ECE167L

Xingyu Zhou

Max Dunne

2020/01/11

Lab 0 Report

Introduction:

This is the first lab of the quarter, and it's designed mainly for us to get familiar with various equipment such as the microcontroller board (UNO32 with io shield) and soldering iron, as well as the software (MPLAB X) that is needed to program the microcontroller. There are six parts in this lab, the first two parts are mainly introductory steps on how to set up the UNO32 microcontroller kit. The rest four parts involves a little bit of soldering work as well as some coding and will be specified in the following paragraphs.

Part 1: Hello World!

This is a pretty common task for basically all the beginner programming practice. In this part of the lab, we need to write codes that display the string "Hello World!" to the serial port. To do that, the program needs to be downloaded onto the UNO32. This can be done by using the PICKIT3. However, instead of using the PICKIT3, a new software, the ds30 bootloader is introduced. Ds30 bootloader acts basically like a PICKIT3 but it's much more convenient. After burning this bootloader on to the UNO32, we can ditch PICKIT3 and use the bootloader instead. Ds30 bootloader also comes with a terminal so if the program is downloaded, "Hello World" should show up in the terminal.

Part 2: Hello A/D

UNO32 comes with an onboard potentiometer and the objective for part2 is to read the potentiometer and print the readings to the io shield that's also on the board. To achieve this, we need to find a pin on the UNO32 that is connected to the onboard POT, read the value on the pin, and print it on the io shield. The pin that is connected to POT is pin A0. Using the AD library provided, the first step is to use the function ADDPin(). The second step is to use the ReadADPin() function to acquire the reading. There are many ways to display the reading on io shield. The method from CSE13E is to first store the reading into a char array using sprintf(), and then use OledDrawString() to display the reading. One important thing to notice is that, the oled screen won't clear itself, which means if the reading goes from 0 to 1023, which is the maximum, and then comes back to 0, expect to see some strange numbers since they are not updated. To avoid this, you can either clear the screen each time, or move the number further right to avoid overlapping.

Part 3: Tone out speaker

The objective of this part is to write a program that controls a speaker which is connected to the UNO32. This is the part where the soldering iron comes in. The part that need to be soldered is an audio amplifier PCB (tpa2005d1). Since the pins can be moving around when being soldered, its recommended to secure it on the breadboard. A speaker will also be connected to the microcontroller through the audio amplifier PCB so again, it's recommended to use a breadboard. For the wiring part, Uno32 will be used as the power source. First, connect the 3v3 and the GND pin on the Uno32 to the common power/ground on the breadboard. Then connect the PWR pin on the PCB to the common power/ground as well. Second, the speaker should be connected to the "Out" pins on the PCB board. Since the sound of the speaker will be decided by the frequency of the PWM, we will be connecting Pin 3 (one of the PWM pins) on the Uno32 and the "In" pins on the PCB board together. Doing so will allow us to control the sound of the speaker using the PWM frequency. Lastly, by using the ToneGeneration file provided, different sounds can be played using the speaker. Block diagram will be provided at the end for reference (see figure 3).

Part 4: Tone adjusted via POT

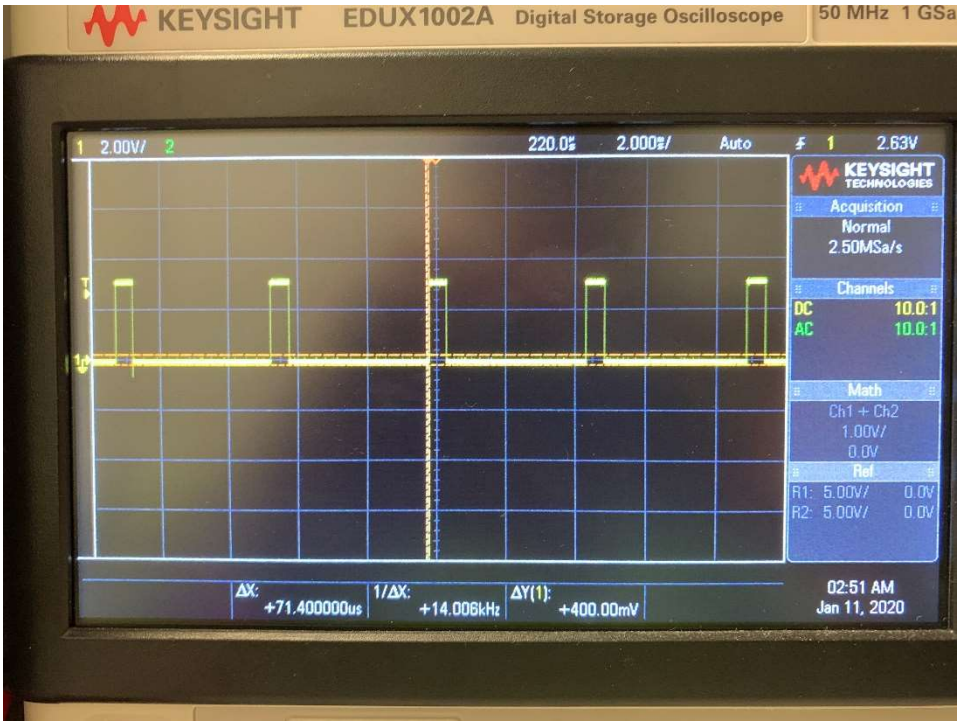

Figure 1: Waveform of the speaker output with software filtering enabled

The objective of this part is to control the sound of the speaker using the onboard POT and develop a software filter to eliminate the scratchy noise generated by the rapid change of the frequency. The first thing to do is to write the control program. This program is a combination

of part 2 and part 3—the PWM reading is acquired from the function AD_ReadADPin() and will be used as an input to the ToneGeneration_SetFrequency() function used to generate the sound. One thing to notice is that as the frequency increases, the sound will be higher and higher and eventually “disappeared”. In order to hear the sound all the time, we need to limit the range of the frequency, which is around 20 Hz – 1000 Hz (different for each speaker). Another thing to notice is that the speaker will make scratchy sound (noise) when the frequency changes. Thus, a software filter is needed to eliminate that situation. There are many ways to achieve that. One of them is taking the average—instead of changing the frequency right away, we can take a certain number of samples, for example, 20, and use the average of that sample to feed the speaker. I used 50 samples but still couldn’t eliminate the scratchy sound at higher frequency. The second method is to increase the gap which the PWM changes. For example, when it’s being increased, the frequency will change like this:0,1,2,3……1000 (with an increment of 1). By increasing the gap to 20 (or maybe higher), the sound generated will be much smoother since small flickering will not affect how the speaker sounds. To do that, we need to take a second measurement of the PWM reading and subtract it from the first PWM reading. When the difference between these readings are greater than 20 or less than -20 (in my case is 20), the second PWM reading will be sent to the speaker. Block diagram will be provided at the end for reference (see figure 3).

Part 5: Basic filtering of output (RC Low pass filter)

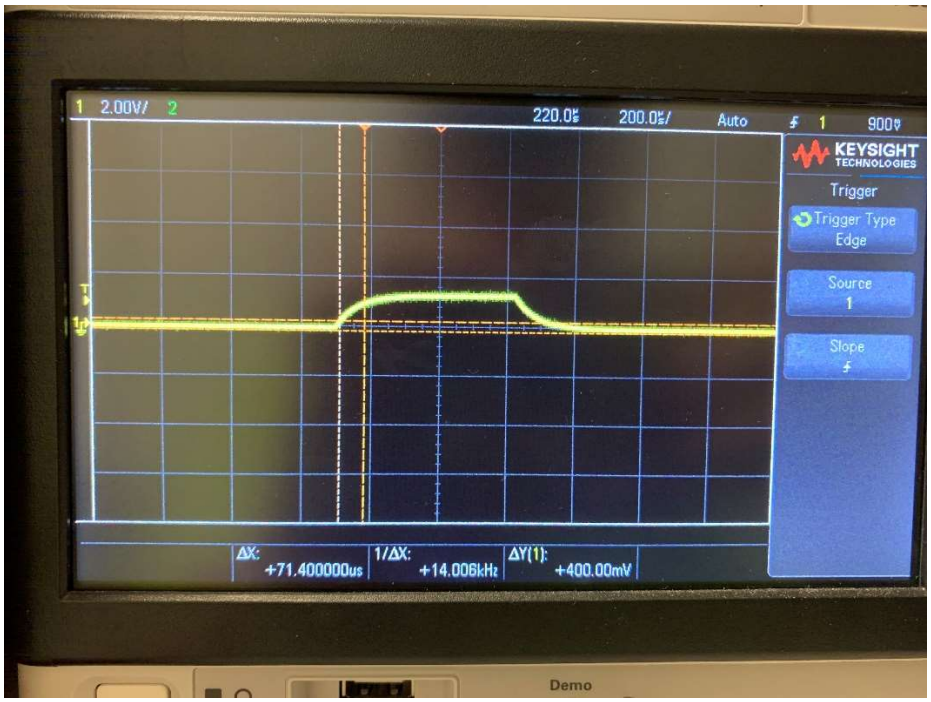

Figure 2: Waveform of the speaker output with RC low pass filter installed

The code for part 5 is the same as the code for part 4. We just need to build a hardware RC low pass filter to make our speaker sound better. In order to decide which resistor and capacitor to

choose, this equation is used: $f = \frac{1}{2*\pi*R*C}$. In this equation, f is the cutoff frequency, which should be higher than the frequency which the speaker will be able to make a sound (told by one of the TA). In my case, the highest frequency that my speaker can make a sound is 950Hz. Thus, I chose my cutoff frequency to be 1061.6Hz (by calculation) with a resistor of 15k Ω and a capacitor of 10nF. The filter reduces the volume of the sound of the speaker and the attached photo shows how the waveform changes. Instead of the square wave in figure 1, the waveform tends to be smoother. Block diagram will be provided at the end for reference (see figure 3).

Part 6: Make some music

In this part, buttons will be added to make the sound more customizable. There's a function provided in the BOARD.h file called BUTTON_STATE(). This function will return a 4-bit number representing which button is pressed. For example, 0001 means button 1 is pressed and 0010 means button 2 is pressed. By using this information, we will be able to customize what sound the speaker will make when a button is pressed. Again, the code from part 4 will be reused (need the software filtering). We were also asked to use the POT value as an offset so that we can make better tones. This is achieved by adding the POT reading to whatever preset tones I have for each button. Block diagram will be provided at the end for reference (see figure 3).

Conclusion

Even though this should be the easiest lab, it still took me sometime to figure it out since I took CE13 and EE101 almost two years ago and needed sometime to refresh the memories. In general, this is a fun lab and I'm glad we are given the time to get ourselves familiarized with everything.

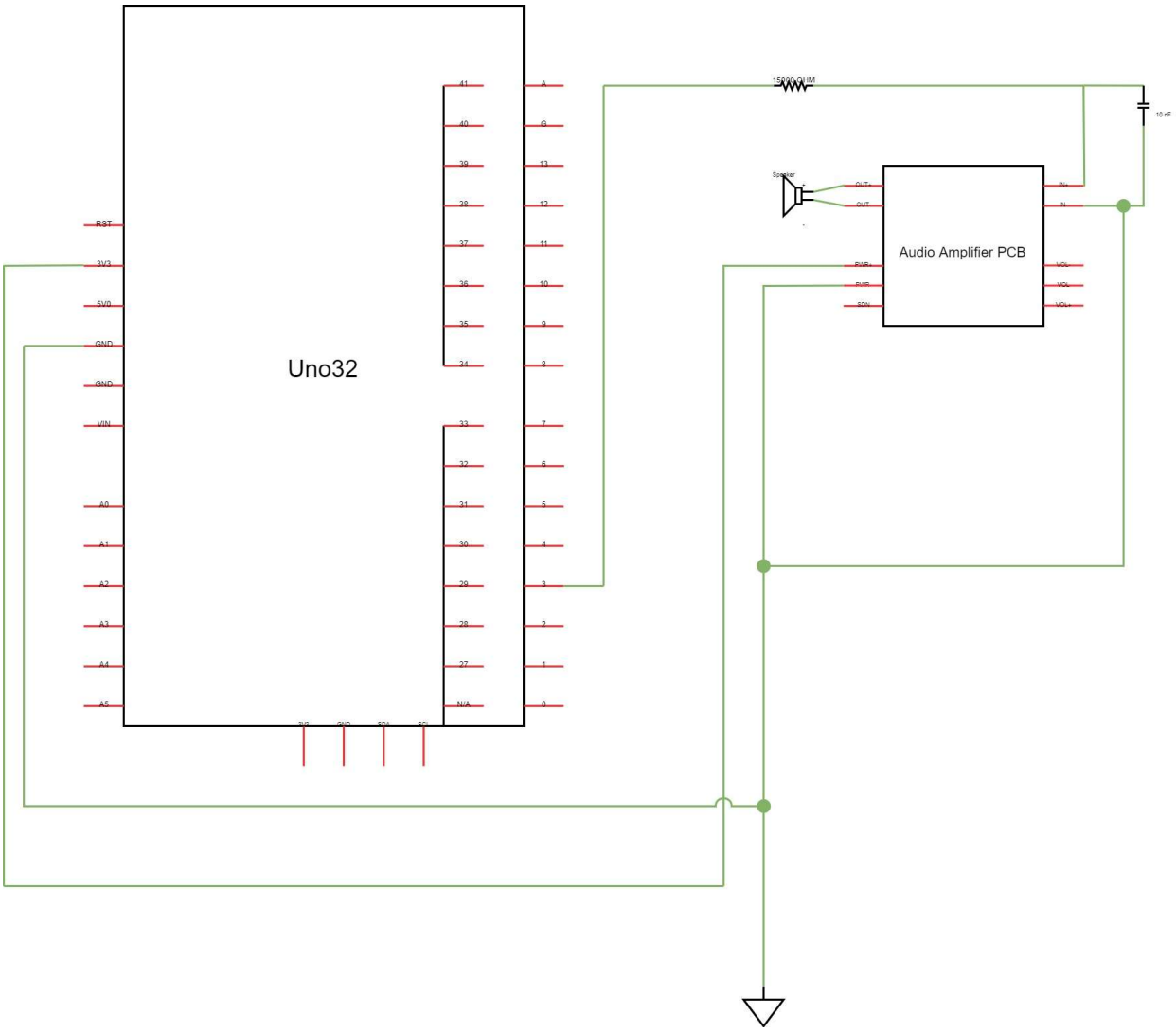

Figure 3: Lab0 Block Diagram (for part 4 and so on)

Note: For part 3, connect Pin 3 to IN+ directly.