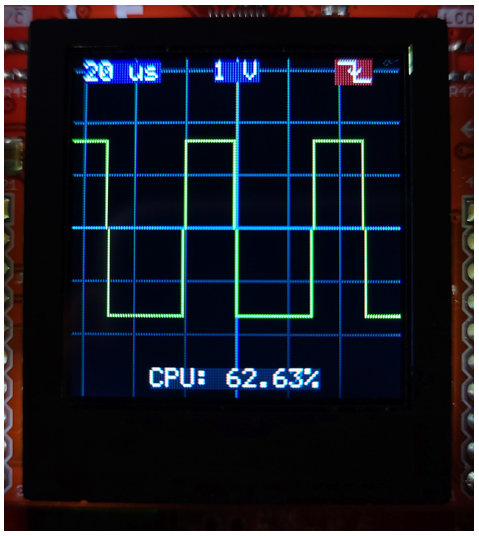
Slater Campbell
ECE3849 Lab 1
4/15/2021

Lab Objectives:

The objective of this lab was to create a 1 megasample per second digital oscilloscope with trigger synchronization, at least two trigger types, and live CPU usage display. I was able to accomplish all of these things, including the extra credit for multiple voltage scales. I picked a green trace color and dark background, reminiscent of vintage oscilloscopes.



Discussion and Results:

Creating a 1 Msps oscilloscope is challenging, especially on a system that only runs at 120 MHz. The ADC ISR is called 1 time per 120 clock cycles. The ADC needs to sample and save data to a buffer within a very tight number of clock cycles in order to not miss deadlines, and it still needs to leave time for other tasks such as reading buttons and drawing to the display. Using RMS fixed priority, since the ADC ISR has the shortest period, it needs to be the highest priority. The button ISR is second, with a period of 200 Hz. There are 8 ISR priorities for the Cortex M4F, in steps of 32. Lower number indicates a higher priority. I chose a priority of 0 for the ADC ISR, and 32 for the button ISR.

For configuring the ADC, some starter code was supplied. Many functions were missing arguments, but helpful comments said what the settings should be. Consulting the guide for library functions, I filled in the arguments for each function. On one of the configuration functions, there was a comment "`enable interrupt, and make it the end of sequence`" on a new line, and I didn't realize that was referring to arguments that went in the function above. This resulted in my ADC ISR never being called, and after going through each function and checking the arguments I discovered what I was missing.

Once I configured the ADC, I started working on the ADC ISR function. The two missing parts was the interrupt flag reset and reading the sample from the sequence 0 FIFO to the circular buffer. For the interrupt flag, I consulted the ADC register map in the datasheet and found register ADCISC. Using the format given in the lab document, I found the in C code it was called ADC1_ISC_R. I confirmed this in CCS by going to its definition, and from there I found that the specific bit field I wanted was ADC_ISC_IN0. "`ADC1_ISC_R = ADC_ISC_IN0;`" clears the interrupt flag. I used the same method of consulting the datasheet and then browsing header files in CCS to find that the ADC1 sequence 0 FIFO was register ADC1_SSFIFO0_R. I then just set the global circular buffer at each index equal to the content of this register. At this point I verified that gADCErrors stayed static when the program was running, and would increment once when the program was paused.

Next, I completed the rising trigger function which was partially provided. First, I added an atomic read of the index into a separate local variable (in case no trigger is found, x must be reset to the initial index position, but the global index will have changed). Then, I found the constants for the LCD width, and used them to set the start of the trigger search as half the display width earlier than the global buffer index. Then, the function scans backward through the buffer, looking for a crossing of ADC_OFFSET (2048, or half the ADC max value of 2^12). The desired crossing direction was based on whether a rising trigger or falling trigger was desired, and I made one function for each with the appropriate inequalities.

Once a trigger is found, a couple of functions I designed are called in a certain order. First, a function called copyBuffer() takes the newly found trigger as it's only argument. It then saves half a screen width of data on either side of the trigger from the global ADC buffer to a separate global buffer, used only by functions called in the main() loop. This is accomplished by taking the trigger, subtracting half the display width, and then wrapping that value, giving the index of what will be the leftmost pixel of the screen. Then, it traverses pixel by pixel for a full screen width, copying the raw ADC output to a global array "rawWaveBuffer[]" which is used by the next functions.

After copying the buffer, a function called scaleWave() is called. This function takes an argument of the voltage per division in the form of a float. It then uses the raw ADC output in rawWaveBuffer[] to produce a scaled output in another global array called displayWaveBuffer[], where the index is the x coordinate and the value is the y coordinate. Each time it is called, it calculates the scale factor based on the fVoltsPerDiv argument and then uses that scale factor for the entirety of rawWaveBuffer[].

Before drawing the wave, we need to draw the grid. If we draw the grid after the wave, the grid will be on top. To make the grid, I made a function that iterates using a for loop and multiplies the loop index by 20, giving me vertical and horizontal lines on a 20 pixel grid. The X and Y axis lines are then drawn again in a lighter color.

Now we can draw the wave. We have an array containing the coordinates of every pixel of the wave, produced by scaleWave(). To make this easy to interpret, instead of drawing the raw pixels, lines are drawn using each pixel as an endpoint. To do this, I made a function called drawWave() which takes an argument of type tContext, which is a pointer to the display buffer used in the graphics drivers. It then uses the global array displayWaveBuffer[], and draws a line between each coordinate by accessing the current index and then the one after it to get the start and end point of each line. All of my display functions work similarly, in the sense that they take a pointer to the screen buffer (type tContext), and return nothing.

I also created a couple other display functions. triggerIcon() takes a parameter for the current trigger mode, in addition to the screen buffer pointer. It then uses a series of line draw operations to draw a rising trigger or falling trigger in the top right corner of the screen, complete with a background. displaySettings() and displayCPUuse() each write various text information to the screen, and the values they write are already stored in global variables so they are not passed with parameters. This is safe because these globals can only be updated by the main loop.

For CPU load calculations, I used the starter code given and made a function called cpu_load_init() to set up the timer. In the starter code, the timer is preloaded to gSystemClock-1, which corresponds to 1 second. I changed this to be gSystemClock/100 -1, which gives a 10 millisecond interval.

For button handling, I used the starter code for the FIFO. In fifo_get(), I fixed the shared data bug by making fifo_head only receive one atomic write, which ensures the value is always valid. I then implemented fifo_put() in the buttonISR, so that if one of the buttons I'm using is pushed, it adds a corresponding character to the FIFO. In my main loop, I have a function that checks whether fifo_get() returns true. If it does, then there is data in the FIFO that must be read. In this case, it dereferences the pointer supplied to fifo_get(), and compares that value to the various characters to see what operation it should perform. This could be a change to the voltage scale, or the trigger mode. This weekend, I plan to add the variable time scale and an adjustable ADC_OFFSET, just because I think it will be interesting.

Conclusion

I really enjoyed working through this lab, solving issues as they came up, doing the visual design for the display, and thinking about the best way to do each thing. One interesting issue I ran into was that my display flickered a lot when I first got waveform drawing working. I realized I was missing a use of the wrap macro in my copyBuffer function, resulting in a discontinuous waveform.

In this lab, I gained experience in implementing the concepts we've been learning in class, and dealing with the challenges this presents. I also gained a comfort with browsing the userguide and header files to learn which registers to use, how to use each function, or how that function works.

One thing I think could be improved in the future is the signal source. The square wave is not very interesting, and I think that implementing a couple other sources could be more engaging. Over the summer, I'm interested in trying to implement a second analog input with the same ISR so I could build an X/Y scope. Then, I could attach a left and right audio source, and try and display some music made to look cool on oscilloscopes. I would be surprised if I could get it to look good on a digital scope like this, but I think it would be a fun project!