CSE 121 Final Project Report

Jacob McClellan

December 12, 2020

Contents

1 Introduction

In this final project I built a two channel oscilloscope. This was the final culmination of all the previous labs, and included just about every concept explored before, barring freeRTOS. This oscilloscope includes frequency measurements, GUI implementation, DMAs, an ADC, potentiometer, a state machine a two way communication system, and a very large mess of wires. the final result was a two channel oscilloscope, with a controllable trigger mode, adjustable xscale and yscale and scrollable waveforms, which can be read as low at 100 Hz up to several thousand Hz.

2 Design Process

2.1 Creating Two Channels

Early this year in lab 2, I had created a single channel frequency reader. The design of this included a single ADC module with one input, and a single DMA with two descriptors writing to separate ping-pong buffers. This program was able to successfully read in data from the AD2, store it in buffers, and translate some frequencies. This served as the basis for my oscilloscope design. My first step was to take this design, improve upon its shortfallings, and add a second channel. The previous design included one input, one DMA, and two buffers. The new design includes four inputs, two DMAs and and four buffers. The four inputs include two signal inputs and two potentiometer readers, which will be covered in a later section. The DMAs transferred the data from the ADC signal inputs to buffers within the main file. I also only included a single interrupt connected to the first DMA. My reasoning behind this was, the DMA are of the same size, with identical descriptor protocols, are hooked up to the same ADC, and would thus be creating interrupts simultaneously, so I only needed one ISR for the both of them.

My first test was to make sure I could successfully read in data from both channels. Using the UART to communicate with the terminal, each time an interrupt was triggered, the ISR set a ready flag, and the main loop would print out the data the from the buffers. I made sure each of the four buffers were receiving data, and that the data received lined up with their expected values and patterns.

2.2 Measuring Frequency

The next step I took was to fix my frequency measurement function from lab 2. In that lab, I had measured frequency using inflection points. I'm not sure why I did, as the method only worked on sinusoidal waves and was pretty ineffective with those. I decided to employ a new method which would hopefully work on all uniform waveforms: midpoints. This method would require a moderate amount of computation time, but would hopefully be worth it if effective. My initial idea was that, all uniform waveforms pass through a midpoint value each half wave length. If I could measure the time between a pair of midpoints, I could extrapolate frequency. The function that calculated frequencies takes in three inputs, two pointers to the buffer arrays to be interpreted, a pointer to an integer array which would store the results.

The function ran through the buffers once to find min and max values. The midpoint is the difference between those values. The function would then run again through the buffers, and would find the first two locations where the function passed through the midpoints. Using the difference between these two points, and the sampling frequency, I could get a good estimate for frequency. This method worked better than the inflection method, worked on all the various signal shapes, but was also only accurate about two thirds of the time, and couldn't find frequencies below 500Hz. The first issue was caused by random noise which could throw off where the function thought the midpoints were, and the latter because there just simply was not enough data collected to view a full wave form. I planned to fix the problems with two changes. First, I wanted to change the buffer size from 256, what it was from lab 2, to 4096, a 16x increase. Secondly, I came up with a new way to calculate frequency from midpoints.

In order to increase the buffers fro 256 to 4096, I needed to increase the amount of data a DMA descriptor was sending. I up-ed each descriptor by changing it from 1 loop of 256 to 16 loops of 256.

My new method of calculating frequencies relied not on counting the distance between a pair midpoints, but the total number of midpoints. This, along with the increased buffer size, would increase my sample size for measuring, and would reduce the effect to which noise would throw it off. This method works since the the sampling rate and buffer size are constant, so the buffers will have a constant time interval. Each sample should have

about two midpoints each wavelength. My new method gave me much more reliable and steady results for the frequency measurements, and could measure 100Hz frequencies reliably. My only concern was noise might make the program detect too many midpoints every now and then, but in testing, this doesn't seem to ever happen at any frequency, or at least not to a noticeable extent.

2.3 GUI and Waveform Display

2.3.1 Start Screen and Background

After I was done creating the buffers and had a working frequency function, I moved on to designing my GUI. I began by connecting the wires, and running the demo code provided. Once everything was connected properly I could see the demo GUI, and I moved onto creating my own implementation. I imported the necessary files and began creating my own GUI functions. My first goal was to create my own start screen similar to the one from the demo, just so I could get a feel for programming the GUI. The start screen was drawn using its own function. It displayed the project name, a subheader and my name as a credit. I experimented with moving text around to different locations, changing color and pen size. After I had a decent understanding of how those methods worked, I moved on to implementing the background and grid functions. The grid function was largely based on how the demo drew its grid. The background function placed a rectangle over the background and called the grid function. I set up the program to initialize all the modules (ADC, UART, GUI) and display the start screen for one second before drawing the background.

2.3.2 Drawing Waves

The next task was to draw the a waveform over the background. My first goal was to convert the buffer data into displayable form. This required scaling, as the values in the buffers were far larger than how many pixels the screen is high and contained more data points than the screen is usable pixels wide (maxpoints). At first, I just wanted to display the entire waveform on screen without including the xscale and yscale factors. I did this by saying for each pixel incrementation, the data buffer increments by the ratio of total bytes to pixels. If the size of the number of pixels that can be used for display is N, the function would take N equally spaced samples of the entire wavelength. I then scaled those samples down by a factor of 50, which brought it into a reasonable pixel range. This conversion occurred in

its own data conversion function. This function took in a raw data buffer, a returned an integer array, the size of maxpoints. I filled a separate x array in the main function, and with those two arrays, I created a spline. Displaying this spline, I saw my waveforms. They were displayed at hard coded locations at first.

I then tried to make the waveforms continuous. Within a for loop, I stored the old data information, got new data information, deleted the old spline, drew a new one with the background color with the old data, and then drew a new line with the new data. When I ran this program, it seemed to run fine at first, but after about 10 or so iterations, it would freeze. I was told this was caused by a RAM error due to excess spline instances, but I was unable get rid of this error by modifying my function. I then took a separate approach. I created my own draw wave function, which took in the converted data array, and used the drawLine() function to create a graph. This new approach was slower, but got rid of my previous issue, at the cost of creating a slight flickering each time the line was updated. After some failed optimization attempts, I decided that since this was a minor cosmetic issue, and the waveform was still clear and continuously moving, I would move on.

2.3.3 Scaling

The next step was to pass yscale and xscale as parameters to my data conversion function. I already had the infrastructure to implement it, in order to have yscale, I needed to change that factor of 50 to some calculated amount, and my incrementation ratio of bytes count / maxpoints to some other ratio related to xscale. For yscale, I needed a buffer value / pixel ratio. I found this value using an ADC scale constant, the yscale, and the size of a y division in pixels. Dividing the buffer values would covert their value to a desired pixel size. In the case of xscale, I needed a samples/pixel ratio. This was found using the sample rate, the size of the x divisions in pixel, and the number of bytes to be converted. This ratio would be the amount to increment the data array for each pixel value. That value would be divided by the yscale. It was conceivable that the xscale value could exceed the total number of data points, if stored waveform data was not long enough to meet the xscale requirements. I added a %Byte count, so that it would loop back upon itself in that case.

I ran some test programs and it seemed to work right away. At this

point, I had not included a communications system, so I was hard coding in xscale and yscale values. the scaling values, along with the calculated frequency, were also given a separate update function, which would display their values on the screen when called by the main function.

2.3.4 Trigger Mode

The next step was to implement the trigger mode with hard coded values. This was an easy enough inclusion it seemed to me. In the data creation function, I included the variables, mode and slope and level. The program would see, if mode was 1, then it would do trigger stabilization. I did this by including a for loop that started at the beginning of the data, and if the program passed through the adjusted trigger level, either down or up depending on the slope parameter, it would mark that index as the beginning point. If mode was zero, this portion would not run, and the beginning index would remain 0. The for loop that creates the data array described earlier, would be adjusted so that its starting index for the data would be the new index.

I tested it with a variably of signals and hard coded parameter. it all seemed to work well enough first go, some signals maybe a little shaky, especially at higher frequencies.

2.3.5 Scrolling

Finally I wanted to include scrolling, and the waveform display portion would be complete. My main issue I had was that my kit only had one potentiometer, although I came up with a decent work around. The scrolling implementation was simple enough. I had a function that took in both channels y position pointer, and a channel parameter. This channel parameter would be the channel that is currently getting scrolled. So if the channel parameter was 1, then channel 1's y position pointer would be the one being adjusted by the ADC value. This value would be between 30 and 210 pixels as a min and max position on the GUI. The plan would be for this to be an addition parameter implemented when the communications system is created. This y position parameter could be included as another parameter in the data creation function. The new pixel location would then be yposconverted data. Again to my luck, this worked upon first pass, and provided me a convenient was to control the y positions with only one potentiometer.

2.4 Communications

The only thing left to implement was the communications system. I had been able to print to terminal for a while, as the UART was already connected for debugging purposes. the goal was now to read in input. I had been communicating with the UART through putty, so I turned on the putty input echoing. In my main loop, I included a revision at the beginning, that if the RX FIFO was not empty, that I would read in those values, and pass them back to the terminal as individual chars, to make sure I was consistently getting in those chars. I then went about putting those chars into a string. This was done using a separate getLine() function I crafted. i then tokenized that string using spaces. In the getLine() function, I had it so when it found the end character, it would replace it with a space to facilitate easier organization, and got rid of some bugs I was having with the strtok() function.

I now had to interpret my newly tokenized strings. This was done by brute force using a bunch of if / else statements. The statements would check the first or first and second tokens for the correct command, such as "set mode", and then interpret another token as a parameter as needed e.g. "run". All the while throwing incorrect input messages wherever needed. If the input was valid, it would change the corresponding global variable accordingly and print a success message to the console. One by one I implemented each possible input, and tested their affect of the waveform. Since the global variables' affects had already been tested, it was just a matter of getting the communication function to change them correctly. The only major bug I experienced was if the "stop" command was entered, the RX FIFO would not have the time to fill up before being accessed again when a user includes new input, which was fixed by adding in some delays.

At the end of implementing the communications function, the program was in full working order and all that needed to be done was some cosmetic code clean up.

3 Final Design Overview

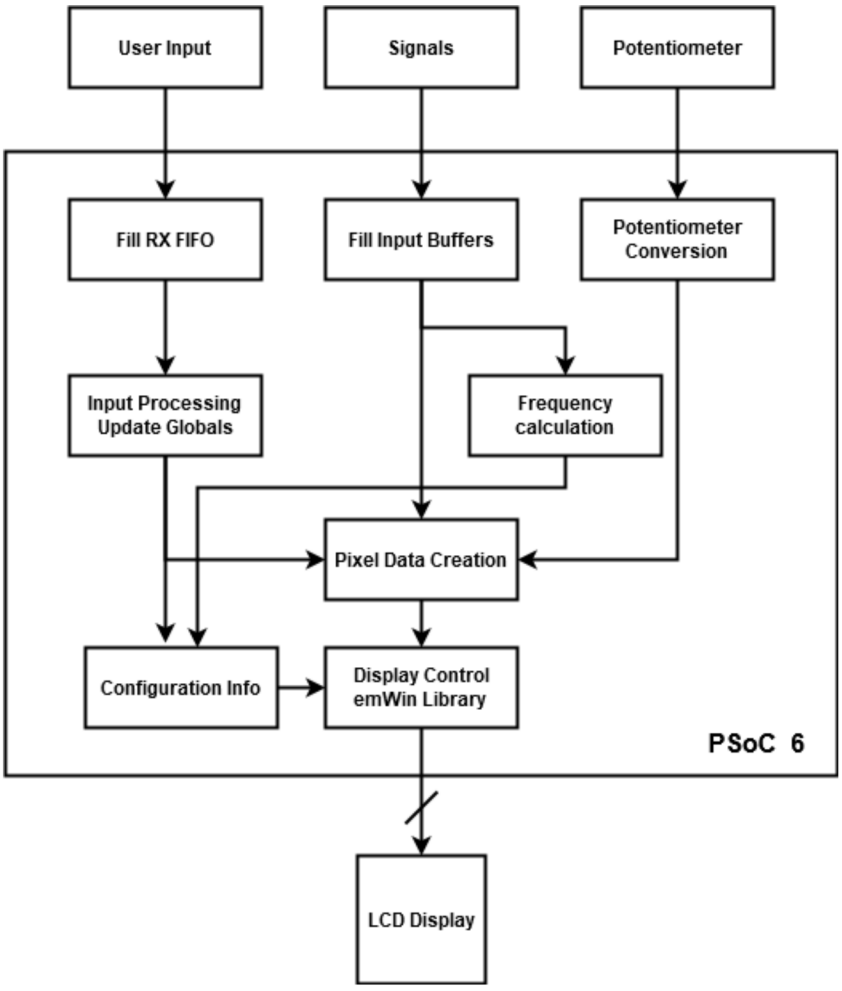

Figure 1: Block Diagram

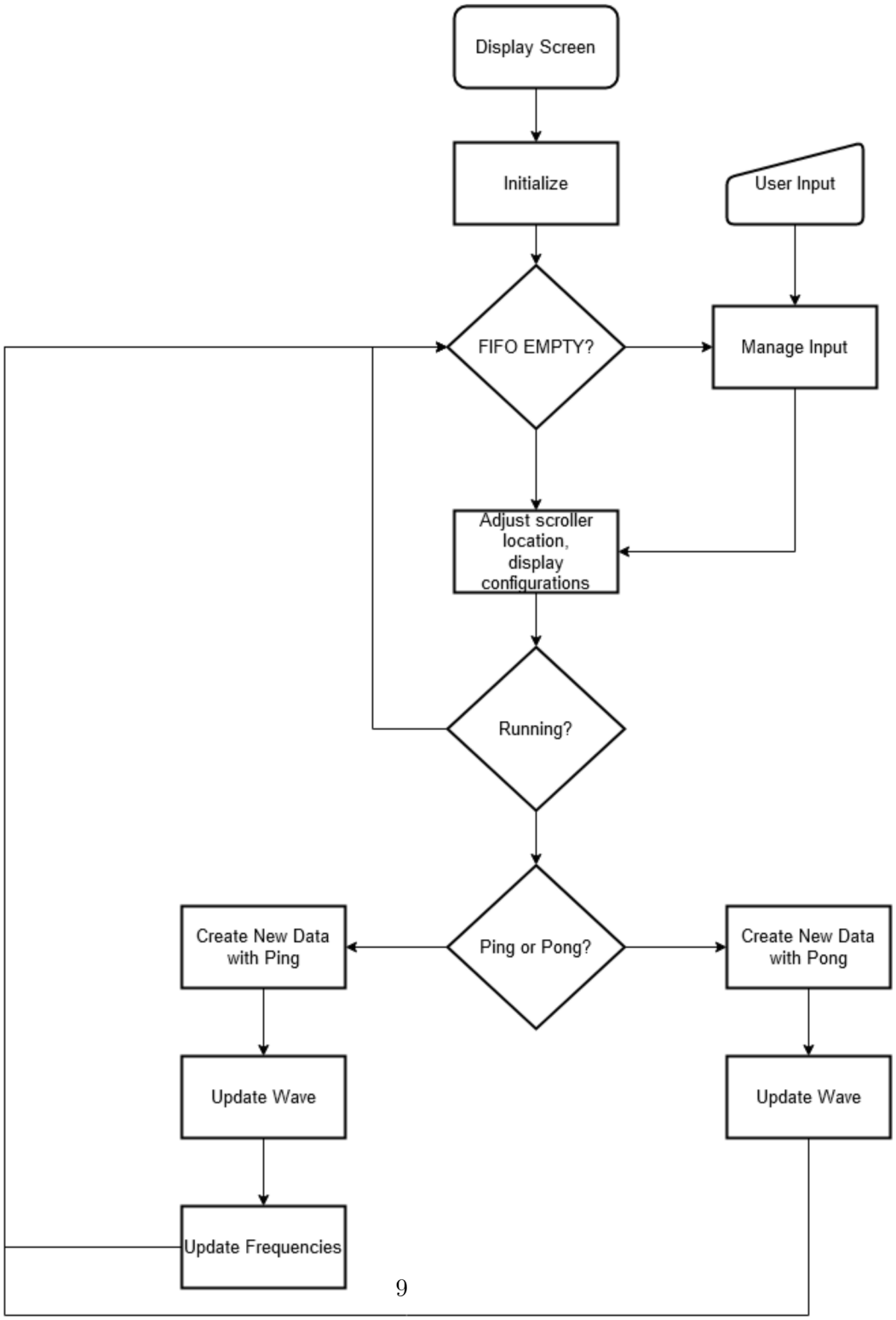

Figure 2: Flow Diagram

Above is the final flow diagram in the main file. The program displays the start screen, calls the initialization functions, and begins the main loop. The main loop begins by checking if there is user input in the RX FIFO of the UART. If there is, a manageInput() function will decipher it and update all the global variables as needed, and print some corresponding output to the terminal. The program then polls the ADC for the potentiometer reading, and updates the vertical position variables. The program then updates the frequency and scale readings on the display. The program then checks if the program is suppose to be running or not. If not, the program repeats the previous steps until the user starts the program again. If the program is running, the program checks which of the ping-pong buffers was most recently updated. Using the most recent buffers, it will then update the display information and update the waveform on the GUI. Finally, if the most recent buffer was ping, it also has the extra task of updates frequency measurements. This is slightly time intensive function, so to save half the time, it only runs every other iteration while the program is running.

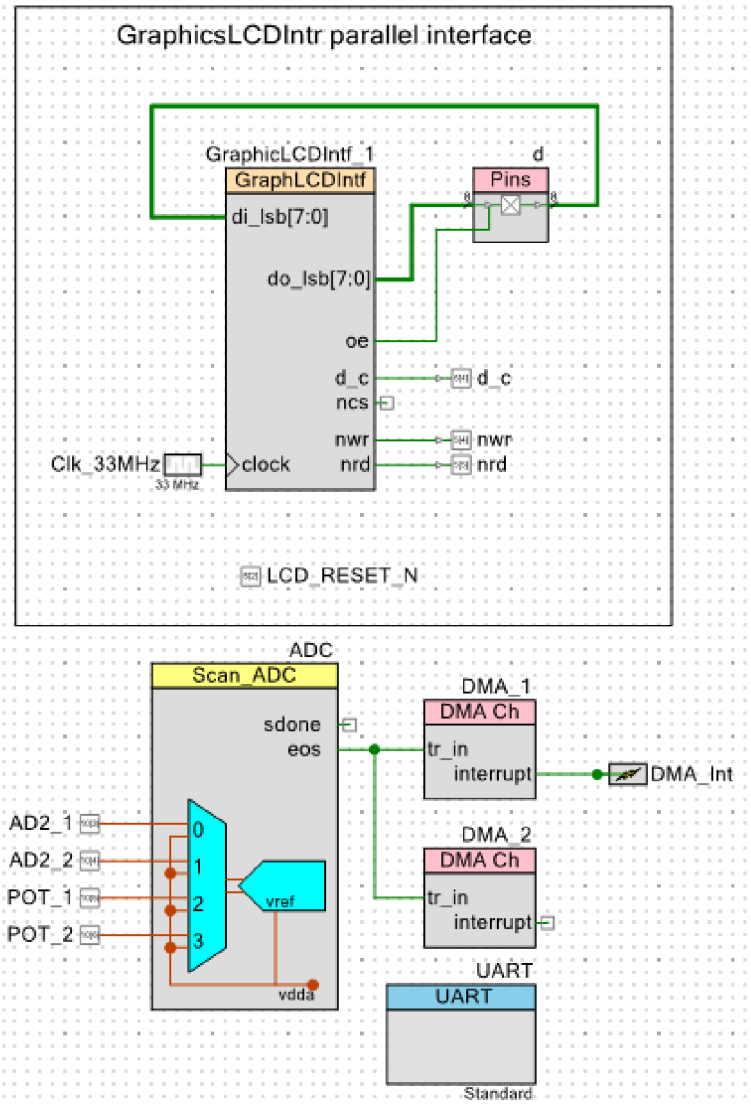

Figure 3: Top Diagram [2]

While there are two potentiometer analog inputs, only one is used. The other is a vestigial organ. While there is only one interrupt, the ISR in the main function clears the pending interrupt from both DMAs, since they are in sync.

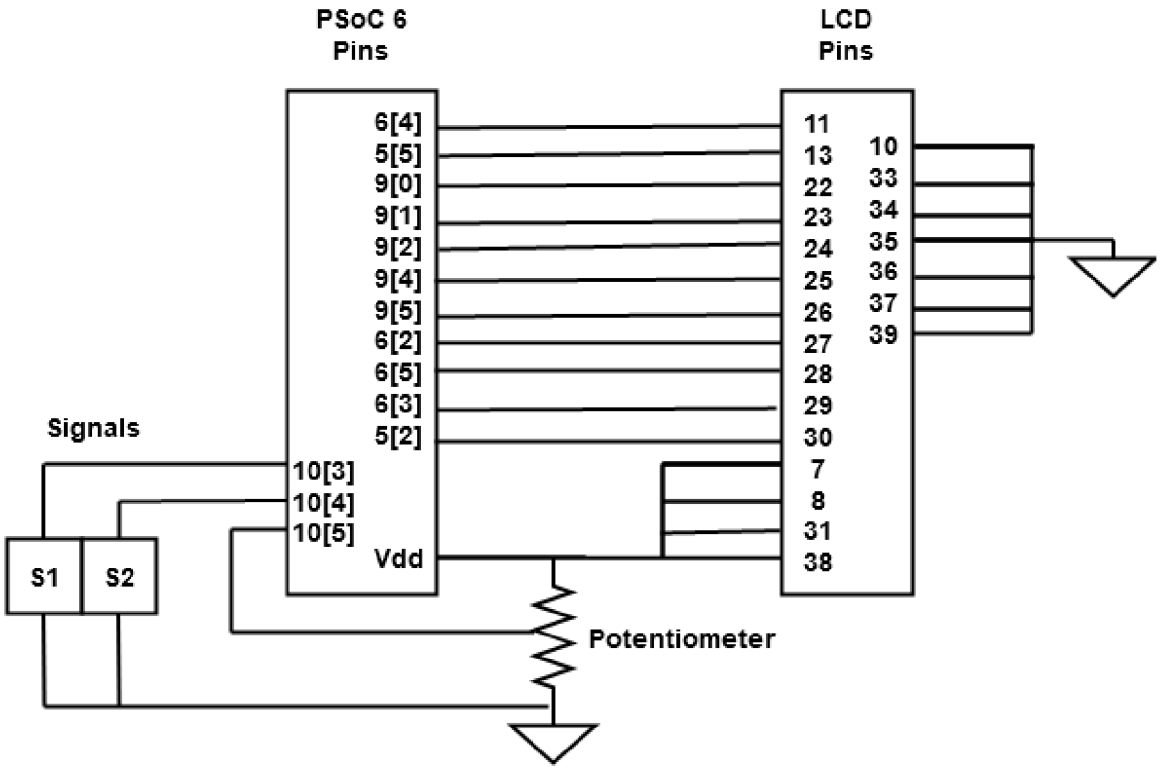

Figure 4: Wiring Diagram. S1 and S2 are the signal inputs.

4 Results and Conclusion

The final product overall accomplished the set out goals. The waveforms were displayed accurately and could be controlled with xscale and yscale settings. The triggered mode worked well enough. The work around for the dual scrolling works very well, and the frequency measurements were decently accurate. One lingering problem is the flickering of the waveforms each time they update. One of the issues that was brought up during check off was the fact that my channels were not in sync with each other when one was being triggered, even when they are the same frequency. This was because my design only delayed one of the waveforms when it was being triggered, which would mean one could be stable, while the other drifted. Another issue was that I was rounding my frequencies unnecessarily, but this was mostly benign.

If I were able to make further improvements, I would first try and re-implement spline function instead of drawing individual lines. I'm sure with better effort and more time I could get rid of the memory issue. Doing this would hopefully save a lot of processing time, and reduce much of the flickering. I would also fix the out-of-sync trigger issue. This would be a pretty easy fix. Right now the program only offsets one buffer when doing the trigger calculations, I just need to extend that to both buffers.

References

[1] emWin Library, Segger Microcontroller Systems

[2] "Final_Project_Demo.cywrk", Anujan Varma