CSE 121 Oscilloscope Project Report

Scott Oslund

8 June 2021

Introduction:

As the culmination of CSE 121, Embedded System Design, the final project was to create a simplified oscilloscope (a TinyScope) using the PSoC 6 microcontroller [3], potentiometers, and a NewHaven display. The features of the TinyScope include:

- Dual-channel oscilloscope with synchronous waveform display
- Allows for input signals ranging from 0 to 3.3 volts relative to the scope's ground
- Sampling rate of ~250 kilosamples per second per channel
- Freerunning mode and trigger mode with configurable trigger level, trigger slope, and trigger channel source
- Adjustable x-scale ranging from 100 microseconds per division to 10000 microseconds per division
- Adjustable y-scale ranging from 500 millivolts per division to 2000 millivolts per division
- Signal frequency calculation and display for both channels
- Two potentiometers for scrolling both waveforms up and down the display
- User interface through a serial communication program

reference [1]

The main objective of the project was to combine the essential concepts of CSE 121 to form a cohesive final project, demonstrating mastery of the topics of CSE 121. The system relies on an understanding of analog to digital converters, DMAs, serial communication through the UART [4], and bare-metal programming. All of these are major parts of the class. The project also provides the opportunity to learn how to build more complex systems as preparation for capstone projects and real-world applications.

Design:

This section will describe the design of the TinyScope. Below is a high level diagram of how the data flows through the system.

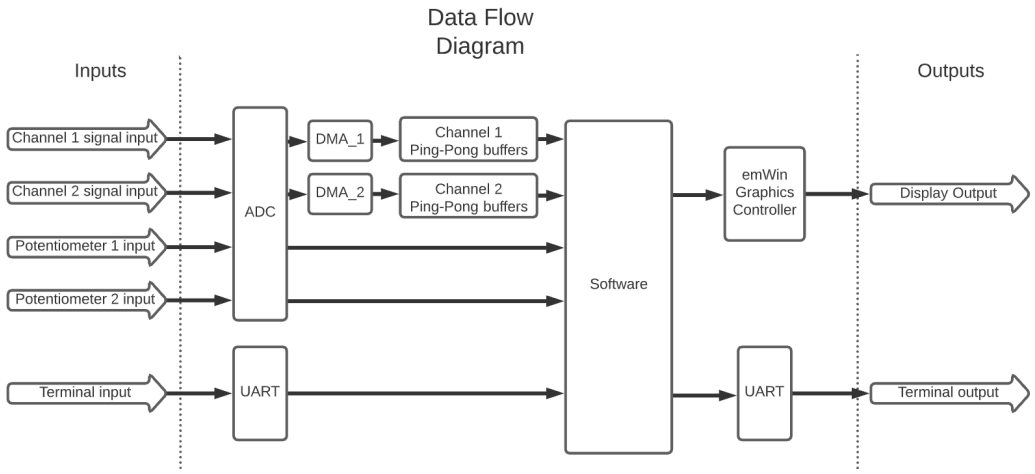

Figure 1: Data Flow diagram

Hardware:

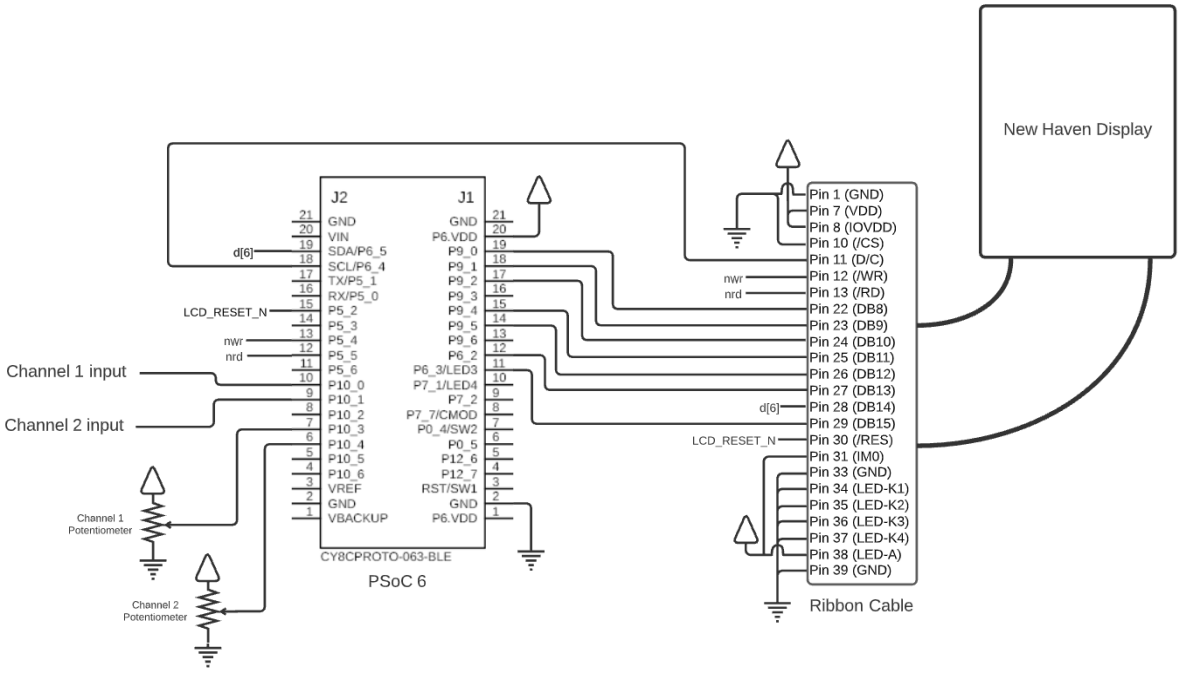

Figure 2: External TinyScope Schematic

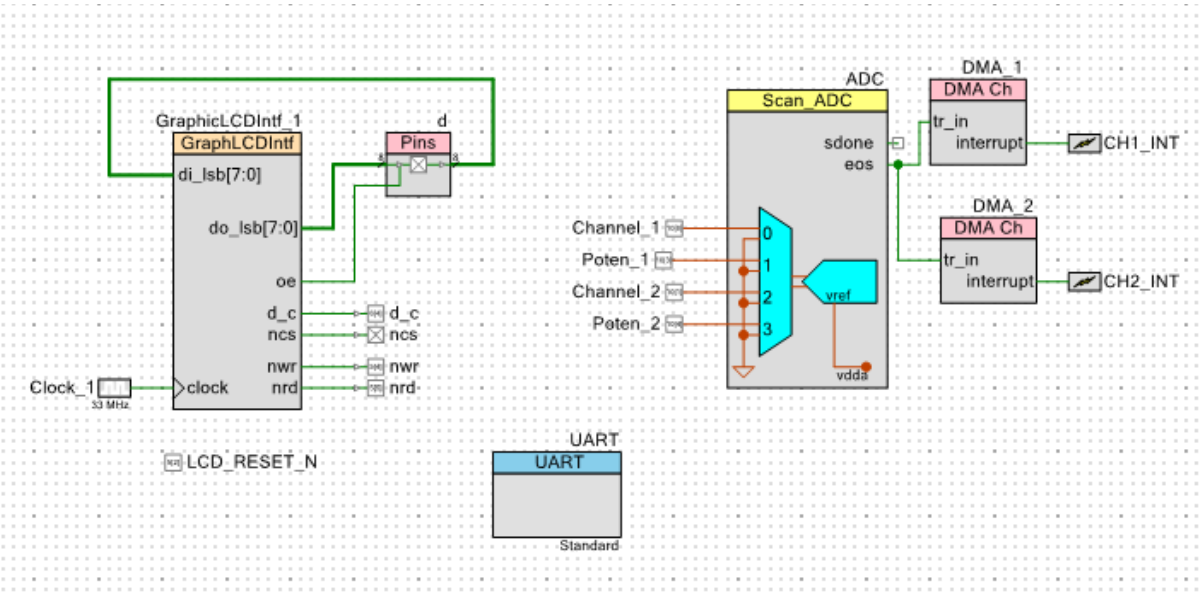

Figure 3: Internal TinyScope Schematic

For the most part, the hardware of the TinyScope is straightforward. The design has five inputs. Two of these are potentiometer readings, which need to be mapped to digital values by an ADC. Similarly, two more of the inputs are the scope's probes which also need their voltage readings converted to digital by the ADC. Lastly, a UART is needed to receive data from the user. The scope also includes two outputs. First, the scope has to be able to send confirmation messages to the user through the same UART and second, it has to be able to display its data through the NewHaven display. I designed the hardware with these specifications in mind.

The bulk of the hardware consists of wiring from the PSoC to the NewHaven display with the use of a ribbon cable. Internally, the signals to control the display are generated by the emWin graphics controller and software library. Next, to allow for the scrolling of the waveform two potentiometers are externally hooked up to ground and VDD. The voltage of the middle terminals of the two potentiometers are then wired to two internal ADC channels. The user can change the voltage reading by turning the potentiometers and the readings are then used for scrolling the waveforms. The oscilloscope's input channels are pins 10[0] and 10[1] of the PSoC. These pins are directly fed to the ADC for sampling the channel voltages. The electrical signal source should share a common ground with the PSoC. To store the data, two DMAs transfer data from the ADC to each channel's corresponding ping-pong buffer everytime the ADC asserts its end of scan (eos) line. To accomplish this, two descriptors, infinitely chained together, continually transfer one element at a time. Lastly, the design uses a standard UART to send and receive data from the user through a serial communication program.

Software:

The software of my project was divided into three files. First, the main file, containing the main function, a function for processing the data of each channel, a function for updating the display, and interrupt service routines. Second, a helper function file including a variety of function definitions for completing tasks such as receiving input from a user, finding the frequency of a signal in an array, copying data between arrays, and more. The third file is a header file including function prototypes, structure definitions, and defines. Each of these files will be discussed independently. I chose to structure my code in this way to keep my code readable, modular, and intuitive for any onlooker.

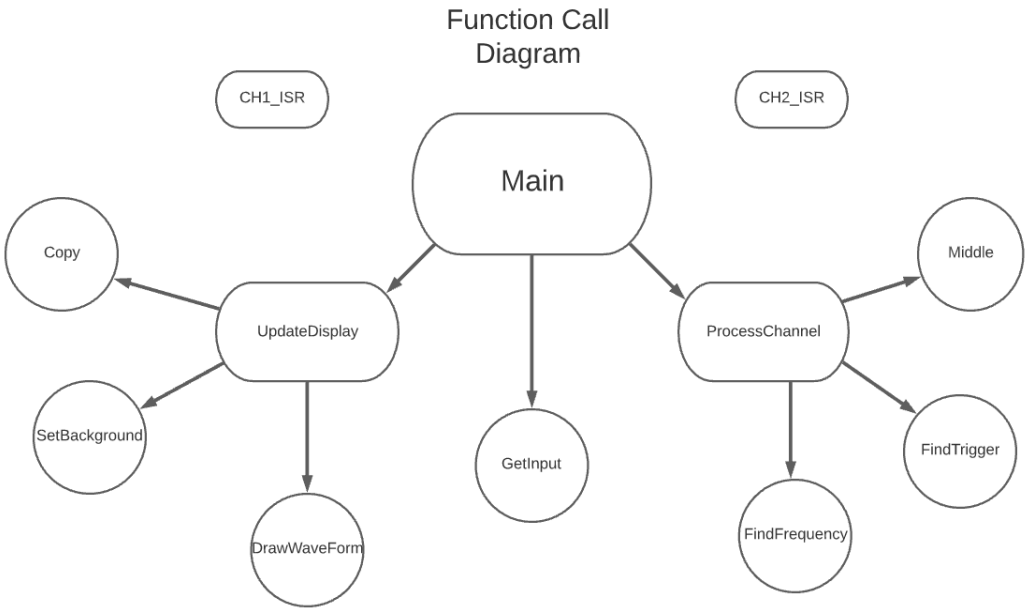

Figure 4: Overarching diagram of which functions call each other. A circle indicates the function is in HelperFuctions.c while an oval represents functions in main_cm4.c. Each function will be described below

HelperFuctions.h:

First, the header file, HelperFuctions.h, is the simplest of the three files. It includes a variety of preprocessor macros, giving useful labels to otherwise hard to interpret numbers. An example of one of these definitions is “PIXELS_PER_X”, a macro for defining the number of pixels in an x-division (32). Next, the file includes a definition of two structures. SCOPE_SETTINGS, a structure keeping track of the

scope's settings includes flags for the run mode, x-scale, y-scale, and more. Next, WAVEFORM_DATA is a structure for keeping track of a waveform's frequency, which ping-pong buffer is being used, the coordinates to draw the waveform, and more. Lastly, the file includes the function prototypes for all of the functions in HelperFunctions.c.

HelperFunctinons.c:

This file contains the many function definitions that are used by the main file. Some are very simple. Each function will be given a brief description and explanation. A diagram of how these functions are called can be seen in figure 4 above.

Copy:

This simple function iterates through an array and copies the data in it into another array. This function is intended to be used for copying the coordinates of a waveform into a separate array so it can be written over (erased from the display) later.

GetInput:

This function will wait for user input from the terminal. Every time it is called it checks for input from the UART and adds any data to a string while filtering out whitespace by using the API function, UART_GetArray [4]. When the function receives an enter ('\n') indicating the end of the input, it compares the received string to each possible command using strncasecmp. When it finds a command that matches, it prints a confirmation message and updates the scope settings depending on the command. If the input does not match any command an error message is printed. Lastly, some commands involve an integer argument. For example, set x-scale has a value to set the x-scale to. My function handles this by using the atoi function to convert part of the string to an integer. As long as the number is within the allowed range, the scope settings are updated to this value.

FindTrigger:

This function is responsible for searching for a trigger in a ping-pong buffer. It loops through a buffer until it finds a point where the data crosses the scope's trigger level in the direction specified by the scope's slope setting. For example, if the scope was set to a 1000 mV trigger level and a positive slope, the function would return the array index at which one element is less than 1000 mV and the following element is more than 1000 mV. To filter out signal noise, the function also checks that elements a noise margin away (noise margin is defined to be 12 elements) also conform to these

specifications. This prevents noise in the signal from causing false positives. Lastly, each data point is checked for underflow from the ADC since this can cause a spike in the data and a false positive trigger.

Middle:

This simple function looks for an approximate middle value of a ping-pong buffer. Using a for-loop, it iterates through the buffer constantly updating the highest value it found and the lowest value it found. When it returns it looks at the difference between the two. If the difference between them is less than the NOISE_THRESHOLD macro (~100) the difference is too close and a 0 is returned indicating there is no significant variation in the signal. Otherwise, the function returns the average of the maximum and minimum values, added to the minimum value. For example, if the max was 1000 and the minimum was 500, this would return 750.

FindFrequency:

The FindFrequency function is responsible for iterating through a channel's ping-pong buffer and calculating the frequency of the data it holds. As arguments, this function intakes one of the buffer arrays and the expected middle value (calculated by the Middle function). The function iterates through the data until it finds a point where the data crosses the middle. The index of this point is stored and the direction of this crossing is remembered (either a negative or positive crossing). The loop then resumes until another crossing in the same direction is found. The frequency is then calculated by dividing the sampling rate (in kilosamples per second) by the difference between these two indexes. This returns the frequency of the signal in hertz

SetBackground:

This function is responsible for drawing the background of the TinyScope display before waveforms are drawn. It iterates 9 times to draw 9 vertical divisions to create 10, 32 pixel x-divisions. It then iterates 7 times to draw 7 horizontal lines to create 8, 30 pixel y-divisions. Lastly, it writes the value of each channel's frequency (passed in as arguments) in the top left corner and the scope's scaling setting (again passed in as arguments) in the top right corner.

DrawWaveForm:

DrawWaveForm is a simple function that intakes coordinates (in the form of two arrays, one holding X coordinates and one holding Y coordinates) and uses them to

draw the waveform they specify. This function loops 320 times (the number of x-axis pixels on the screen) and draws a line between each of the coordinates until it reaches the end of the coordinate arrays. In this way, the function is able to generate drawings of the waveforms on the display.

main_cm4.c:

The main_cm4.c file contains the main function as well as the major task functions of the TinyScope. Within this file, instances of the previously discussed structures defined in HelperFunctions.h are instantiated with default values. Also in the file are 2 ping-pong buffers per oscilloscope channel and flags to be set to true or false which indicate when an ISR has been entered.

CH1_ISR / CH2_ISR:

These very simple ISRs are called every time a DMA is finished writing data into one of the channel's ping-pong buffers (indicated by a descriptor finishing). The ISR then clears the interrupt, updates which buffer is safe to read from, and raises a flag to indicate a transfer is finished.

UpdateDisplay:

UpdateDisplay is a short function responsible for updating the NewHaven display with new data. The function starts by erasing the previous waveforms by drawing over the same waveform from the previous call in the background color. The background is next set with a call to SetBackground and the new waveform is drawn to the screen. An offset to the waveform is added by adding a scaled-down reading of the potentiometer voltage to the coordinate array. This allows the user to scroll the waveforms up and down by turning the potentiometer. Lastly, the data of the waveform is copied to a previous waveform coordinate array so the function can draw over the waveform the next time it is called.

ProcessChannel:

By far the most complex function within the project is ProcessChannel, responsible for extracting data from the channel ping-pong buffers including the frequency and coordinate array. The function is called every time the channel 1 array raises its flag indicating it finished writing to its ping-pong buffer. A flowchart diagram of how the function works can be found below (figure 5).

First, within the function, the iteration count (a static integer) variable is checked. It keeps track of timing in the function and what should be done in each call. At the end of every function call, it is incremented by one, giving the function a new task to complete the next time it is called. If the iteration count is equal to READY_TO_START (arbitrarily defined as 35) the ReadyToDraw flag is set to false. This is because the function is getting ready to calculate new frequencies and waveforms next time it is called and if the display was refreshed at this time, it could cause breaks in the waveform. Next, when the iteration count equals FIND_MIDDLE, the function will calculate the middle value for one of each of the channel's ping-pong buffers with a call to the Middle helper function. Which buffer is read from is decided by checking which buffer is not currently being written to as kept track of in the WAVE structure. During the next call to the function, when the iteration count equals FIND_FREQ (defined as 38), the ping-pong buffers of each channel not being written to are passed with the calculated middle value from the last iteration to the FindFrequency function. As long as the FindFrequency function does not return an error, each channel's frequency is updated. If an error is found, the frequency is not updated and the previous frequency is kept.

The next part of the ProcessChannel function is the creation of the coordinate arrays for drawing each channel's waveforms. This starts when the iteration count reaches FORMAT_DATA (41). At this point, the function looks for where to start in the buffer. If the scope is in freeRun mode, the index starts at zero, meaning we begin construction of the coordinate array at the start of each channel's ping-pong buffer. Otherwise, if the scope is set to trigger mode, the channel that is set as the source has its buffer passed to the FindTrigger function. If this function is unable to find a trigger point in the data, it returns an error and the ProcessChannel function ends without incrementing the iteration count so that ProcessChannel can try to find a Trigger point next time it is called. Otherwise, the starting index is set to the point found by the FindTrigger function.

Finally, these data points beginning at the index are added to the coordinate arrays from the ping-pong buffers for each channel. The value of each data point is scaled down in accordance with the scope's y-scale value and the data is checked for underflow. Then, the index is increased a variable amount based on the x-scale setting. If at any point the index becomes greater than the size of the array, an array size is subtracted from the index and the function is ended. The next time the function is called the creation of the coordinate array will pick up where it left off since the variables are static. This process continues until there is a coordinate for each point on the x-axis of the screen (320 pixels long). Once this is done, the iteration count is set to zero and the ReadyToDraw flag is set to True, causing the function to start over next time it is called.

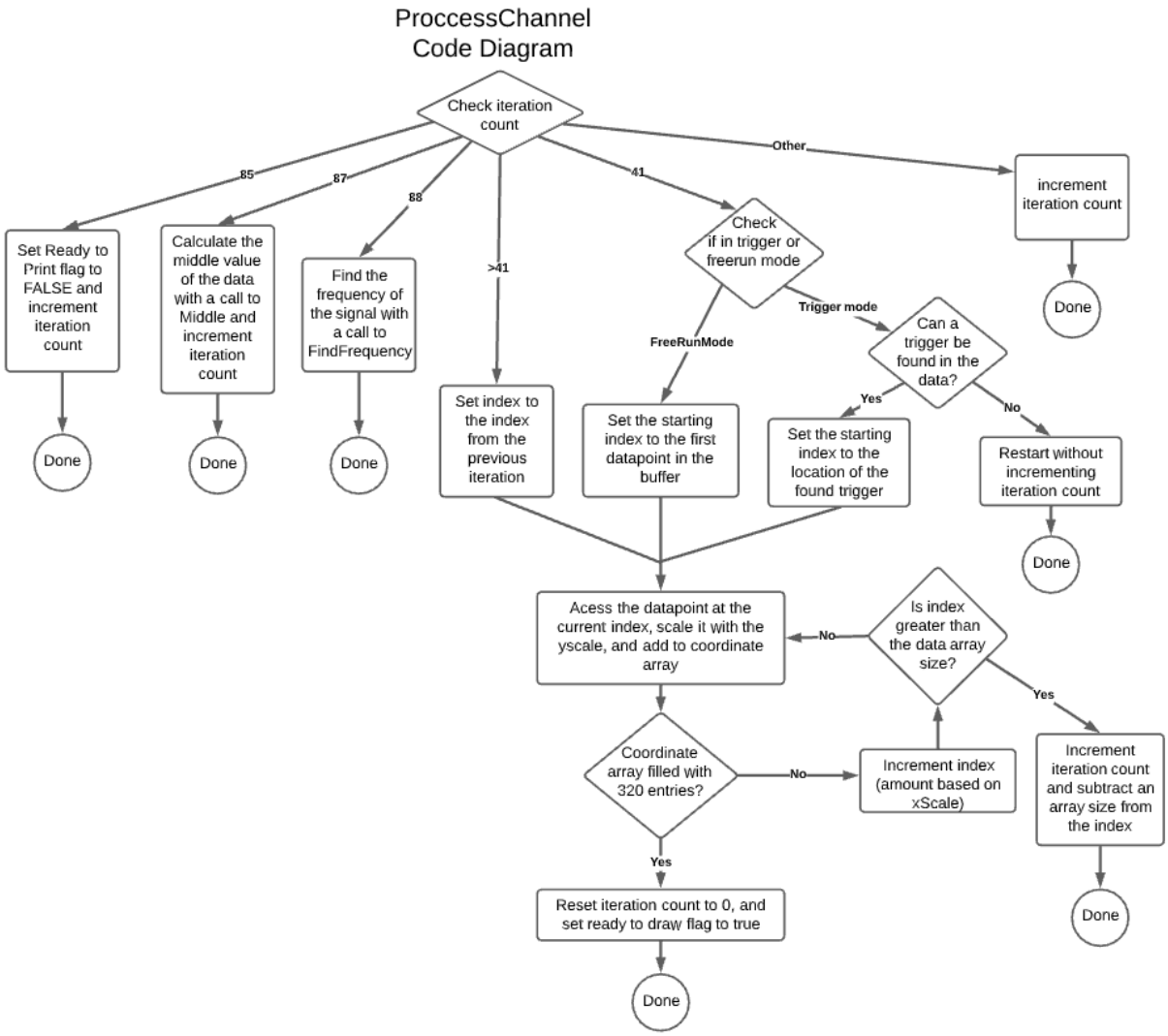

Figure 5: ProcessChannel flow diagram

main:

Lastly, the main function is fairly simple. It first initializes the UART and waits in an infinite loop until the user enters the start command, constantly checking with the GetInput function. Once the scope has been started, the main function breaks out of the loop and starts the initialization of the interrupts, hardware, and DMA. It also starts the ADC scanning, DMA transfer, and sets the display's background. Next, the function enters the main infinite loop. Here, user commands are constantly received with the GetInput function. Whenever the channel 1 ISR flag is raised, the ProcessChannel function is called to deal with the new data and the flag is lowered. Lastly, the UpdateDisplay function is called and the ReadyToDraw flag is lowered whenever the scope's ReadyToDraw flag is raised. The UpdateDisplay function is also periodically

called if the scope is stopped to allow the user to scroll the waveforms even in the stopped state. A diagram of the main function is shown below.

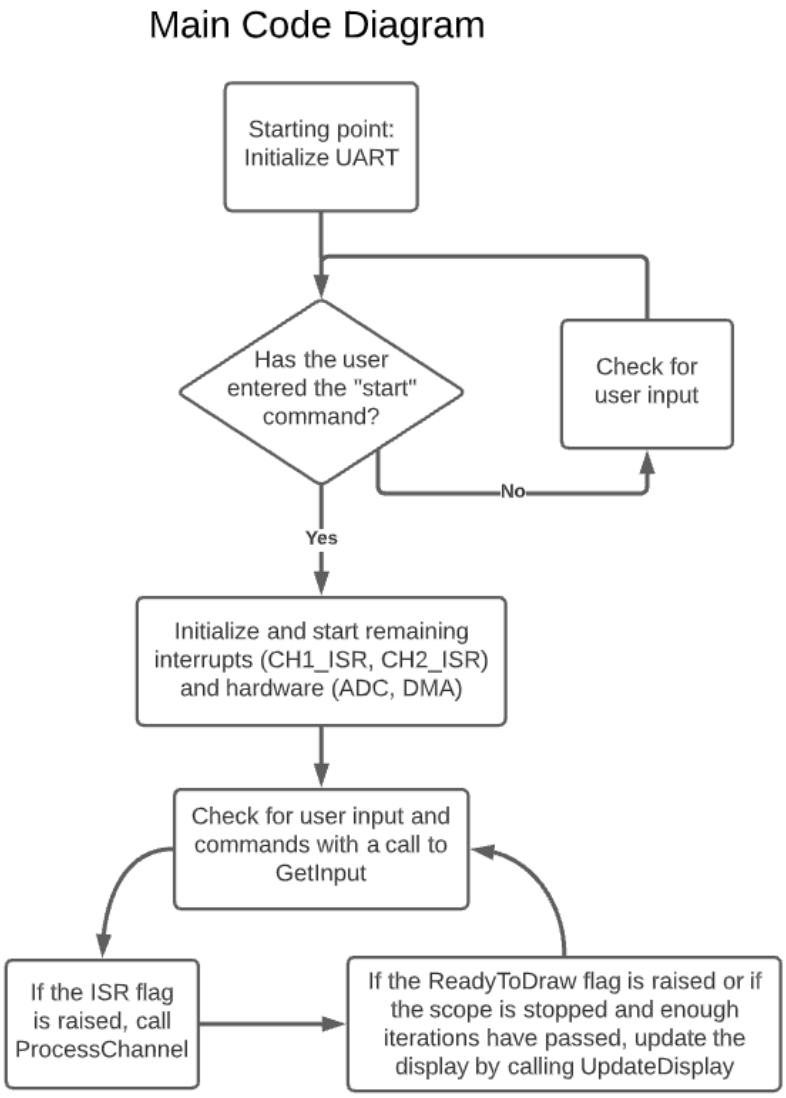

Figure 6: main code diagram

Testing and Design Process:

I took my time in the design process, slowly building up the tiny scope incrementally. I chose to start my design process by first building the function to get input from the user through the terminal (the GetInput function). By searching through the PSoC 6 API I found a suitable function to use, UART_GetArray [4], for getting user input and storing it to a string until the user pressed enter. Then, I confirmed the simple

design worked by calling the function in an infinite loop and having it echo each command it received back to the terminal. In testing this, I found it to be successful.

From there, I created my hardware and set up the wiring from the PSoC to the display and added the potentiometer and ADC. I tested to make sure my wiring was correct by running the provided sample code [2] to print to the display. Once I had this working, I built my own function to set up the background of my display including the divisions, scales, and frequency results. I also added in the ping-pong buffers DMAs and ISRs to allow for data transfer from the ADC to the buffers. To test these worked, I went into debug mode and manually viewed the ping-pong buffers to make sure they had reasonable values.

With the basics of the project done, I began to code the bulk of the project, displaying and interpreting the waveform data from the ADCs. I began by simply plotting the data in one of the buffers occasionally to make sure I understood how to display graphics and to ensure the ADC was working properly. I then built some of my helper functions to find the frequency of the signals and find a trigger point as well as y-scaling to adjust the millivolts per division of the scope. Lastly, I created the x-scaling using the technique described in the ProcessChannel function description. This was the most intimidating part of the scope so saved it for last and got everything else working first.

A major issue I encountered was my two scope channels being asynchronous. Just when I thought I had finished my design, I reread the lab manual [1] to make sure everything was up to standard. I realized that each waveform was not being sampled at the same time as the project specifications required. This was because originally I had two ProcessChannel functions for each channel. Thankfully, with some restructuring of my code, I managed to fix this issue by only keeping one ProcessChannel function and including the code to sample both waveforms within it. In this way, I could guarantee that each waveform was sampled at the same time as the other. Beyond this, I did not have any major issues with the project. Most of my troubles came from figuring out how to break down the project into smaller parts and to get the courage to start working.

Finally, for testing my complete design I used the Analog Discovery 2 oscilloscope and waveform generator. I set the AD2 to generate waveforms on two different channels and hooked the waveform channels up to both the TinyScope and the AD2's oscilloscope. Then, by comparing the results on my scope to the results on the AD2, I was able to confirm that the TinyScope worked correctly. I continued by altering the waveform frequencies, shapes, and the x/y scales and trigger modes of both the AD2 and the TinyScope to make sure everything was functioning correctly. I also made sure the calculated frequency agreed with the time it took to complete a cycle. For

example, if the x-scale was set to 1000 microseconds, and a wave cycle completed in 1 x-division, we would expect the frequency to be about 1 kHz. Lastly, I tested my design by continually changing the modes back and forth and made sure errors did not occur. My design was able to pass all of these tests.

Results:

Below is a picture of the oscilloscope and the final results of my project. My design meets all specifications and is able to display waveforms, calculate the frequencies of waveforms, go to a trigger mode with a variable slope and channel source, and intake commands from a serial terminal. I also exceeded design specifications by building the scope so that it can be set to any integer value for the trigger level, y-scale, and x-scale within the allowed range. For example, my scope is able to be set to an x-scale of 3241 (this was arbitrarily chosen as an example) microseconds per x-division as well as the standard required values (ex: 100 us per division, 5000 us, 10000 us). I also found my scope to be stable. I do not observe any breaks or glitches in the waveforms, I do not find unexpected trigger behavior, nor do I find incorrect frequency calculations. For these reasons, I am confident that the scope is fully functional and meets the requirements.

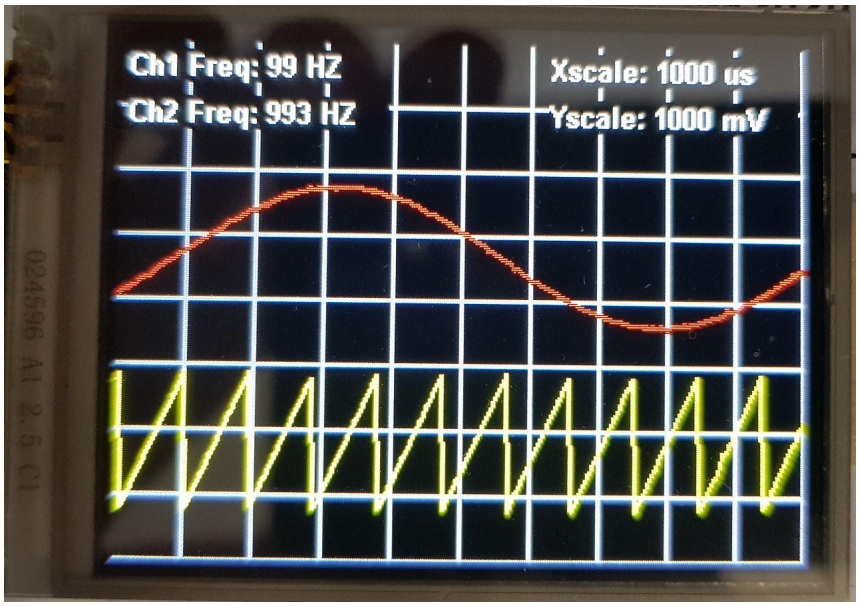

Figure 7: Image of a sample waveform on the TinyScope

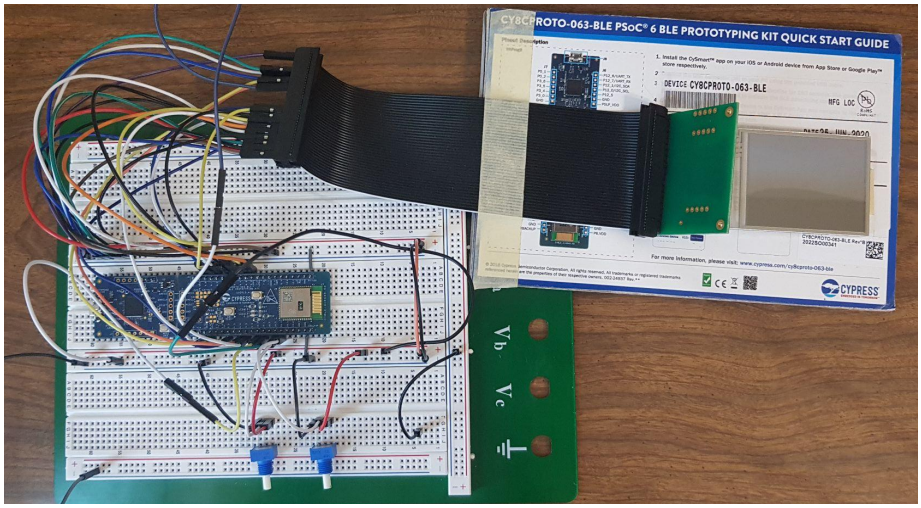

Figure 8: Image of the completed TinyScope

Conclusion:

The final project of CSE 121 combines the many concepts from the class together to create a simple oscilloscope. The scope uses an ADC to convert an analog electrical signal to a digital representation capable of being displayed on a NewHaven display. It uses DMAs to transfer data from peripherals to the memory of the CPU. It uses a UART and FIFOs to read commands from users and return confirmation messages. The scope also includes various settings allowing it to change the scale of the x-axis in microseconds, the scale of the y-axis in millivolts, and much more.

The tiny scope shows what embedded systems and microcontrollers are capable of with the use of their built-in peripherals. Microcontrollers are able to use their peripherals to interact with and measure the environment in ways microprocessors on their own are not capable of. This was demonstrated in the Lab-Project with the PSoC 6's ability to use it's UDB and ADC peripheral to measure electric signals. Microcontrollers also have the advantage of being small compared to full-scale computers while still being able to process data. Lastly, microcontrollers such as the PSoC 6 are less expensive and more power-efficient than microcontrollers. All these reasons make microcontrollers such as the PSoC 6 the perfect choice for embedded systems and the TinyScope from this lab.

There are many possible ways to improve the TinyScope. The first, most obvious limitation is the relatively low sampling rate preventing accurate measurement of signals about ~ 10 kHz. Unfortunately, this limitation comes from the limited PSoC 6 ADC. However, a higher sampling rate could be obtained by reducing the number of

potentiometers, and therefore ADC channels needed. If the NewHaven display's touch screen function were to be used for scrolling waveforms up and down, this would mean only two ADC channels would be needed, consequently doubling the sampling rate of each remaining channel and allowing for higher frequency measurements. Another possible improvement would be to optimize and simplify code. While I did my best to keep the code as simple and concise as possible, it is inevitable that it could be improved in places. One way to improve the design would be to search through the code and find optimizations. Lastly, a third channel could theoretically be added to the scope by adding in another ADC channel, more ping-pong buffers, and duplicating code to process the channel. However, there are also challenges with this idea since adding another ADC channel will decrease all ADC other channel's sampling rates. Furthermore, the screen could become crowded when displaying three different waveforms.

In the end, this lab took me about a week and a half of consistent work (2-3 hours per day) to complete. It was the most difficult lab so far, but also the most fulfilling. When I first read about the lab I was very intimidated, but thanks to what I learned in CSE 121, I found the project to be surprisingly intuitive and very doable if you put in the work. My greatest takeaway from this project was I should be sure to carefully read the directions before getting started. Having created an asynchronous design at first (as described in the testing section) caused me a lot of frustration and unnecessary stress. Luckily I managed to solve the problem and I am now left with a great final product.

References:

[1] Anujan Varma. 2021. Lab-project.pdf.
https://canvas.ucsc.edu/courses/42390/files/folder/Lab%20Project?preview=3970355

[2] Anujan Varma. 2021. Final_project_demo.cyprj.Archive01.
https://canvas.ucsc.edu/courses/42390/files/folder/Lab%20Project?preview=3729292

[3] Cypress. 2018. CY8CPROTO-063-BLE PSoC 6 BLE Prototyping Board Guide.
https://www.cypress.com/file/450941/download

[4] Cypress. 2018. Cypress Peripheral Driver Library - UART (SCB)
file:///C:/Program%20Files%20(x86)/Cypress/PDL/3.1.4/doc/pdl_api_reference_manual/html/group__group__scb__uart.html