<u>Electronic Guitar Tuner</u>

Project completed for the course:
Project 2

Project by: Jared Snape

Presented to: Mr. Gregory Mulcair

2015/05/25

Table of Contents:

Résumé:

Le rapport formel qui suit décrit la création de mon projet final pour la classe Projet 2. Le projet est la création d’un accordeur électronique de guitare que musiciens, professionnels et amateurs seraient en mesure d'utiliser. Ce rapport traite des objectifs primaires de mon projet: créer un accordeur qui fonctionne et qui est à la fois simple d’utilisation et qui a un aspect professionnel en utilisant les techniques et connaissances apprises au cours de mes années en Engineering Technologies, la méthodologie et théorie qui expliquent les décisions que j’ai prises lors de ce projet, ainsi que les étapes et techniques de dépannage utilisés et le travail accompli pour chaque constituant de ce projet. La création d’un accordeur électronique a été choisie comme projet afin d'approfondir ma connaissance de circuits AC, d’amplificateurs opérationnels, de microcontrôleurs et d’applications acoustiques ainsi que pour aider les musiciens qui ne sont pas capables d’accorder leurs instruments par ouïe et requérant un aide.

Executive Summary:

This formal report outlines the creation process of my project for the class Project 2. The project itself is the creation of an electronic guitar tuner for use by both musicians and beginners. This report covers the main objectives of my project: creating a working tuner that is user-friendly and professional in appearance using the knowledge acquired throughout my years of Engineering Technologies, the methodology and theory behind the decisions made during the creation stages of this project as well as the troubleshooting steps taken and work done for each component of this project. This project was designed as a method to both expand my knowledge of AC circuits, operational amplifiers, microcontrollers and acoustic applications as well as to benefit musicians who cannot tune their instrument by ear and require an aid.

Introduction:

My project began as an idea for a device that would complement one of my passions: music. Specifically, I play guitar and desperately need a tuning device. Without a proper tuning device, a musician is forced to rely on his/her hearing in order to tune their guitar and this can lead to an inaccurate setting for the instrument. I sought to understand the fundamentals of how these devices could tune an instrument better than a human ear. With the goal of making a tuner that was both efficient and user-friendly, my project began.

Objectives:

The main objectives of my project are as follows:

- Learn the fundamentals behind a tuner and how one can create such a device.
- Create the code that will be programmed to the microcontroller from scratch.
- Design the circuit that would convert the guitar's signal to a TTL logic wave.
    - The circuit should include filters to reduce the noise and harmonics, a half-wave rectifier to turn the signal into a DC wave and a power source (9v battery).
        - The battery should power the microcontroller as well
- Fit the entire project into several small-size PCBs in order to allow modular design.
- Create a 3D printed case to contain the circuitry.
- Make the user interface as user-friendly as possible and as similar to a marketed product as I could.
    - The guitar needed to be plugged into the device. No microphone.
    - The user interface was to include an LCD screen for instructions, no fewer than seven LEDs to indicate proximity to the ideal note and no more than eight buttons to control the settings.

Methodology/Theory:

Theory behind code:

The first part of this project involved creating the code that would allow me to control the LCD screen, the button inputs, the LED outputs and the input waveform. To this end, I studied various sources online for ideas on frequency counting. The concept I decided to use was the use of two different timer modules on the chip. The first module would be used as an "event counter" to count the number of clock edges that appeared on the input pin. The other module was used as a "time counter" to count real time. These modules provided me with all the information the microcontroller would need to calculate the frequency of the input waveform. To do this, the microcontroller simply had to count the number of rising edges detected on the input waveform, over the course of a known time base. By assuring ourselves that the time base is a second, we can determine the frequency of the wave to be exactly equal to the event counter's result.

Theory of Hertz and frequency:

Because a guitar is a stringed instrument, each string of the guitar is “tuned” to a very specific tautness. This tautness is what will produce the frequency that we hear once the string is strummed. The string will vibrate, producing sound waves whose individual frequencies can be recognized as individual notes.

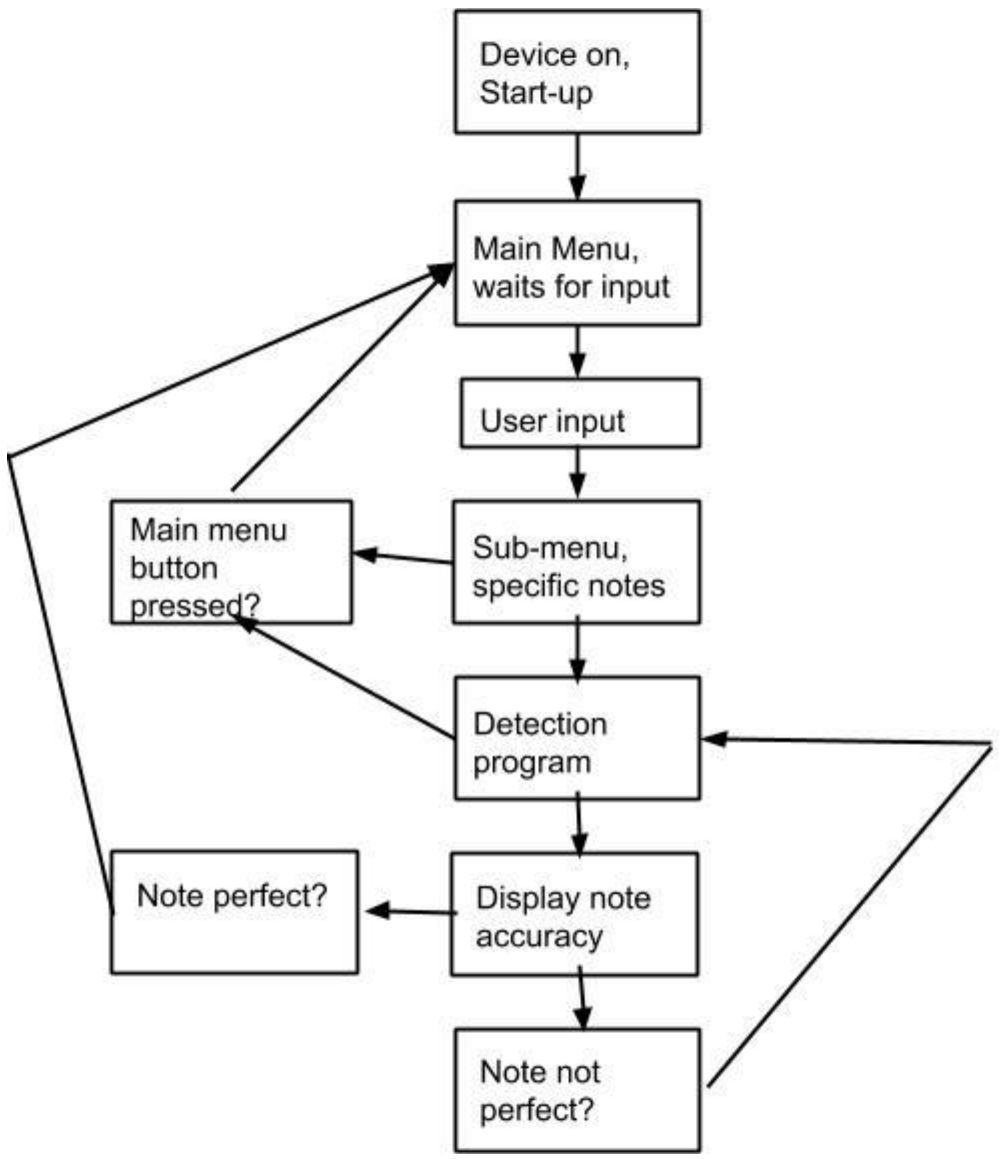
The unit of frequency is Hertz (Hz). This is defined as one cycle of a waveform per second. Therefore, if a waveform has a frequency of 60 Hz, it means that the waveform cycles every 60th of a second. This is important because musical notes have a very precise “fundamental frequency” that allows them to be reproduced, characterized and evaluated. The following chart shows the fundamental frequencies of the seven musical notes I am attempting to detect with my microcontroller. The first note in the list is what guitarists refer to as “Drop D.” This means that, normally, that D is not on the list of available guitar notes (the typical available notes are the 6 following notes in the table). Some guitarists will choose to detune their low E2 string to turn it into a low D2. This is typical for rock and metal guitarists and I decided to add the functionality to tune to this drop D.

Note name (Letter notation/ Solfège notation)	**Frequency (Hz)**
D2 / Ré2	73.4
E2 / Mi2	82.4
A2 / La2	110.0
D3 / Ré3	146.8
G3 / Sol3	196.0
B3 / Si3	246.9
E4 / Mi4	329.6

The microcontroller unfortunately does not have the ability to read and detect fractional quantities. As a result, I have rounded the above frequencies to the next nearest integer. This simplifies the code dramatically as there is a way to account for fractional quantities but it would require a large amount of work for little to no return in efficiency of the device.
The microcontroller I decided to use is the PIC16F887. I chose this microcontroller because I’ve worked on it before and know how to code it to do what I want. The data sheet for the device is easily found online.

Coding:

The first decision I needed to make with regards to the coding was how I wanted the code to “flow”. What series of actions, reactions and options would greet the user and direct them to the crucial part of the process, the frequency detection range, while also remaining user-friendly?
I decided to emphasize making my menu system and options easy to understand over all else except for the frequency detection. As such, my working code flow and algorithm was the following:



Essentially, this algorithm would indicate the process that a typical user would go through while using my guitar tuner. The algorithm begins with program start-up and never reaches a distinct end. Instead, it constantly loops back to an earlier stage in order to prevent the user from becoming confused with what needs to be done to progress or reset the machine.

The way that I chose to implement this in coding was through the use of a massive *while* loop. This loop would constantly run through the basic “awaiting user input” code and provide a basis for the main menu step of the process.

The next portion of code that requires discussion is the frequency detection function. This function (named *detect* in the code) would await for a string on the guitar to be strummed by constantly verifying the current state of the event counter that counts the frequency of the input. Once this event counter exceeds a specific threshold, the *detect* function begins the counting and computing process, where it takes the event counter’s result after an extremely specific sample time and compares it to the sample time in order to determine its frequency (events divided by time) in Hz. At any point in all this process, a user could press the main menu button to return to the original *while* loop used as the main menu. This quick movement from one function to the next is done using a function known as a *break*. A *break* forces the code to stop doing everything that it is currently doing and leave the current function. Through judicious use of *break*s and *goto*s, a function that forces the code to stop doing everything it is currently doing and go to a specific point that has been labelled by the programmer, this main menu button works in every situation where the user could conceivably need to return to a main menu.

The final “section” of the code can be defined as the series of comparison functions that depend on the user input (named *compare_dD* - *compare_hE* in the code). These functions are triggered by two things: a frequency detected by the *detect* function and the user input at the beginning which decides to which note the frequency is to be compared. The specific frequencies for the note chosen are pre-programmed into each of these functions in order to provide a constant basis for the microcontroller to decide on. For example, if the user had previously selected the string E, then strummed a string on his guitar that has a frequency of 80-84 Hz (the acceptable range for an E string’s frequency), the microcontroller would compare this to the value stored in its internal memory in these functions and would then decide whether to advise the user to tighten or loosen the string or declare the string’s setting as perfect and move on.

See appendix for examples of all the code discussed in this part.

Analogue circuitry:

Next came the analogue to digital conversion circuit. This circuit would be in between the guitar and the microcontroller because the microcontroller can only take 0-5V DC while the guitar outputs an extremely small AC wave (around 100 mV - 1V in amplitude). This circuit needed to use operational amplifiers to increase the strength of the signal and filter out what is known as “harmonics” before converting the end result into a 0-5V DC square wave.

An operational amplifier is a small IC which can amplify a small signal to a large one as well as serve as filters to remove distortion from a circuit. They are highly versatile chips and they are used for both of these applications in the following circuitry. I am using a TL072 op amp IC because it has two amplifiers packed into a very small package allowing a more condensed circuit.

The first instance of the operational amplifier in my circuit is used to amplify the initial signal from the guitar. To this end, I connected the op amp to two resistances: a 1 kOhm resistor for the input and a 110 kOhm resistor for the feedback. These two resistance values are key to determining the total amplification (referred to as gain) provided by the amplifier. The equation is as follows for an inverting op amp like the one I am using:

$Gain = \frac{-Rf}{Ri} * Vin$ where Rf = feedback resistance, Ri = input resistance and

. Vin = input voltage

This formula shows that the ratio created by the resistance values is what determines the output voltage. I used a fairly large ratio of 110 to amplify my signal by quite a lot in order to be able to then modify the sound wave much more easily.

This initial gain stage then leads to the three filters that I have set up in order to filter the harmonics.

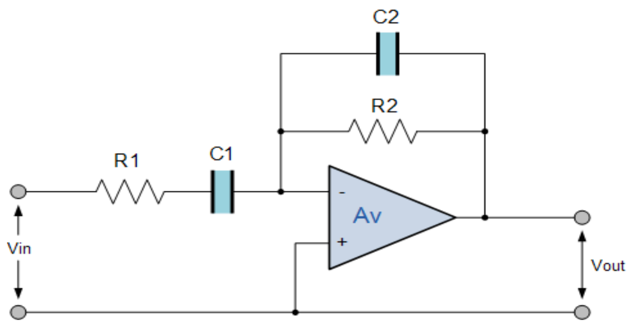
Harmonics are additional frequencies that can be found in all musical notes. These additional frequencies are responsible for creating the characteristic “sound” (called *timbre*) of an instrument. For example, a trumpet and a guitar might be playing the exact same note on the exact same octave but the harmonics present, and their amplitude relative to the initial note’s (*fundamental’s*) amplitude make the two instruments sound extremely different to the human ear. This is important for my project because the harmonics’ frequencies are always whole integer multiples of the fundamental’s frequency. Therefore, a properly tuned E2 string has a fundamental frequency of 82 Hz and various harmonic frequencies: 164 Hz, 246 Hz 328 Hz, etc.

These harmonics need to be filtered out in order to avoid the possibility of the microcontroller detecting the wrong frequency and producing a false negative result.

To filter out the unneeded frequencies, I made use of three 2nd order Butterworth band-pass filters. A band-pass filter is a filter which is simultaneously a high-pass and low-pass filter. It amplifies only the frequencies that fit into its “pass band.” All other frequencies are attenuated. The pass band can be decided upon by the designer of the filter. The following equation is used to determine the “cutoff frequency” (the frequency at which the filter, ideally, begins attenuating the signal).

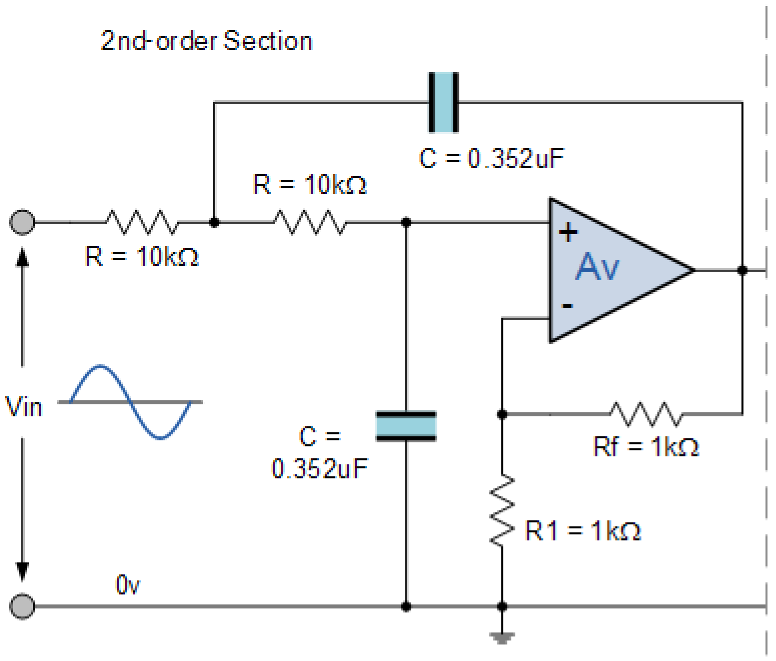
$$fc = \frac{1}{2\pi RC}$$

The symbols R and C represent the resistance and capacitance values connected to the operational amplifier. A typical *active* (meaning that uses an op amp) band pass filter circuit looks like the following image:



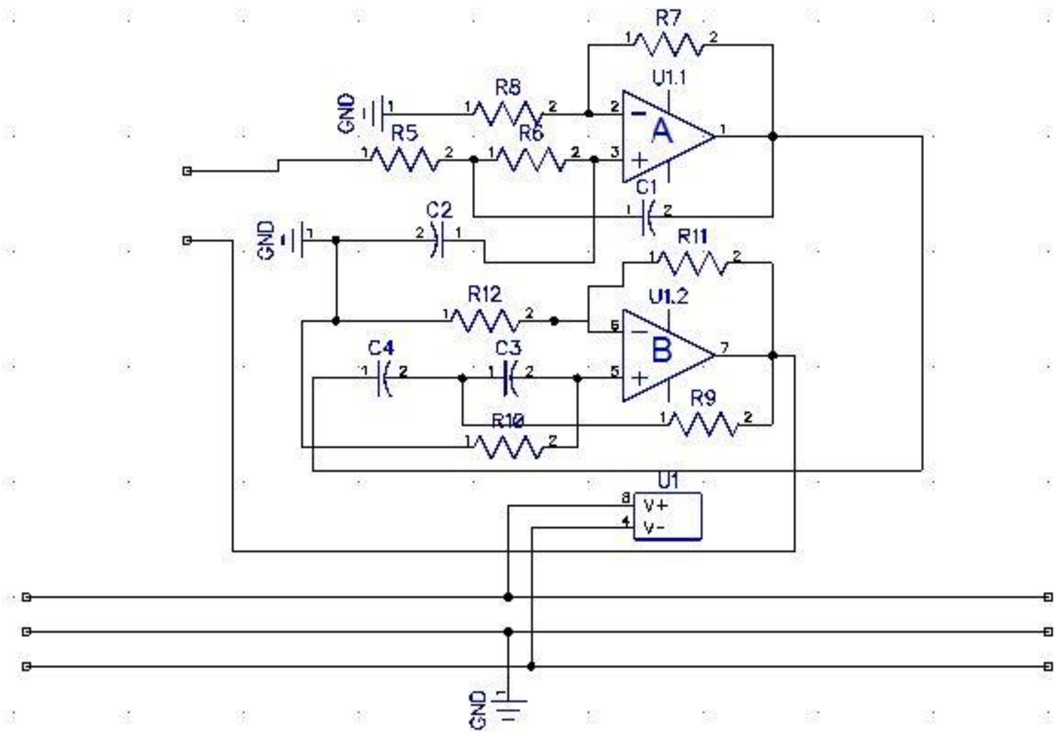
Original Circuit Found at electronics-tutorials.ws

However, I chose to specifically use a 2nd order Butterworth filter. A Butterworth filter is an advanced type of filter which has a slower attenuation than most filters (meaning it takes longer for it to attenuate frequencies to a significant degree) but it has no ripple whatsoever, which allows for very smooth and simple operation. In contrast, a Chebyshev filter cuts off much more quickly but has a large spike of ripple at the end of its pass band. The comparison waveforms are to be found in the appendix.
The circuit design that I chose to use for my 2nd order Butterworth filters was the following:



Original circuit found at electronics-tutorials.ws

This circuit represents the low-pass section of the band pass. I had to combine this original circuit with a similar one, albeit configured for high-pass operation, in order to create the final circuit for my band-pass filter:



This circuit combines the low-pass filter (uppermost op amp) and high-pass filter (lower op amp).

See the appendix for tables indicating filter component calculations and filter performances.

These three filter circuits are wired in parallel to each other, meaning they all receive the same input from the gain circuit. The three outputs of the filters are then sent back to an op amp configured as a summing amplifier. This amplifier takes the largest voltage among its many inputs and amplifies only the one with the highest initial voltage. This is done in order to prevent the possibility of harmonics being allowed through the filters due to similar frequencies. For example, one of the E string's highest harmonics sits squarely in the detection range of the highest filter being used, which has a pass band of 265 to 350 Hz. This means that the harmonic would probably be allowed through. To prevent this from influencing the microcontroller, the summing amplifier takes the highest voltage found (which is, by definition, the fundamental) and only amplifies it. In this fashion, the summing amplifier serves as an additional gate to prevent harmonics from having any significant impact.

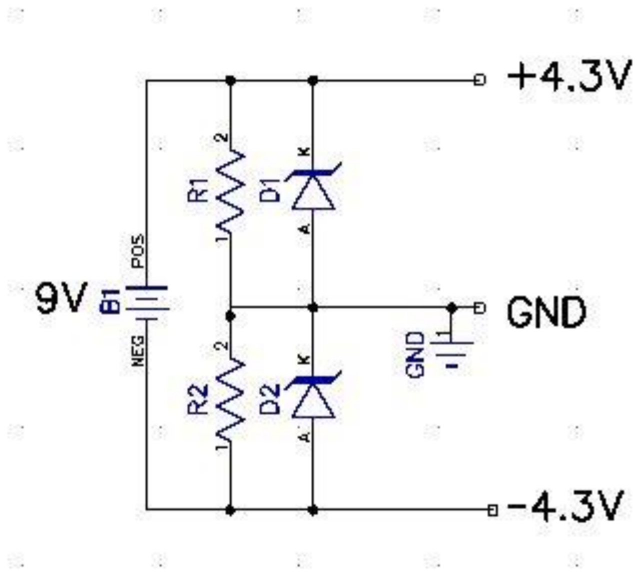
The output of the summing amplifier is then connected to the input of a final op amp configured as a Schmitt trigger. This configuration is similar to a comparator in that a threshold is specified on one of the pins of the amplifier and, as the input oscillates from above to below this threshold level, the output changes from a positive maximum voltage to negative maximum voltage, making the output very square-like. The key difference between a Schmitt trigger and a regular comparator, however, comes with the Schmitt trigger's ability to have a threshold level that depends on the output. Its feedback is, essentially, a voltage divider that takes the output voltage and changes the threshold level to reflect a fraction of the output. As such, it minimizes ripple in an analogue circuit. An example of the Schmitt trigger circuit I used (the design for which was found at PCBHeaven.com) as well as its corresponding threshold waveform will be included in the appendix.

The output of the Schmitt trigger is then sent to the final analogue circuit: the H11L1 opto-isolator circuit. This IC is, essentially, composed of a diode and phototransistor pair. An AC voltage applied to the anode of the diode, turns it on and off in time with the frequency of the AC signal. The diode, when turned on, causes the phototransistor to ground the output of the IC. When the diode is off, the output of the IC is pulled up by a resistor. This is combined with a Schmitt trigger function in the IC which helps keep the wave square and ripple free. The output signal of this IC is, therefore, a perfect square wave of the voltage applied to the IC's positive voltage pin. In my circuit, all positive voltage pins are set at 4.3V. Therefore, the square-wave output of the H11L1 can be used by the microcontroller to measure its frequency as it is a digital signal.

For detailed pictures of all circuits discussed in this part, please consult the appendix.

Voltage regulation circuit:

The battery I have chosen to power this circuit is a typical 9V battery. In order to create a positive voltage terminal and negative voltage terminal to use with my operational amplifiers (which require both a positive and negative voltage as opposed to a positive voltage and ground as most ICs require). In order to do this, I created a voltage divider circuit. This voltage used two resistors of 470 Ohms in series to create 3 voltage points. The topmost point would be a positive 4.5V in relation to the middle point, which we would use as ground. The bottom point would be a negative 4.5V in relation to that same middle point. I used two Zener diodes in order to make sure that these voltages stayed fixed and did not change because of possible impedance from the circuit.



Microcontroller (MCU) circuit:

The microcontroller's circuit is relatively simple and consists mostly of a very large number of inputs and outputs. Primarily, the microcontroller sits in the middle of the circuit with thirteen output pins. These pins connect to two peripheral circuits which I will discuss more in depth in the next section. Of the remaining pins of the microcontroller, there are nine input pins, eight of which are connected to the final peripheral circuit and one of which is the input from the analogue circuitry, and two pins reserved for the microcontroller's crystal oscillator. This crystal oscillator determines the processing speed of the MCU as well as the speed of the timing functions within the MCU. For these reasons, this crystal was crucial to the proper operation of the MCU because without it the microcontroller could not operate as quickly as I would like it too, nor could it time itself nearly as accurately, meaning that the time base for my coded frequency detector might not have worked if the crystal was installed incorrectly. The crystal also required two capacitors connected in series to its pins, connected to ground. The capacitance of these capacitors was determined to be around 12 pF.

Peripheral circuits:

There are three circuits that I deem “peripheral” in the sense that they were not necessarily related to any of the other circuits except the microcontroller’s circuit. These circuits served as outputs or inputs for the MCU and as user interface devices.

The first peripheral is the LCD and its circuit. The LCD is connected to six outputs from the microcontroller and each of these inputs has a current-limiting 1000 Ohms resistor. Furthermore, the third pin of the LCD is used as a variable voltage input to control the contrast of the LCD’s letters. This pin is connected to the wiper of a 10k Ohms potentiometer which is calibrated for optimal contrast on the lettering of the LCD.

The second peripheral circuit is composed of seven LEDs. One of these LEDs is green while the others are red. These LEDs serve as a visual indicator as to the accuracy of the note being played by the guitarist. If a note is too sharp, the red LEDs to the right of the green LED in the centre light up depending on how far from the actual note the detected note is. If the strummed note is too flat, the red LEDs to the left of the green LED light up depending on proximity to the actual note. If the note is perfect, the green LED lights up. Each of these is connected on their anode to a MCU output and are connected on their cathode to a current-limiting resistor that is connected to ground.

The third and final peripheral circuit is the switchboard. This board is equipped with eight switches which serve as the user’s input to the MCU. As such, they occupy eight of the nine input pins on the MCU board. Each of these switches is a single pole, dual position switch. This means that they have a common point which connects to one of the switches’ inputs when the button is not pressed and connects to the other input when the button is not. The normally closed input in this case is the ground line while the normally open input is the positive voltage line, meaning that when any of the buttons is pressed, the MCU detects a positive voltage spike from that specific input and makes a decision accordingly.

All of the above circuits are included in the appendix.

Creation of the box:

I knew I wanted to create a sleek casing for this project in order to make it look as professional as possible. To this end, I used the knowledge I learned in my Design and Simulation class to create a box design on Solidworks. The Solidworks parts were made perfectly to scale by measuring every individual piece of the device that needed to fit in the box.

The circuits described in the Methodology/Theory section were all redesigned by myself in order to suit the project’s needs, using the knowledge acquired throughout my years in Engineering Technologies. The concepts, ideas and implementation examples of these circuits, however, come from many sources scattered across various media and I take no responsibility in their initial design or creation, I simply modified them to suit my needs as a technologist.

The redesigns of these circuits were all converted to PCB layouts and designed by myself. The PCB layouts are unique and required several hours of careful planning in order to have them operate properly.

The PCB layouts were then printed, once again by myself with supervision by Dave Martin, a teacher from the department.

Work and Troubleshooting:

This section will discuss the work process involved with each circuit from the previous circuit and the troubleshooting steps taken for each.

Coding:

The code began pretty simply but, before long, required a large amount of troubleshooting as I modified it. The first problem encountered was making sure the main menu worked properly and could be returned to at any point. To test the code before implementing it, I used a software called MPLab IDE. This is a powerful software that can serve as a compiler, debugger and programmer all at once while also allowing you to “step through” the code and watch the variables defined change as the code progresses. It even allows the user to change values on the fly to simulate various conditions. This software allowed me to spot the various bugs in my initial code and proved invaluable in my overall comprehension of the code as it let me test how the code would react given various stimuli.

The second major problem I encountered with the code was due to a combination of hardware limitations and relatively beginner programming attitude. The first instance of the *compare_x* functions required far too much memory from the MCU because all the functions were being placed “in-line” meaning they were being treated as one large function due to the code being over-cluttered. The solution I found to prevent this from happening was to purposefully define every function as being separate from the others. This was done by adding a line stating “#SEPARATE” above each function and its definition. This solved the problem without having to resort to large changes in the code which, while possible, would have caused a huge delay in my timeline for finishing the code.

Analogue circuitry:

Many hours of troubleshooting were spent on the analogue circuitry of this project and, as such, it will be further subdivided for ease of reading.

Gain amplifier and summing amplifier circuit:
This circuit required minimal troubleshooting, the ICs worked perfectly on the first attempt. However the trace connecting the negative voltage line to the negative voltage source pin on the op amp had become eroded after a few days. As a result, I was forced

to connect a jumper wire from this pin to the voltage line in order to ensure proper operation.

Schmitt trigger and opto-isolator circuit:
This circuit also required minimal troubleshooting, the ICs worked perfectly on the first attempt. However the output would, on occasion, stop being a perfect square wave as expected and would become flatter, the voltage level dropping to about 1.5V on average. Upon investigation, I discovered a bad solder joint connecting the output to its pull-up resistor. Once I repaired the connection, the output never faltered again.

Filter circuits:
These circuits alone caused hours of troubleshooting. The first problem occurred when I was verifying the first filter's (unattached from the others) performance and noticed an occasional faltering in performance where it would allow frequencies through despite them being beyond its pass band. It took me some time to discover that a few resistors and capacitors were badly connected to the pins of the IC and, after reconnecting them properly, the first problem was fixed.

The second problem occurred while testing the full circuit, with all three filters connected in parallel to each other. The performance of each, while not spectacular, was more than acceptable. However, the output being sent to the microcontroller was rife with inaccurate frequencies. These frequencies should not have been allowed to pass through the filters or the summing amplifier. After hours of research and troubleshooting attempts, including multimeter voltage level testing, oscilloscope readouts and disconnections and reconnections, I realized the problem: certain harmonics of guitar strings are particularly powerful in amplitude and, because of this, the summing amplifier was having great difficulty detecting the correct voltage to amplify. Furthermore, it had come to my attention that all three filters, each covering a specific pass band, in parallel still allowed more harmonics than I had initially planned for to enter the rest of the circuit. Together, these harmonics provided enough distortion to make the MCU completely unable to detect the correct frequency among them. If I were to recreate these circuits, I would increase the order of all the filters, making them all 4th order band pass filters and reduce their tolerable range to include individual notes only. This would mean much more circuits and, potentially, more troubleshooting but the payoff would be ultra-high-performance filters and no worries related to harmonics. Another, more moderate, approach I would take if I had to revisit this project would be to include transistor switches to allow the microcontroller to only enable single filters at a time. This would allow the filters to work without interference from each other and if this tactic were combined with the previous one, could very well lead to an extremely precise guitar tuner, similar to products found on the open market. In order to stop the filters from interfering with one another, I removed the two higher range filters to allow at least the bottom three strings perfect functionality. If not for this decision, the project timeline would never have been met.

Voltage regulation circuit:

This circuit required some troubleshooting that was quickly fixed. The first problem was caused by the regulator circuit's resistance becoming in parallel with the other circuits' impedance. This meant that, despite the fact that I was using two of the same resistors, the voltage drop across them was vastly different. To solve this problem, I added two 4.3V Zener diodes to the circuit. Their addition caused the voltage drop across the resistors to remain at a set 4.3V (close enough to 5 for the logic elements of the circuits to work) and forced the resistor to stop being viewed as a parallel connection to the circuit.

The second problem encountered, however, was caused by a simple mistake in my calculations. The current load that I had accounted for was far too low. As a result, the devices were not powering up properly. Upon revisiting my calculations, I saw my error and quickly fixed it, replacing the original resistors with much smaller values in order to account for the approximate 9 mA of current necessary.

Microcontroller circuit:

The microcontroller's circuit, in and of itself was simple and required no troubleshooting. However, I found out (only after soldering them in place) that the headers I had chosen to hold my MCU were far too long to allow a connection between the MCU and its circuit. After learning this, I replaced the headers with a much smaller, more efficient MCU-specific holder.

Peripheral circuits:

The various peripheral circuits had several troubleshooting issues as well.

The LCD board had several bad solder connections, eroded traces and was not working as expected in any way. I tested the LCD itself to make sure I hadn't made some error in its pin placements and it worked when used with the school's MCU trainer boards. As a result, I decided to try recreating its board on a simpler level using perforated board and it began working again. I decided to get rid of the faulty board and replace it entirely with the perforated copper board.

The switch board suffered immensely during the etching process. While soldering the switches onto the board, I noticed erosion on nearly every trace. To repair this damage, I created a long line of solder over some of the lines to protect them further and ensure a strong electrical connection. Next, I needed to repair some of the connections between the switches. To this end, I added silver wires connecting some of the switches to each other in order to bridge the gaps. During every step of this process I used the function on my digital multimeter that would alert me to gaps in the lines and missing connections. This proved an invaluable tool to seeing the mistakes that had occurred during the etching process.

The LED board worked perfectly without need for troubleshooting whatsoever.

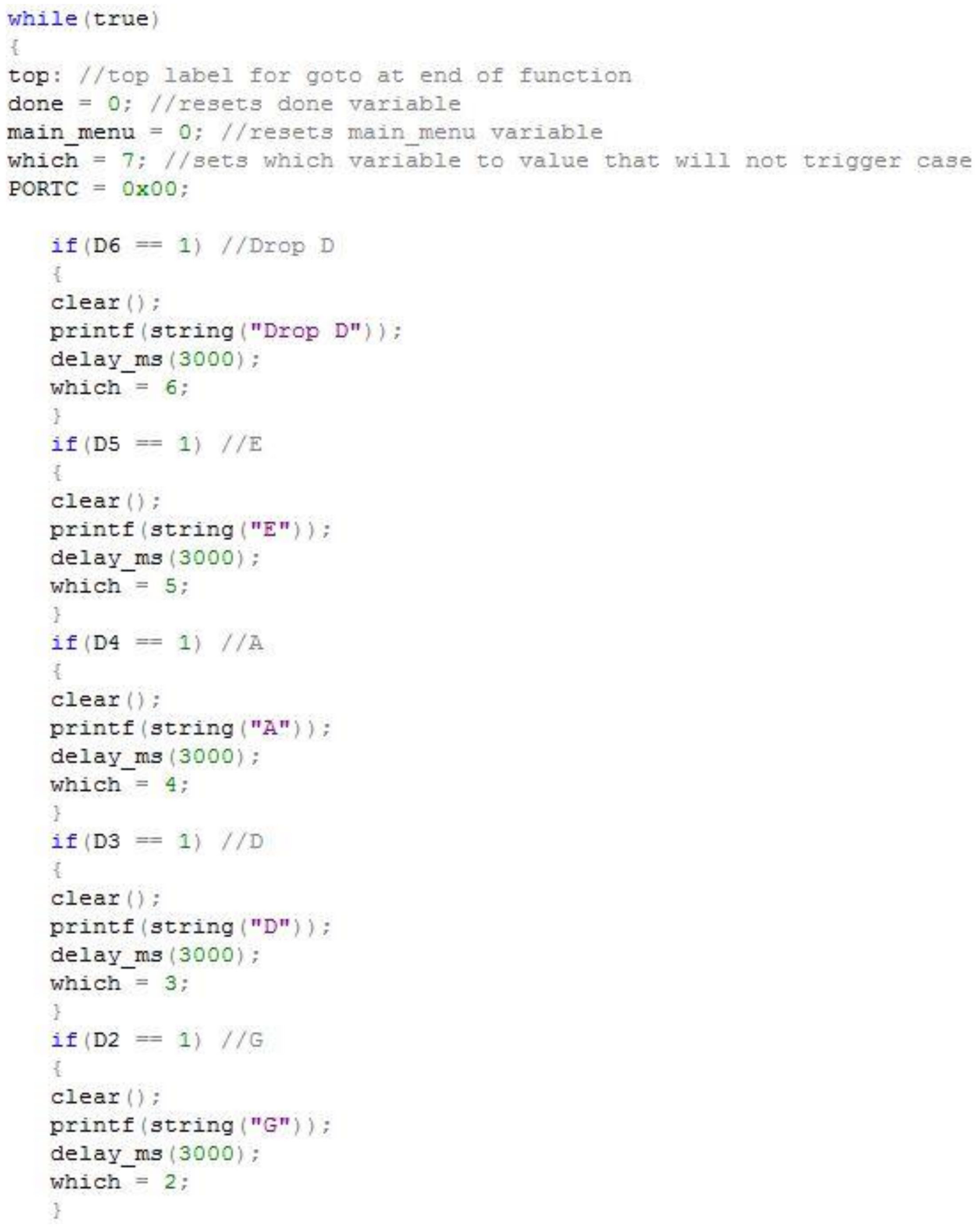
Conclusion:

In conclusion, I believe that the original objectives of this project were met admirably. Three of the six strings can be tuned perfectly and, given more time, I'm sure I could easily repair the problems causing the three higher strings to not be operational. Working on this project taught me a lot about operational amplifiers, microcontroller coding in C and troubleshooting techniques. I believe the end result to be ergonomic, user-friendly and professional-looking.

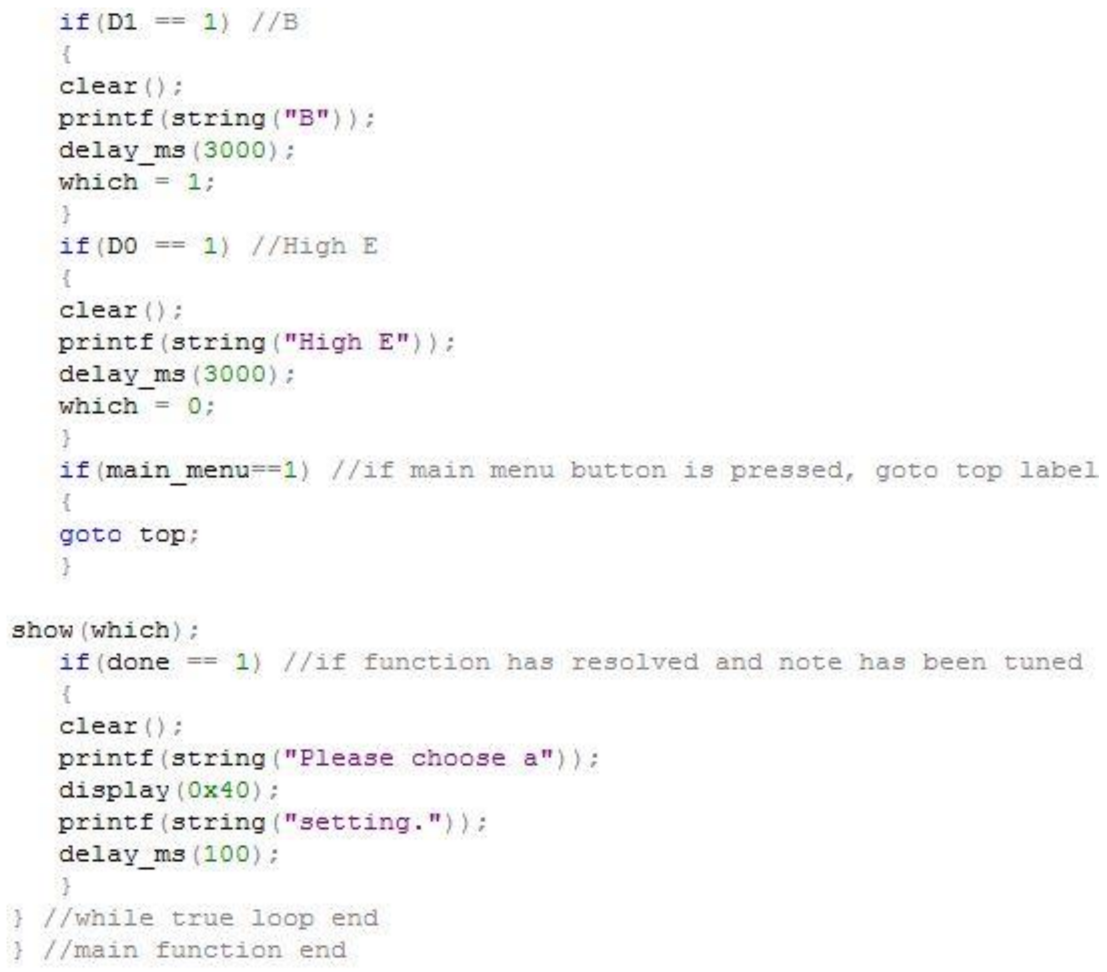
Appendix:

Code:

Main menu:

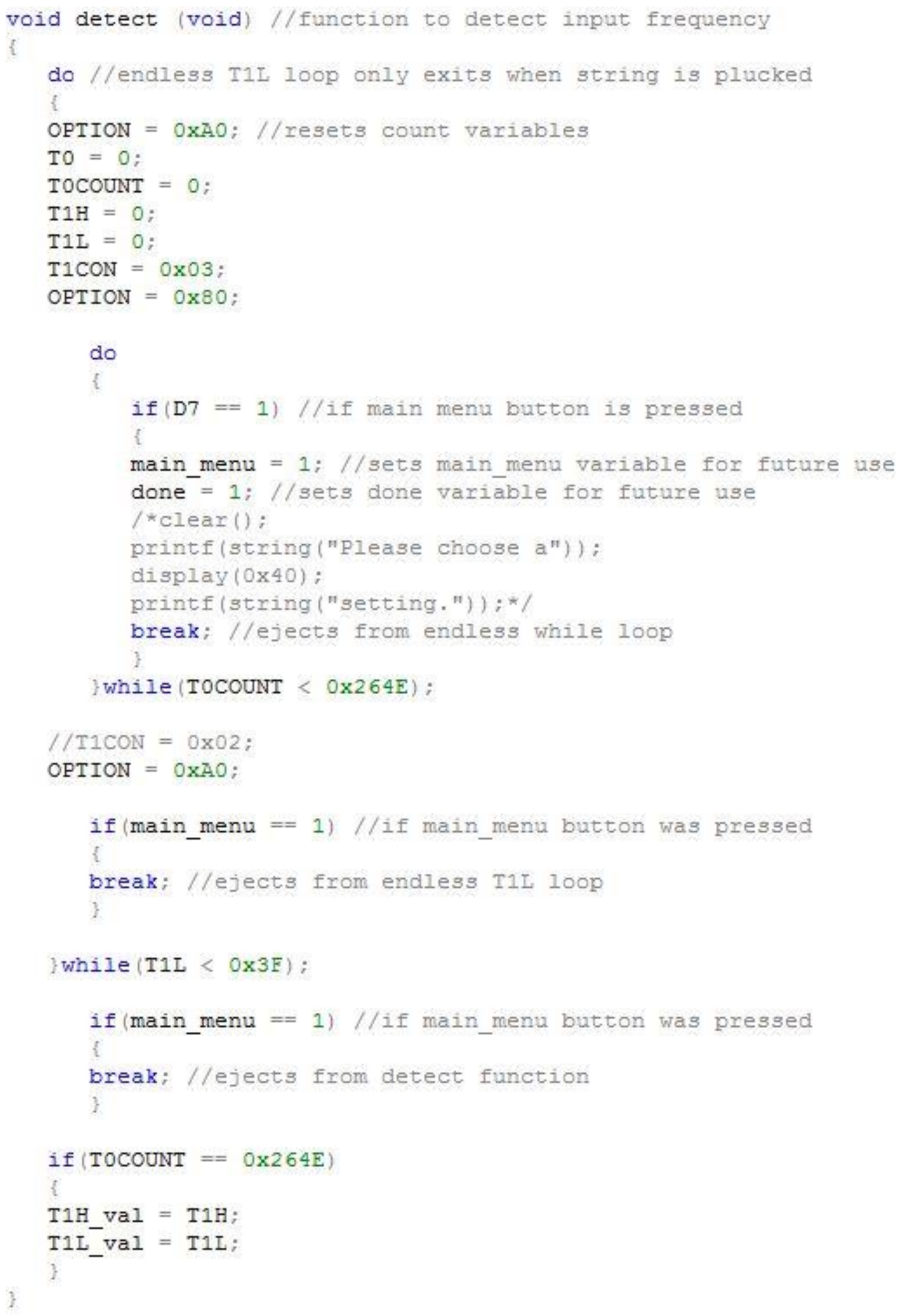


```
while(true)
{
top: //top label for goto at end of function
done = 0; //resets done variable
main_menu = 0; //resets main_menu variable
which = 7; //sets which variable to value that will not trigger case
PORTC = 0x00;

   if(D6 == 1) //Drop D
   {
   clear();
   printf(string("Drop D"));
   delay_ms(3000);
   which = 6;
   }
   if(D5 == 1) //E
   {
   clear();
   printf(string("E"));
   delay_ms(3000);
   which = 5;
   }
   if(D4 == 1) //A
   {
   clear();
   printf(string("A"));
   delay_ms(3000);
   which = 4;
   }
   if(D3 == 1) //D
   {
   clear();
   printf(string("D"));
   delay_ms(3000);
   which = 3;
   }
   if(D2 == 1) //G
   {
   clear();
   printf(string("G"));
   delay_ms(3000);
   which = 2;
   }
```



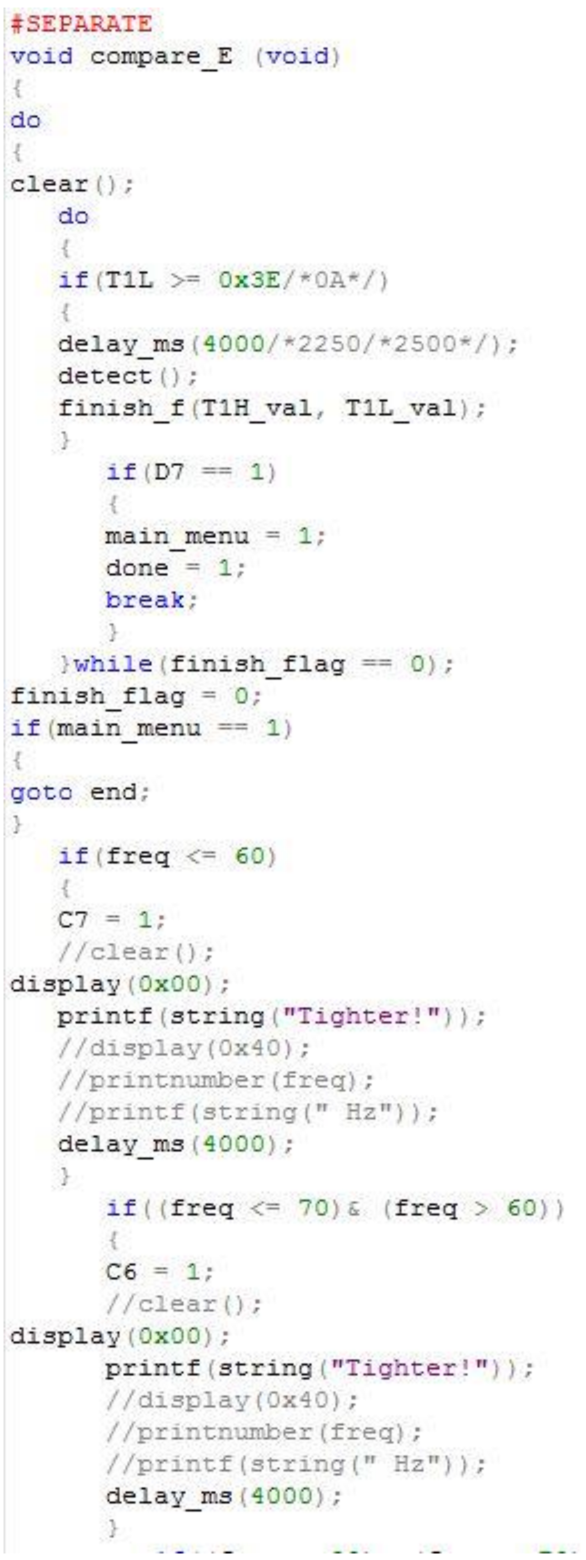
```
    if(D1 == 1) //B
    {
    clear();
    printf(string("B"));
    delay_ms(3000);
    which = 1;
    }
    if(D0 == 1) //High E
    {
    clear();
    printf(string("High E"));
    delay_ms(3000);
    which = 0;
    }
    if(main_menu==1) //if main menu button is pressed, goto top label
    {
    goto top;
    }

show(which);
    if(done == 1) //if function has resolved and note has been tuned
    {
    clear();
    printf(string("Please choose a"));
    display(0x40);
    printf(string("setting."));
    delay_ms(100);
    }
} //while true loop end
} //main function end
```

Detect:

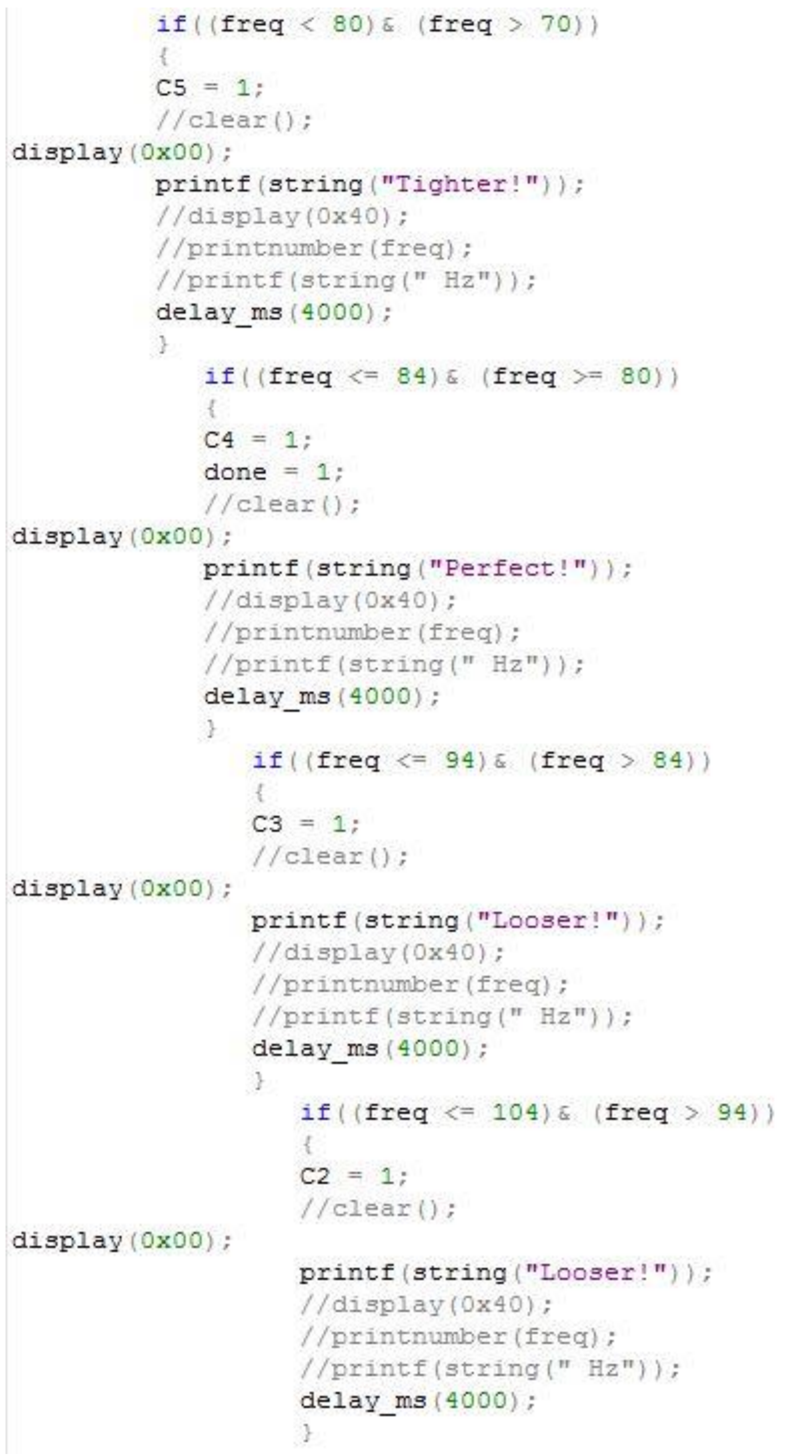


```
void detect (void) //function to detect input frequency
{
    do //endless T1L loop only exits when string is plucked
    {
    OPTION = 0xA0; //resets count variables
    T0 = 0;
    T0COUNT = 0;
    T1H = 0;
    T1L = 0;
    T1CON = 0x03;
    OPTION = 0x80;

        do
        {
            if(D7 == 1) //if main menu button is pressed
            {
            main_menu = 1; //sets main_menu variable for future use
            done = 1; //sets done variable for future use
            /*clear();
            printf(string("Please choose a"));
            display(0x40);
            printf(string("setting."));*/
            break; //ejects from endless while loop
            }
        }while(T0COUNT < 0x264E);

    //T1CON = 0x02;
    OPTION = 0xA0;

        if(main_menu == 1) //if main_menu button was pressed
        {
        break; //ejects from endless T1L loop
        }

    }while(T1L < 0x3F);

        if(main_menu == 1) //if main_menu button was pressed
        {
        break; //ejects from detect function
        }

    if(T0COUNT == 0x264E)
    {
    T1H_val = T1H;
    T1L_val = T1L;
    }
}
```

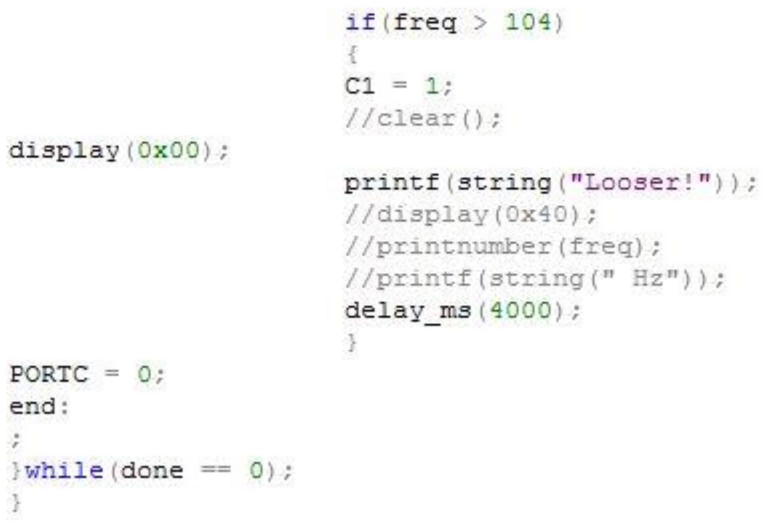
Compare:



```
#SEPARATE
void compare_E (void)
{
do
{
clear();
   do
   {
   if(T1L >= 0x3E/*0A*/)
   {
   delay_ms(4000/*2250/*2500*/);
   detect();
   finish_f(T1H_val, T1L_val);
   }
      if(D7 == 1)
      {
      main_menu = 1;
      done = 1;
      break;
      }
   }while(finish_flag == 0);
finish_flag = 0;
if(main_menu == 1)
{
goto end;
}
   if(freq <= 60)
   {
   C7 = 1;
   //clear();
display(0x00);
   printf(string("Tighter!"));
   //display(0x40);
   //printnumber(freq);
   //printf(string(" Hz"));
   delay_ms(4000);
   }
      if((freq <= 70)& (freq > 60))
      {
      C6 = 1;
      //clear();
display(0x00);
      printf(string("Tighter!"));
      //display(0x40);
      //printnumber(freq);
      //printf(string(" Hz"));
      delay_ms(4000);
      }
```



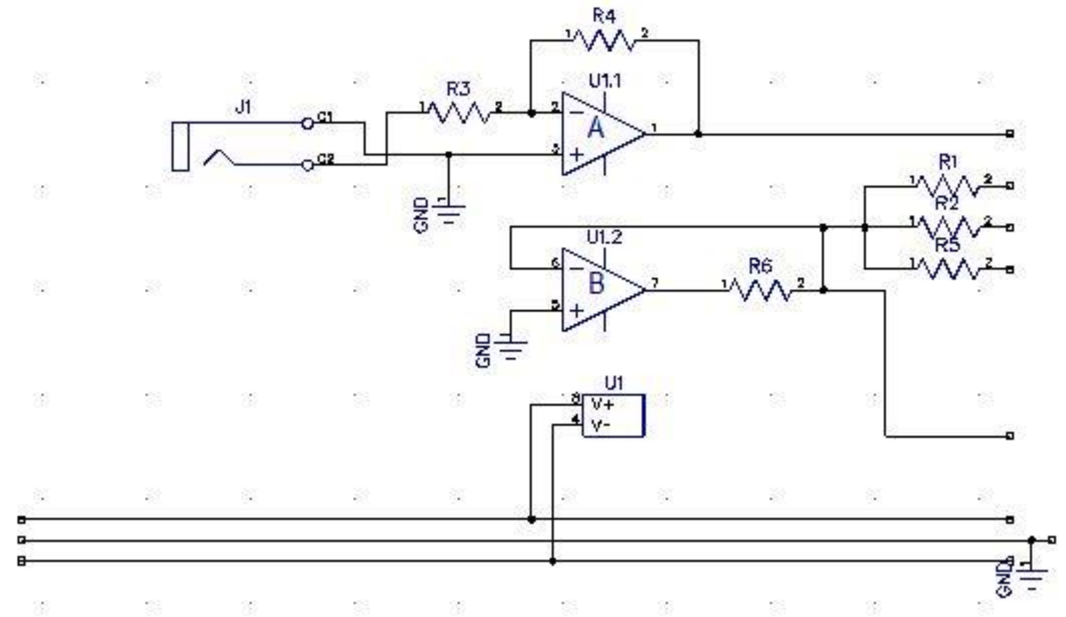
```
         if((freq < 80)& (freq > 70))
         {
         C5 = 1;
         //clear();
display(0x00);
         printf(string("Tighter!"));
         //display(0x40);
         //printnumber(freq);
         //printf(string(" Hz"));
         delay_ms(4000);
         }
            if((freq <= 84)& (freq >= 80))
            {
            C4 = 1;
            done = 1;
            //clear();
display(0x00);
            printf(string("Perfect!"));
            //display(0x40);
            //printnumber(freq);
            //printf(string(" Hz"));
            delay_ms(4000);
            }
               if((freq <= 94)& (freq > 84))
               {
               C3 = 1;
               //clear();
display(0x00);
               printf(string("Looser!"));
               //display(0x40);
               //printnumber(freq);
               //printf(string(" Hz"));
               delay_ms(4000);
               }
                  if((freq <= 104)& (freq > 94))
                  {
                  C2 = 1;
                  //clear();
display(0x00);
                  printf(string("Looser!"));
                  //display(0x40);
                  //printnumber(freq);
                  //printf(string(" Hz"));
                  delay_ms(4000);
                  }
```

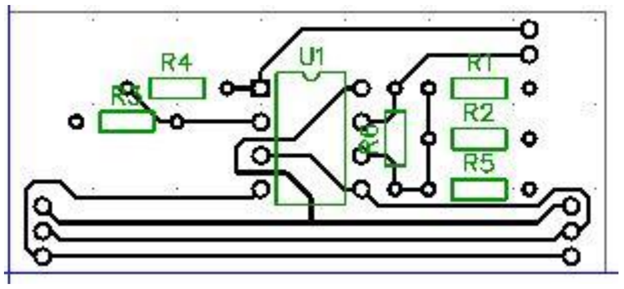


```
                            if(freq > 104)
                            {
                            C1 = 1;
                            //clear();
display(0x00);
                            printf(string("Looser!"));
                            //display(0x40);
                            //printnumber(freq);
                            //printf(string(" Hz"));
                            delay_ms(4000);
                            }
PORTC = 0;
end:
;
}while(done == 0);
}
```

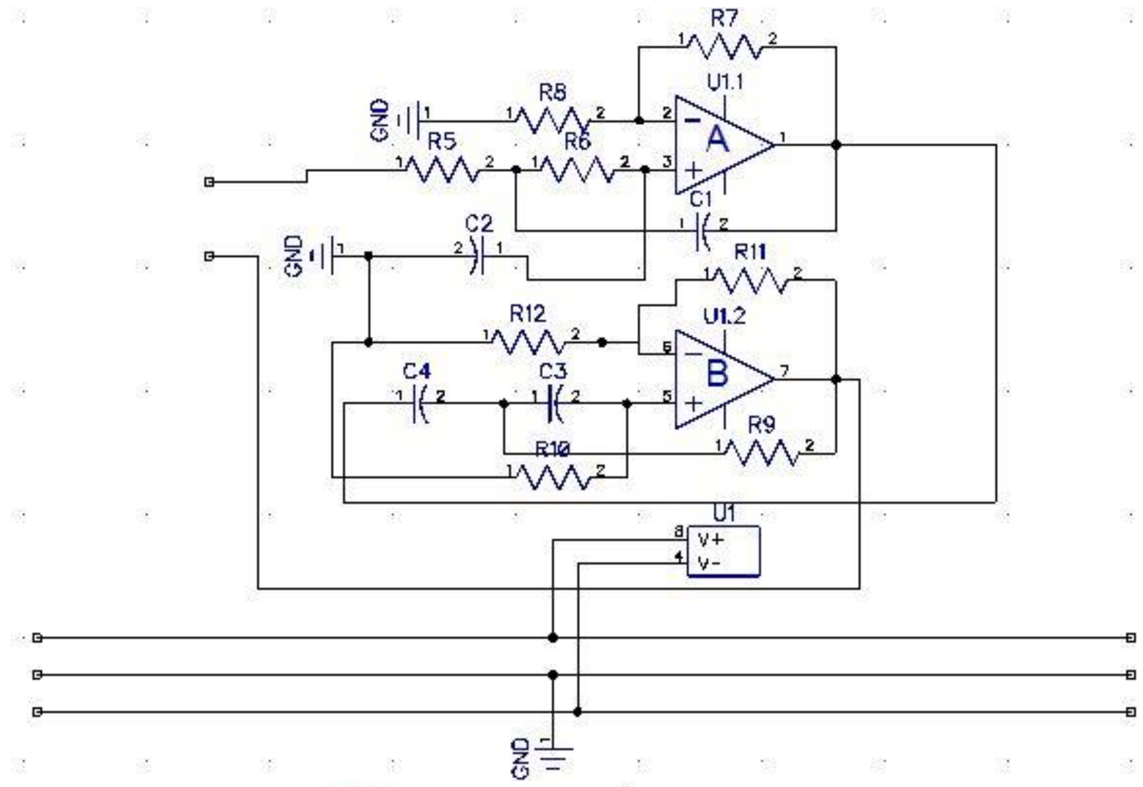
Analogue:

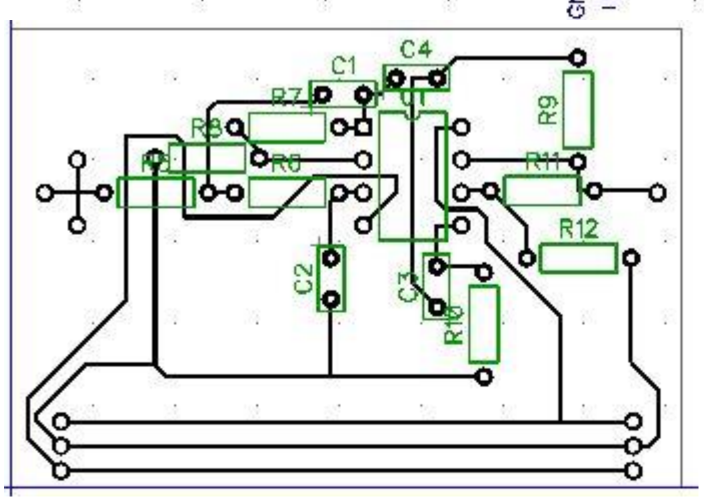
Gain amplifier and summing amplifier:



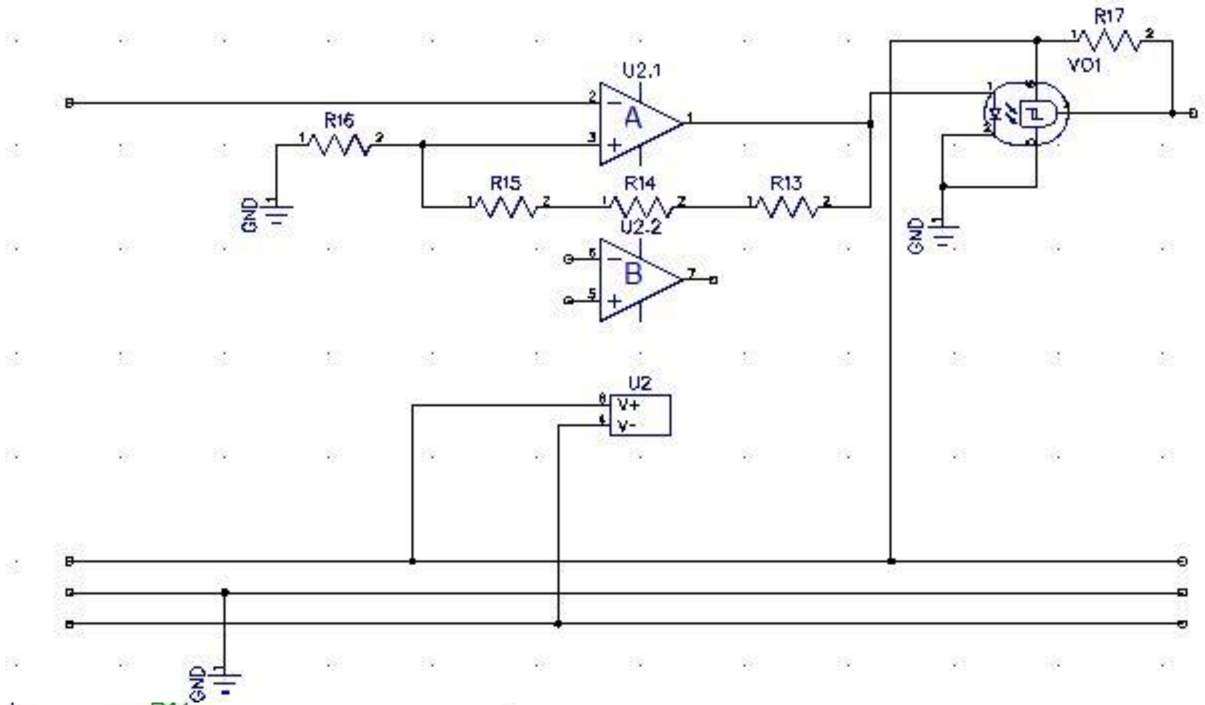


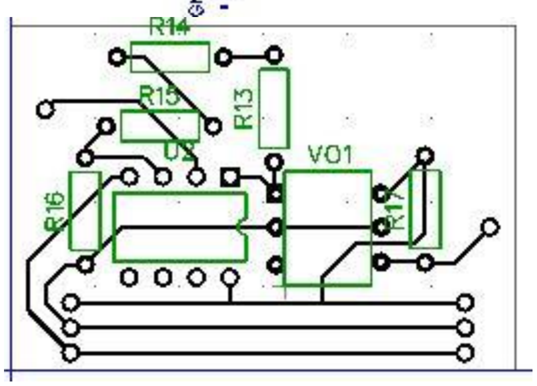
Filters:



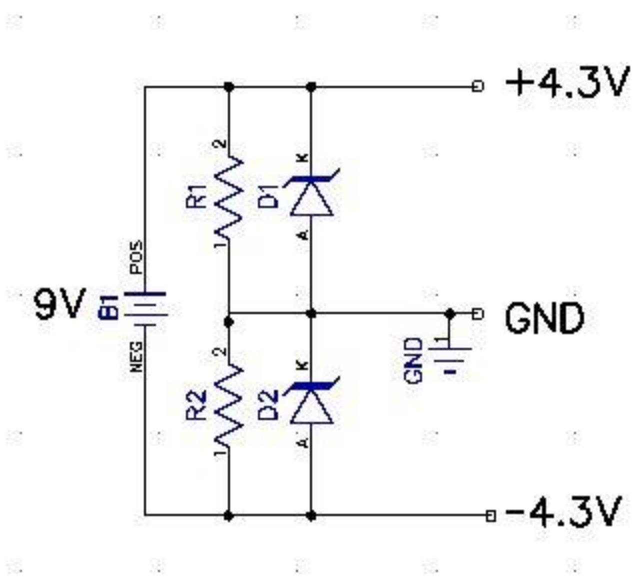


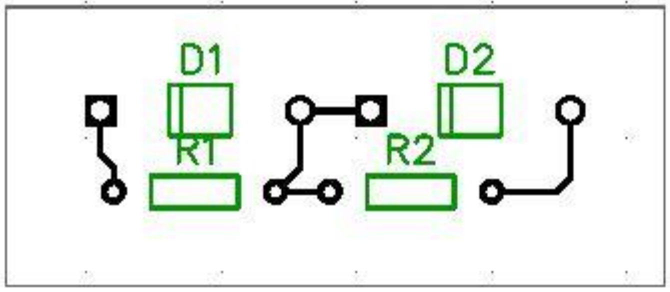
Schmitt trigger and opto-isolator:



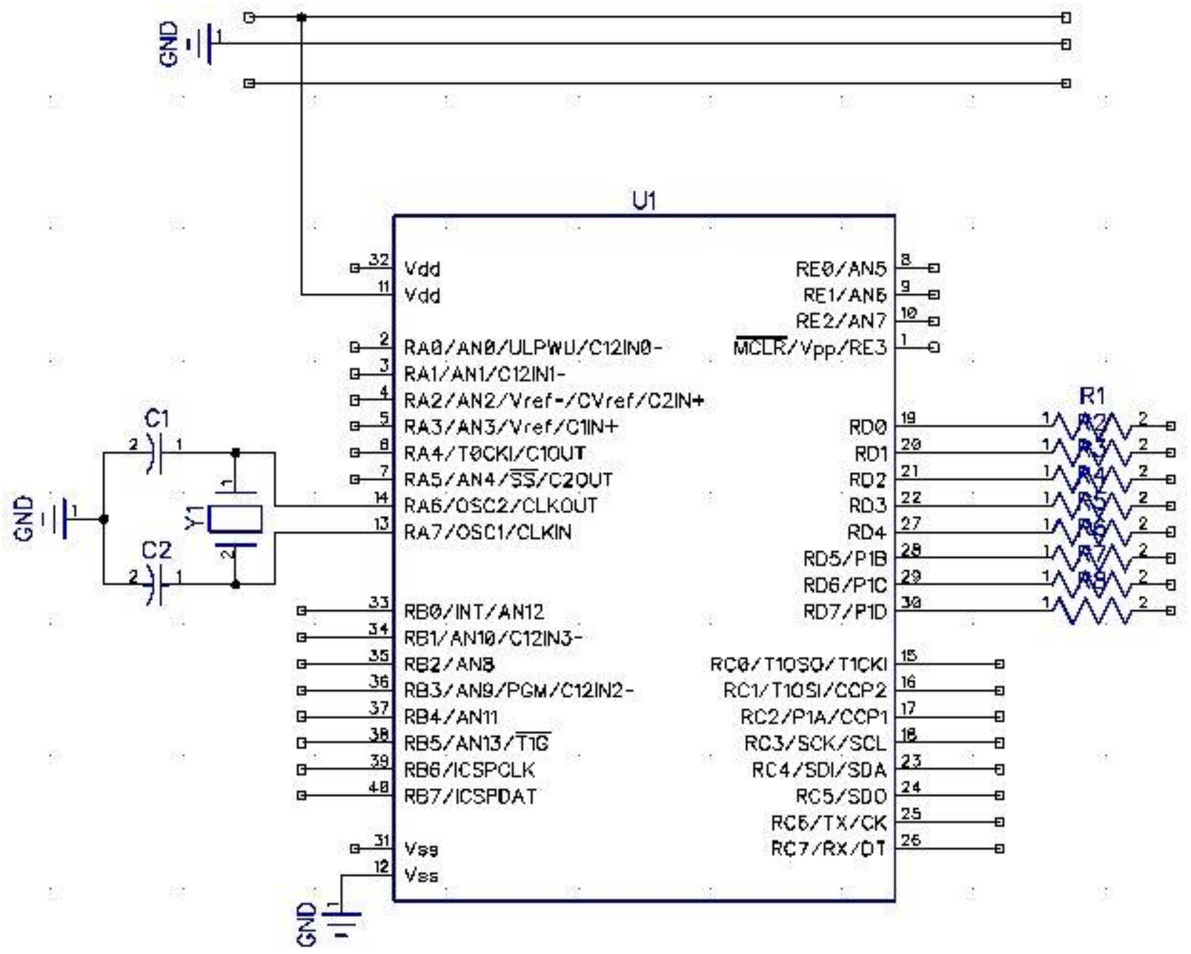


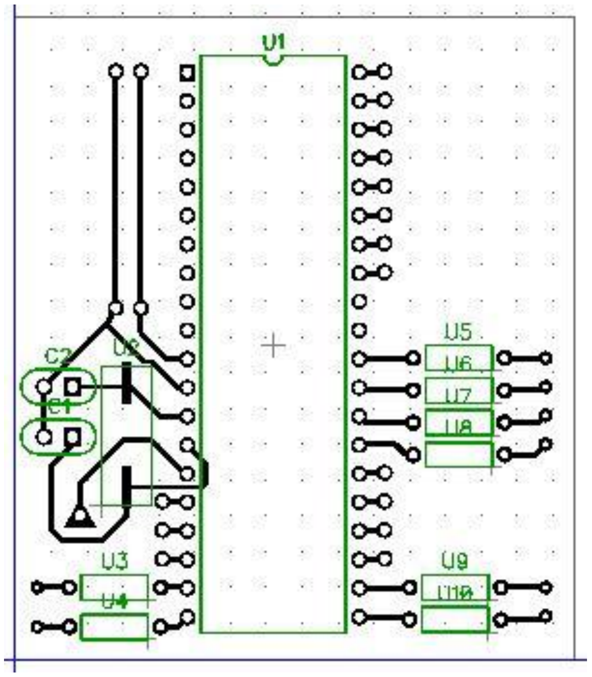
Voltage Regulator:


+4.3V
GND
-4.3V
9V
B1
POS
NEG
R1
R2
D1
D2
K
A
GND


D1
D2
R1
R2

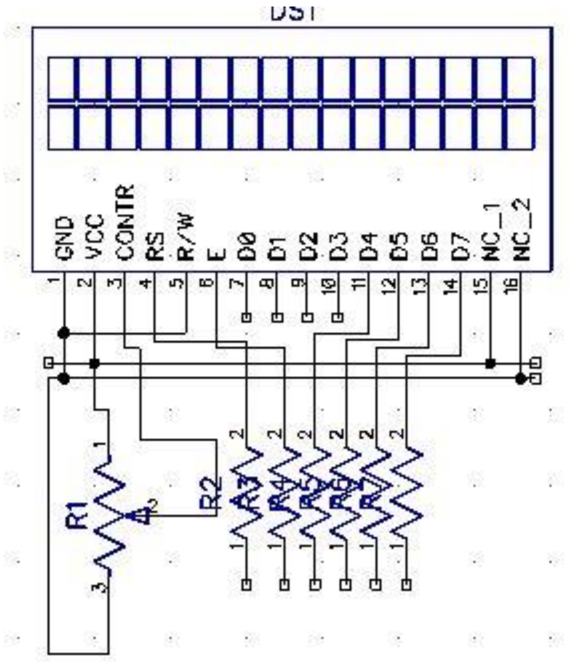
MCU:



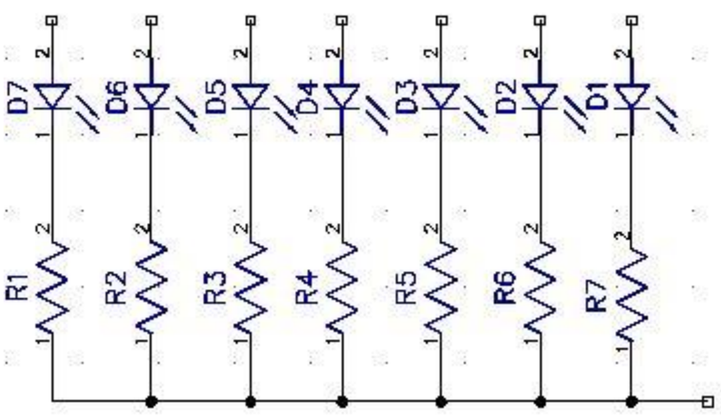


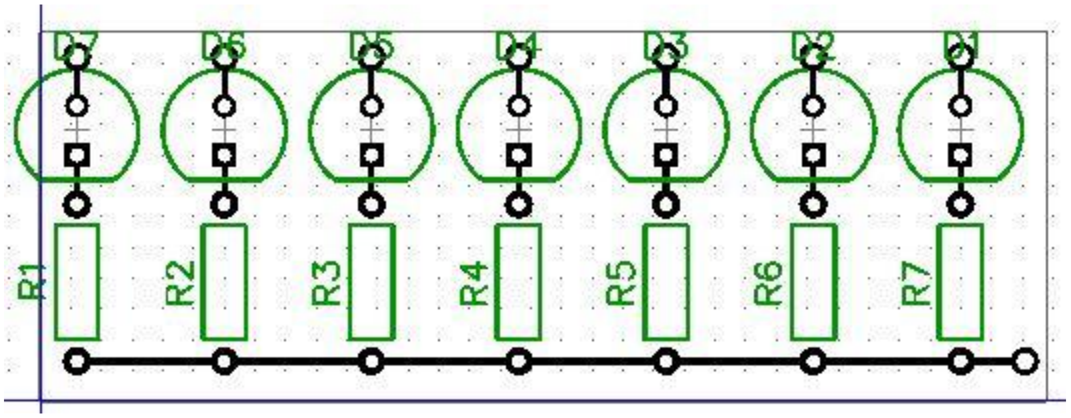
Peripherals:

LCD:

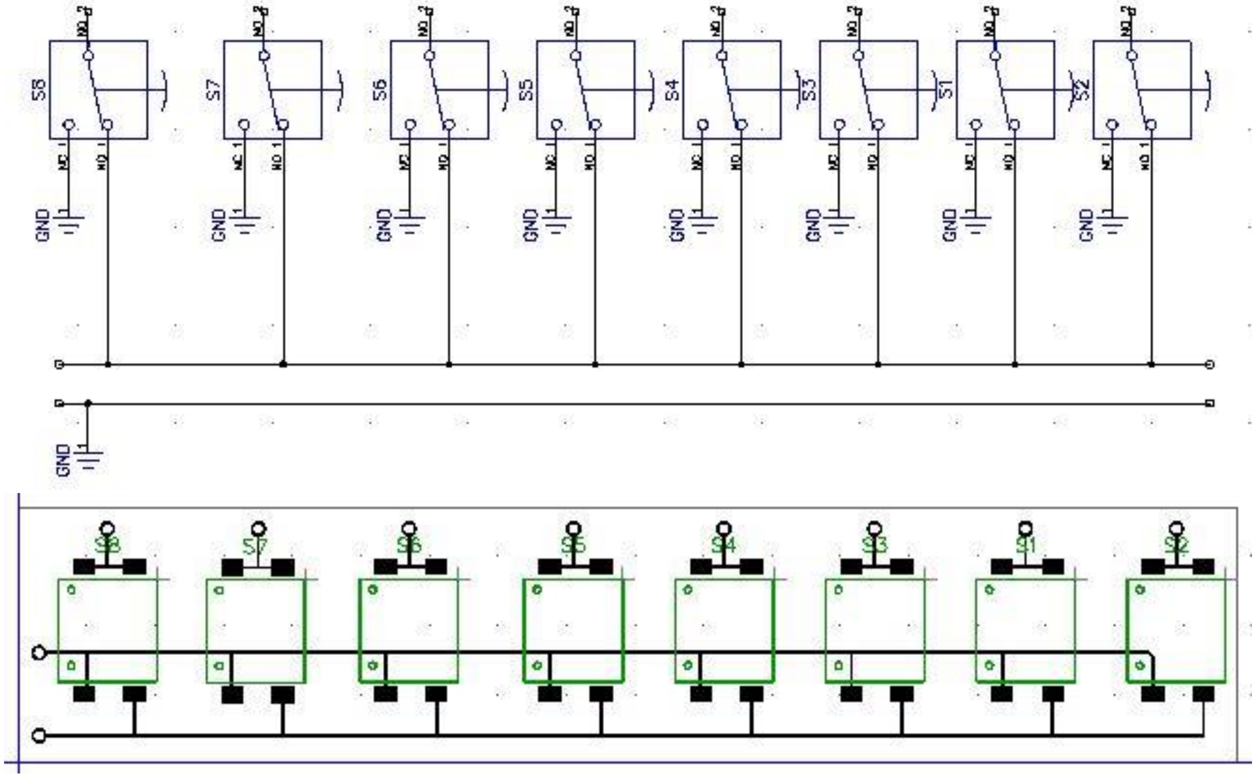


LEDs:





Buttons:



Tables:

Frequency calculations:

BUTTERWORTH	Drop D, E, A			
HIGH PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
5.46E+02	4.70E-06	62	6.17E+01	5.49E+02
LOW PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
2.71E+02	4.70E-06	125	1.25E+02	2.70E+02

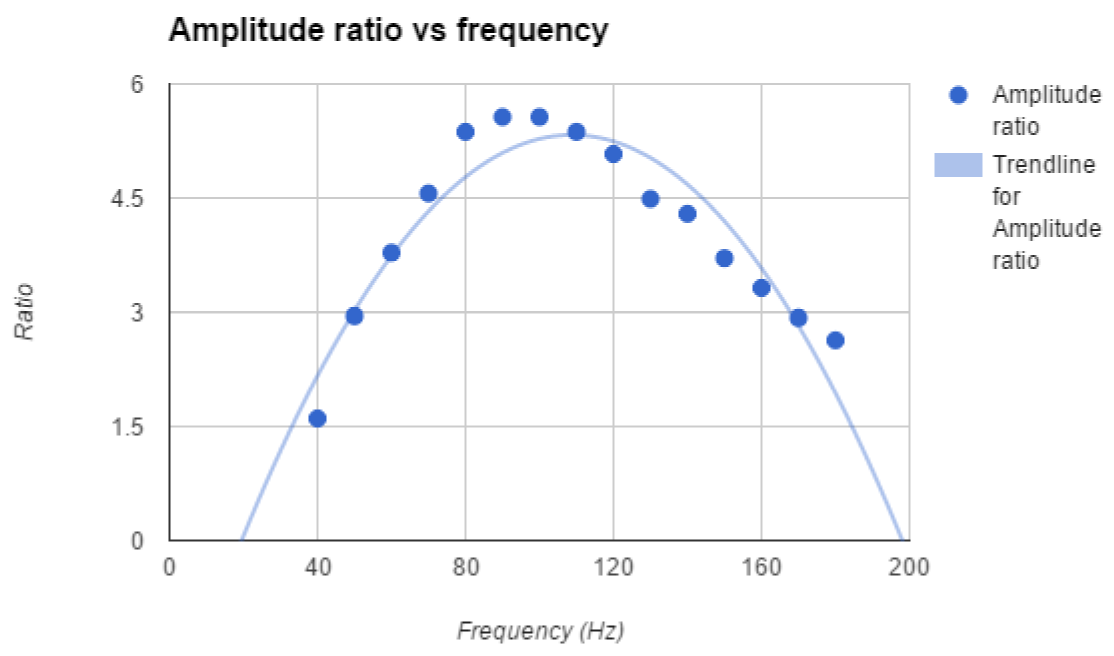
BUTTERWORTH	D, G, B			
HIGH PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
2.71E+02	4.70E-06	125	1.25E+02	2.70E+02
LOW PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
1.28E+02	4.70E-06	265	2.67E+02	1.27E+02

BUTTERWORTH	High E			
HIGH PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
1.28E+02	4.70E-06	265	2.67E+02	1.27E+02
LOW PASS				
R (Ohms)	C (Fahrads)	Desired f (Hz)	Actual Hz with R	Actual R
9.68E+01	4.70E-06	350	3.47E+02	9.76E+01

Frequency response:
Low Filter:

Frequency	Amplitude	Input	Output

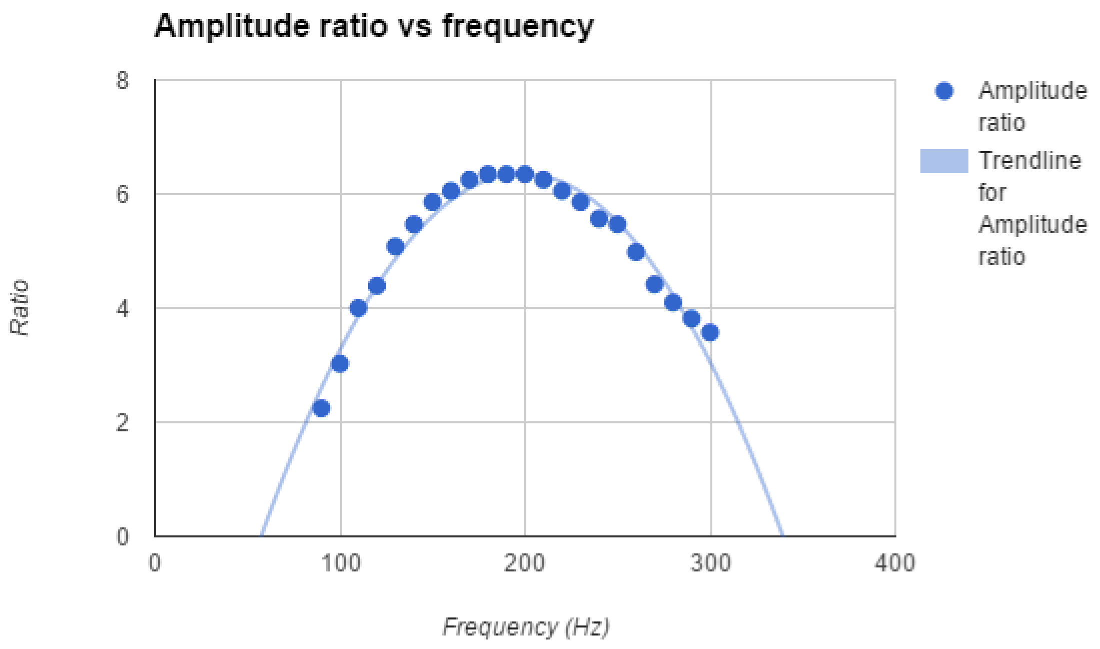
	ratio	amplitude (mV)	amplitude (mV)
40	1.604878049	205	329
50	2.951219512	205	605
60	3.780487805	205	775
70	4.56097561	205	935
80	5.370731707	205	1101
90	5.565853659	205	1141
100	5.565853659	205	1141
110	5.370731707	205	1101
120	5.07804878	205	1041
130	4.487804878	205	920
140	4.292682927	205	880
150	3.707317073	205	760
160	3.317073171	205	680
170	2.926829268	205	600
180	2.634146341	205	540



Mid Filter:

Frequency	Amplitude ratio	Input amplitude	Output amplitude (mV)

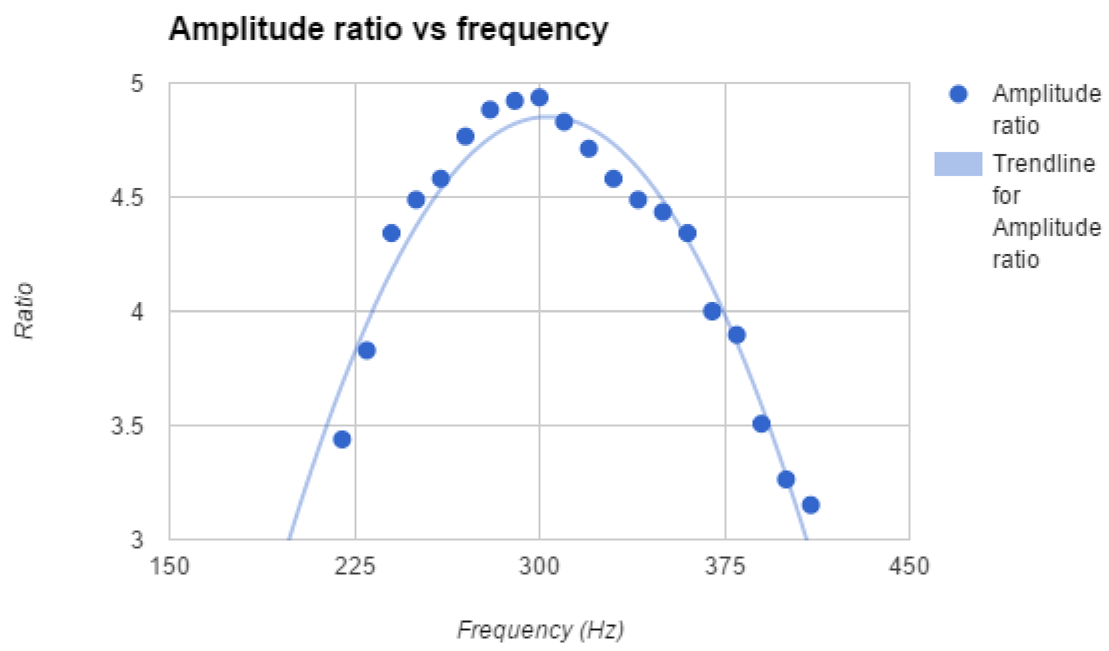
		(mV)	
90	2.243902439	205	460
100	3.024390244	205	620
110	4	205	820
120	4.390243902	205	900
130	5.07804878	205	1041
140	5.468292683	205	1121
150	5.858536585	205	1201
160	6.053658537	205	1241
170	6.248780488	205	1281
180	6.346341463	205	1301
190	6.346341463	205	1301
200	6.346341463	205	1301
210	6.248780488	205	1281
220	6.053658537	205	1241
230	5.858536585	205	1201
240	5.565853659	205	1141
250	5.468292683	205	1121
260	4.980487805	205	1021
270	4.414634146	205	905
280	4.097560976	205	840
290	3.814634146	205	782
300	3.570731707	205	732



High Filter:

Frequency	Amplitude ratio	Input amplitude (mV)	Output amplitude (mV)
220	3.43902439	205	705
230	3.829268293	205	785
240	4.341463415	205	890
250	4.487804878	205	920
260	4.580487805	205	939
270	4.765853659	205	977
280	4.882926829	205	1001
290	4.92195122	205	1009
300	4.936585366	205	1012
310	4.829268293	205	990
320	4.712195122	205	966
330	4.580487805	205	939
340	4.487804878	205	920
350	4.434146341	205	909
360	4.341463415	205	890
370	4	205	820
380	3.897560976	205	799

390	3.507317073	205	719
400	3.263414634	205	669
410	3.151219512	205	646



References:

"Butterworth Filter Design and Low Pass Butterworth Filters." Basic Electronics Tutorials. Electronics Tutorials, 14 Aug. 2013. Web. <http://www.electronics-tutorials.ws/filter/filter_8.html>.

"Dr. Calculus Technical Calculators." Dr. Calculus - PCB Heaven. PCB Heaven, n.d. Web. <http://pcbheaven.com/drcalculus/index.php?calc=st_sym>.

"C Programming Goto Statement." Programiz - C Tutorial. Programiz, n.d. Web. <http://www.programiz.com/c-programming/c-goto-statement>.

Floyd, Thomas L., and David M. Buchla. "Diode Rectifiers & Basic Op-Amp Circuits." Electronics Fundamentals: Circuits, Devices, and Applications. 8th ed. Upper Saddle River, NJ: Prentice Hall, 2010. 735+. Print.