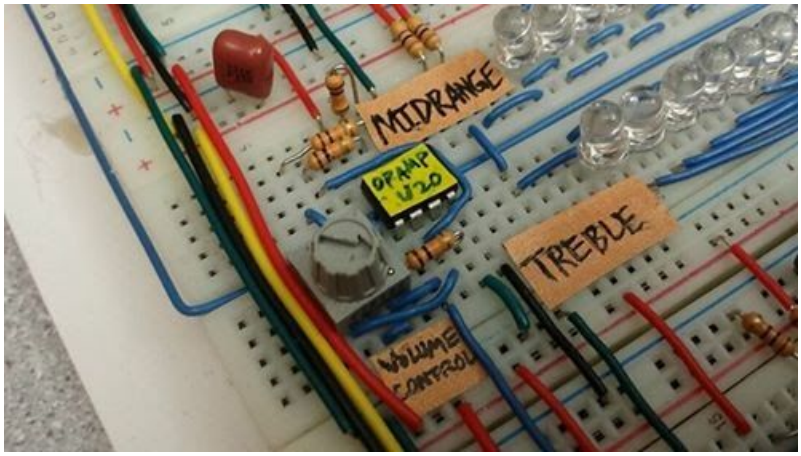
```
void setup() {

  //Declare pin modes
  pinMode(CLK_PIN, OUTPUT); //TLC5916 Clock pin
  pinMode(DATA_PIN, OUTPUT); //TLC5916 SDI
  pinMode(LATCH_PIN, OUTPUT); //TLC5916 LATCH PIN
  pinMode(DISPLAY_SWITCH, OUTPUT); //MOSEFET connected TLC5916 Vdd pin
  pinMode(MIDRANGE_INPUT, INPUT); //input pin from midrange
  pinMode(TREBLE_INPUT, INPUT); // input pin from treble
  pinMode(BASS_PIN, INPUT); //input from bass comparator
  pinMode(ONES_CLK_PIN, OUTPUT); //clock for first digit on Numitron
  pinMode(TENS_CLK_PIN, OUTPUT); //clock for second digit on Numitron
  pinMode(NUMITRON_RESET_PIN, OUTPUT); //reset pin for Numitron counters
  pinMode(MOTOR_PIN, OUTPUT); //haptic feedback pin for bangers

  BeatDetected = false; //initialize ISR flag
  attachInterrupt(digitalPinToInterrupt(BASS_PIN), setISRFlag, RISING);
//attach interrupt to Bass input pin
  digitalWrite(LATCH_PIN, HIGH);        //set latch to high (active low)
  digitalWrite(DISPLAY_SWITCH, LOW);    // Turn on the display driver
(active low)
  turnAllLEDOff();                       // Turn off all LEDs
  resetNumitronDisplay();               //diplay zeros on Numitron

  if (debugMode) {
    Serial.begin(9600);
    Serial.println("Initialization complete.");
  }

  //value initialisation
  BeatCntr = 0; //initate variable to hold number of counts
  StartHapticFeedback = 0; //initiate variable to hold time when haptic
feedback starts
}

void loop() {

  //run when debug mode is active
  if (debugMode) {
    Serial.print("BeatDetected and beat counter: ");
    Serial.print(BeatDetected);
    Serial.print(" ");
    Serial.println(BeatCntr);
  }


  //increment beat counter if beat is detected
  if (BeatDetected) {
```

```
    BeatCntr = BeatCntr + 1;           //increment beat counter

    //store time when haptic feedback starts
    StartHapticFeedback = millis();

    //clear ISR flag
    BeatDetected = false;
  }

  //keep haptic feedback active for HapticFeedbackTime
  if (StartHapticFeedback + HapticFeedbackTime > millis()) {
    analogWrite(MOTOR_PIN, 255); //make motor vibrate at max
  } else {
    analogWrite(MOTOR_PIN, 0); //turn off motor
  }



  //display beat counts on the numitron display
  displayDigit(onesDigitFunc(BeatCntr), tensDigitFunc(BeatCntr));


  //display midrange and treble level on array of LEDs
  turnLEDOn(midrangeLevel(), trebleLevel());

  //decrease amount of flickering by latching value for LEDDisplayTime
milliseconds
  if (digitalRead(LATCH_PIN) == HIGH) { //if not latched
    LatchStart = millis();              //set time when latched
  }

  digitalWrite(LATCH_PIN, LOW); //activate latch(active low)


  //only deactivate latch when LEDDisplayTime has elapsed
  if (millis() > LatchStart + LEDDisplayTime) {
    digitalWrite(LATCH_PIN, HIGH); //deactivate latch
  }

}

//ISR function that changes BeatDetected to true when beat is detected
void setISRFlag() {
  BeatDetected = true;
}
```

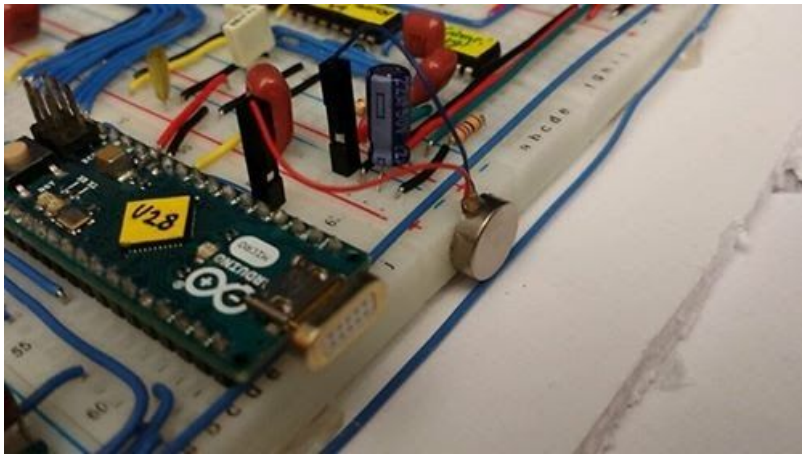
II. Additional Pictures



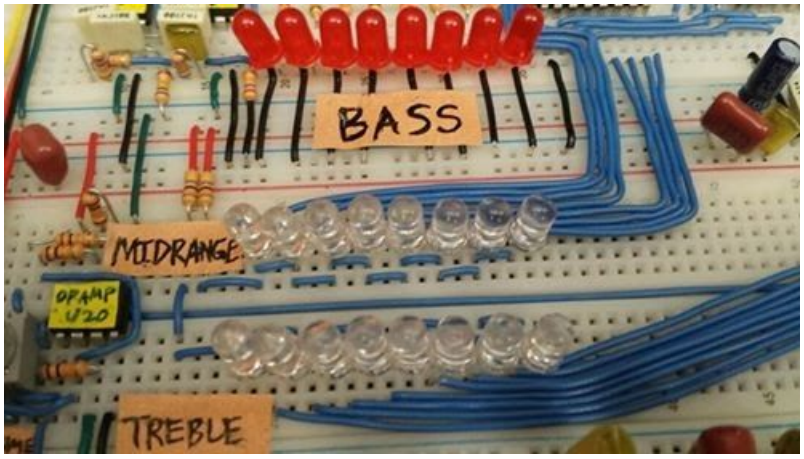
Audio Input jack.



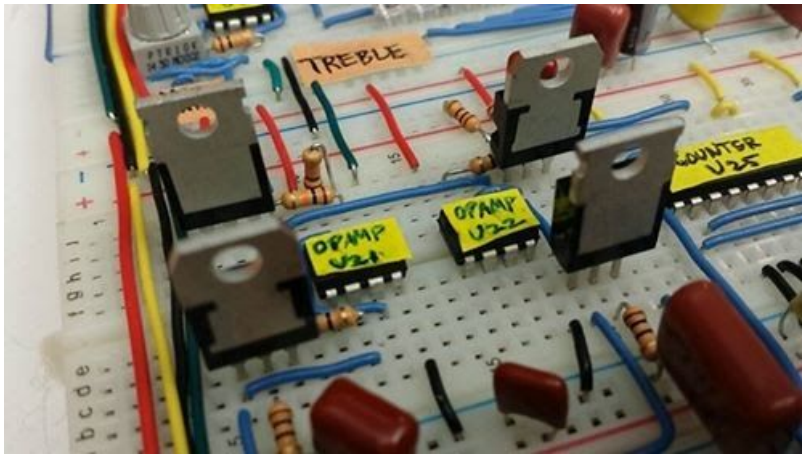
Volume control knob.



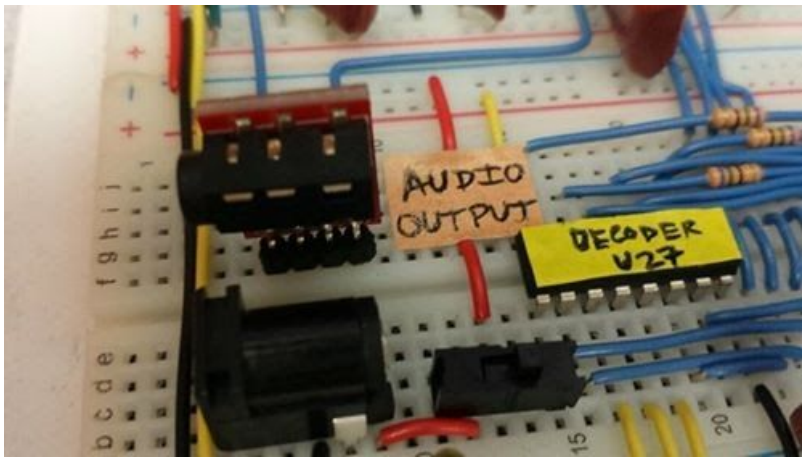
Arduino µController and 3V motor.



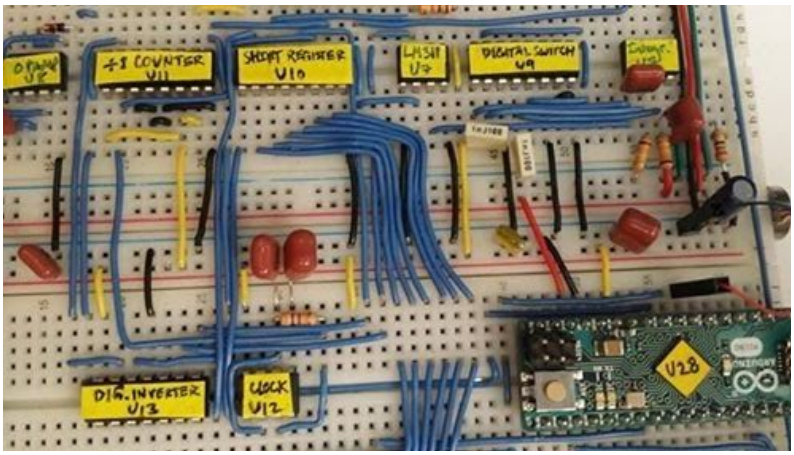
Three 8-LED arrays (bass, midrange, treble).



Class B push-pull amplifier.



Audio output jack.



Components of the single-slope ADC.

III. Full Schematic