Justin Fortner

ID# 1481947

jfortner@ucsc.edu

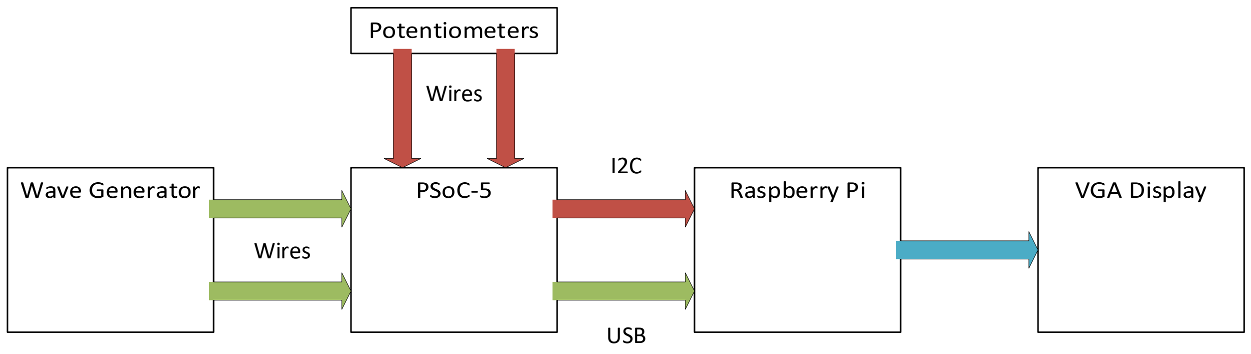
Lab Thu 5-7PM

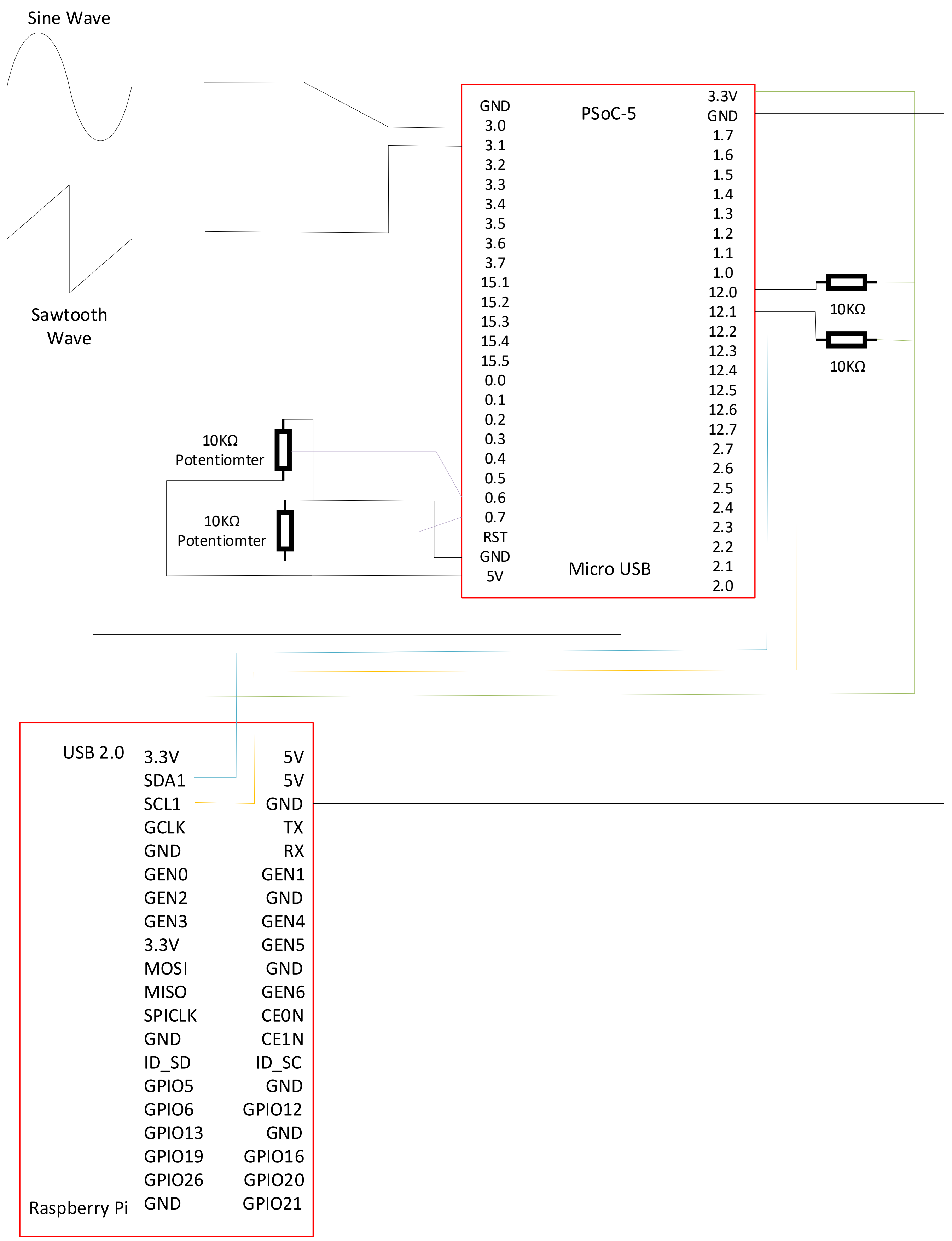
Final Project Lab Report

This is the final project for CMPE121/L, Microprocessor System Design. The challenge is to create a fully functioning and feature rich oscilloscope and logic analyzer. A PSoC-5, Raspberry Pi and various computer peripherals will allow for this all to happen. This lab combined many of the skills learned throughout the first 5 lab exercises as well as new skills that needed to be learned for this lab. Thus, both hardware and software methodology were inspired by those past labs as well as examples given to the class by Professor Varma.

Part 1: Oscilloscope

This oscilloscope has six main features. It can display two separately controlled waveforms. These waveforms can run in either free or trigger mode. If trigger mode is selected the trigger can be configured to either channel as well as a positive or negative trigger. The scaling of these waves can happen in both the x and y direction independently. These waves can also be moved up and down on the display independently with the use of two potentiometers (one for each wave).




Sine Wave
Sawtooth Wave
PSoC-5
GND
3.0
3.1
3.2
3.3
3.4
3.5
3.6
3.7
15.1
15.2
15.3
15.4
15.5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
RST
GND
5V
Micro USB
3.3V
GND
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
12.0
12.1
12.2
12.3
12.4
12.5
12.6
12.7
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2.0
10KΩ
10KΩ
10KΩ Potentiomter
10KΩ Potentiomter
USB 2.0
Raspberry Pi
3.3V
SDA1
SCL1
GCLK
GND
GEN0
GEN2
GEN3
3.3V
MOSI
MISO
SPICLK
GND
ID_SD
GPIO5
GPIO6
GPIO13
GPIO19
GPIO26
GND
5V
5V
GND
TX
RX
GEN1
GND
GEN4
GEN5
GND
GEN6
CE0N
CE1N
ID_SC
GND
GPIO12
GND
GPIO16
GPIO20
GPIO21

The features of the oscilloscope are as follows. The oscilloscope is designed to be configurable in order to display a variety of different waveforms. The oscilloscope is configured before the program is run using various commands in the command line. The x and y scale are controlled by the commands -x and -y respectively. By default, both scales are set to 1000 (1ms/ division for the x scale and 1V/division for the y scale). The y scale can have a value of 100, 500, 1000, 2000 and 2500, while the x scale can have values of 100, 500, 1000, 2000, 5000, 10000, 50000 and 100000. The mode of the oscilloscope is controlled by the command -m. By default, the oscilloscope is set to free run mode. In this mode the oscilloscope will display the wave naturally as it enters the program. In order to achieve a more stable waveform the oscilloscope can be set to trigger mode. When in trigger mode more command options become available to control the waveform. The first of which being the control over the trigger slope using -s. The slope is set to positive (pos), forcing the wave to start on a rising edge, by default, but can be set to negative (neg), forcing the wave to start on a falling edge. The next command that opens when in trigger mode is -t. This command allows a trigger level to be set. The trigger level default is 2500 (2.5V) but can be set to anywhere between 0 (0V) and 5000 (5V) in increments of 100 (.1V). The trigger level tells the oscilloscope what value the waveform needs to cross, either positive or negative depending on the trigger slop setting, in order to trigger the waveform drawing. The final command, -c, informs the oscilloscope which waveform the drawing will trigger on. By default, the waveform will trigger on channel 1, but can also be set to channel 2. When the oscilloscope is running with the parameters that have been configured or left to the default value two waveforms will appear on the screen. These waveforms can be changed while the program is running through the waveform generator that is supplying them. The waveforms can also be adjusted up and down on the screen using two potentiometers, one for each waveform. In addition to the waveforms, the oscilloscope will also display a reference grid, scales, and trigger parameters.

The system layout involves various components outlined in the block diagram above. A PSoC receives two waves from a wave generator. These waves are converted from analog to digital and then passed to the Raspberry Pi through a bulk USB transfer. The PSoC-5 also receives the input from two potentiometers. Again, this data is converted from analog do digital. However, this data is relayed to the Raspberry Pi through I2C. The Raspberry Pi allows for the user to configure various setting of the oscilloscope. Once those setting are adjusted, and the program is run, the Raspberry Pi converts the data to coordinates, based on the setting given before the program is ran, to be displayed on the VGA screen. Data is constantly being loaded from the waveform generator through the PSoC-5 to the Raspberry Pi, thus the Raspberry Pi is also constantly resetting the coordinates for the VGA display and redrawing the finalized waveform.

The hardware design was vital for this project. A PSoC-5 will be used to receive two channels of analog information from the Analog Discovery 2 that is located at each lab station and convert that information to digital data. This is done through two separate SAR ADCs. The digital data from the ADCs is stored within two separate ping pong buffers (four totals between both waves) with the use of two DMAs (one for each wave). The DMAs are set to use two TDs in a loop in order to take full advantage of the ping pong buffers. The data within one ping pong buffer will be sent to the Raspberry Pi, through USB bulk transfer, while the other is filling up. The roles of these buffers are controlled by an interrupt generated at the end of each 64 byte DMA transfer. The Raspberry Pi will then analyze that data and display it to a monitor. Data is also sent to the Raspberry Pi from the PSoC-5 through I2C. The I2C transfer is responsible for transmitting the data given by the potentiometers. This data also needs to travel through an ADC. However, unlike the waves, two potentiometers are converted with the use of one Sigma Delta ADC. This is accomplished by only passing the data from one potentiometer to the Sigma Delta ADC at a time. This control is provided using one 2-1 MUX. I2C is also allows for the Raspberry Pi to send commands to the PSoC-5. The Raspberry Pi can stop the sampled data to be passed from the PSoC-5 to the Raspberry Pi. In order to ensure the PSoC-5 met and did not exceed the power requirements of the Raspberry Pi during the I2C transfers the rail connecting the PSoC-5 to the Raspberry Pi needed to be isolated from the 5V power flowing into the Raspberry Pi from the USB. In order to accomplish this the R15 resistor was removed from the PSoC-5 board. The 3.3V power from the Raspberry Pi was then connected to the VDD input on that isolated rail. The ground on the Raspberry Pi was connected to the ground on this rail as well. In addition to this, the I2C pins from the PSoC-5 to the Raspberry Pi were set to and configured to be resistive pull up. Thus, the PSoC-5 would not exceed the 3.3V power limit of the Raspberry Pi. The exact layout of this can be seen in the hardware schematic.

The goal for the software in the oscilloscope program was to write fast, simple, well formatted and commented code. I accomplished fast code by ensuring only necessary code was implemented and running. This includes breaking the program into as small of pieces as possible so that no excess commands are ran. The code that was implemented and running was made as simple and linear as possible. This allowed all time constraints to be met and for the frequencies of the various components to be set to the maximum values outlined in the lab manual. The code was similarly formatted across all devices and commented extensively in order to ensure ease of understanding. The final goal of the code was to ensure that errors were checked in all the appropriate places. I did my best to account for as many as I could think of.

The oscilloscope has a simple communication protocol. Waveform data is passed into a 64 byte array within the PSoC. Once this array is full is it passed to the Raspberry Pi using a 64 byte USB bulk transfer. Because there are two waveforms, and two ping pong buffers two different endpoints are used when transferring the data between the two devices. The ping pong buffers are set up so that while one array is filling up the other is transferring. This I handled within the interrupt of the DMA. In order to combat the two endpoints mixing data only one USB bulk transfer takes place at a time. This is also handled within the interrupt of a DMA. The data being received by the raspberry Pi is stored in a 64 byte array. Each waveform has their own 64 byte array. This data is then dealt with by the Raspberry Pi to display the correct waveform on the VGA display.

For I2C data is taken from a Sigma Delta ADC and stored in one byte of a two byte array. The MUX channel is then switched to the next potentiometer where the other byte of data is stored in the next byte of the two byte array. This data is then transmitted through the I2C to the raspberry pi. This data is stored in two separate variables and used to move the waveforms up and down individually.

For the PSoC the program flows as follows. All variables are initialized, and all components are set up and started. The main for loop is then entered. If a command was received through I2C it stores that value in a buffer and clears the status. The program then checks if this value was set to 2. If it was then no data will be transferred to the Raspberry Pi and it will start at the top of the for loop. Otherwise it continues through the program as normal. If it is the first USB bulk transfers turn to send data then it will wait for the buffer to be empty, check which ping pong buffer to send and complete the bulk transfer. If it is the second USB bulk transfers turn it follows the same steps but with the other two ping pong buffers. Once the transfers are finished the PSoC enters the I2C protocol. The MUX channel is set to 1. The potentiometer value is then read and passes through boundary checking. The MUX channel is then changed to two. A small delay occurs to give the mux time to change channels. The potentiometer value is then read and passes through boundary checking. These values are placed in a buffer and sent over I2C to the Raspberry Pi. At some point the DMA will generate an interrupt and switch the roles of the two buffers.

For the Raspberry Pi the program flows as follows. All libraries and structs are initialized. Then modular portions of the code are initialized and set up. Then the main loop occurs. Variables are set up and initialized to their default value. Command arguments are then dealt with. The terminal is then saved, and user input is enabled. I2C is then set up followed by USB. The main while loop is entered. The VGA screen is started. A value is written to the PSoC over I2C. The potentiometer data is read. The data is received from the USB bulk transfers and stored in a larger data array. If the mode is set to free, then process the waves and dials them in free mode. Otherwise we are in trigger mode. The code then displays checks what channel and what trigger slope is detected. The correct set of wave data is then processed and displayed on the VGA screen. Various counters are reset, and the screen is ended. The while loop is then reran.

Errors were dealt with whenever necessary. This included invalid command arguments, invalid bulk transfers, and invalid I2C or USB setups.

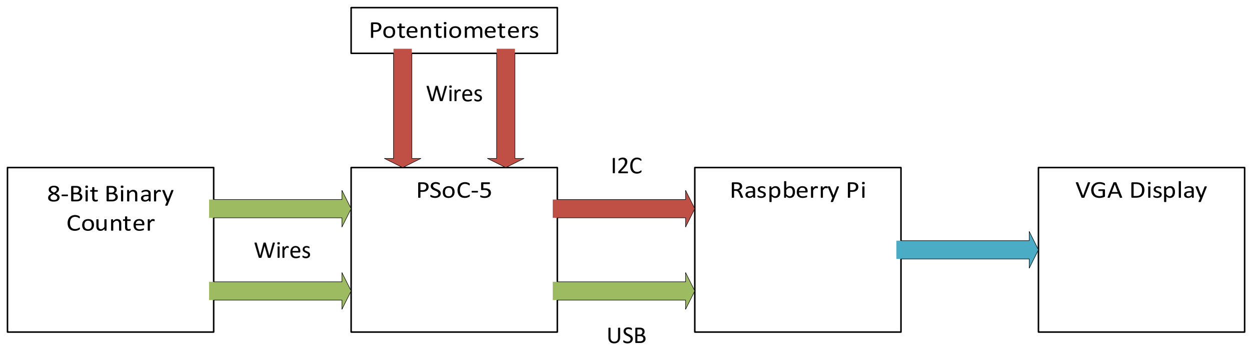
I tested this design in as small of chunks as I could in order to ensure that each change, I made along the way was current. After each implementation of each feature I would test all previously implemented features as well to ensure that I had not accidentally messed one of those up. Once the design was competed in its entirety I tested as many end cases as I could think of. This brought some issues t my attention. In order to solve these, I once again would change and test small chunks at a time. Retesting old portions as well. Once I believed to fix all the issues, I tested edge cases again. If I found more issues, I would repeat that process.

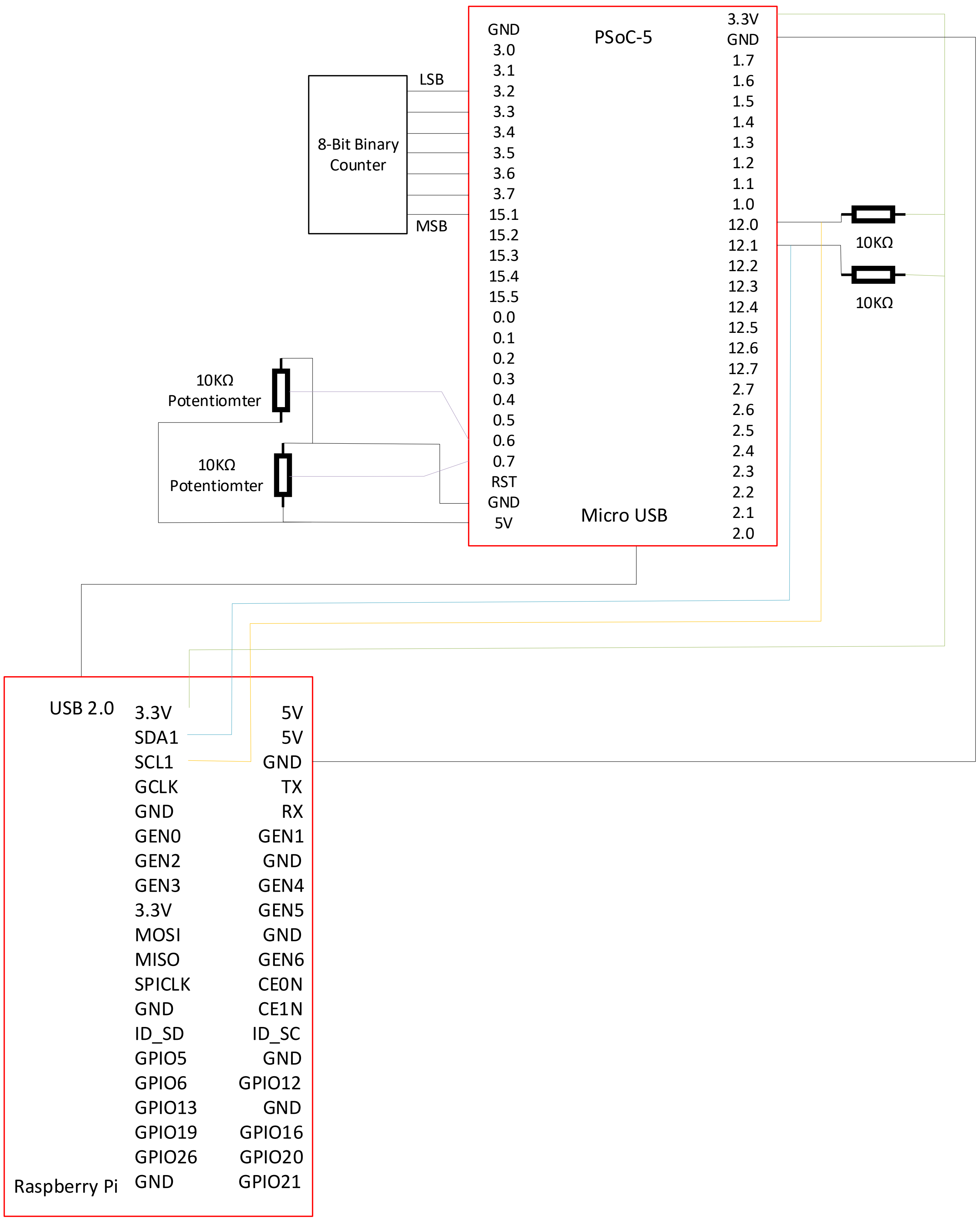
I did not encounter any specific difficulties with this portion of the lab. I was able to use incremental testing to quickly debug any problems that I had very quickly. This led for a quick implementation of the oscilloscope. I wish I could have said the same for the logic analyzer.

The biggest way in which I would like to improve this design is to allow it to be configurable while the program is running. I find it very slow and inconvenient to have to exit the program in order to reconfigure the oscilloscope setting. Especially because the waveform can be adjusted while the program is running.

Part 2: Logic Analyzer

This logic analyzer simultaneously samples either eight, four or two different channels. The data for these channels is an eight bit binary counter. Each channel is responsible for the displaying of one bit in the channel. The logic analyzer also scans a file that contains one logic expressions. This logic expression is used to decide when the waveform will begin its drawing on the screen. The drawing can either happen when the logic expression goes from false to true, positive, or true to false, negative. In addition to triggering the waveform the amount of data the waveform displays on the window is also configurable. As is the sample frequency and x scale. The analyzer will also have a cursor. This cursor can be moved left and right using one of the two potentiometers in the system. The other potentiometer moves all the waves left or right.




PSoC-5
GND
3.0
3.1
3.2
3.3
3.4
3.5
3.6
3.7
15.1
15.2
15.3
15.4
15.5
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
RST
GND
5V
Micro USB
3.3V
GND
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1.0
12.0
12.1
12.2
12.3
12.4
12.5
12.6
12.7
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2.0
8-Bit Binary Counter
LSB
MSB
10KΩ Potentiomter
10KΩ Potentiomter
10KΩ
10KΩ
USB 2.0
Raspberry Pi
3.3V
SDA1
SCL1
GCLK
GND
GEN0
GEN2
GEN3
3.3V
MOSI
MISO
SPICLK
GND
ID_SD
GPIO5
GPIO6
GPIO13
GPIO19
GPIO26
GND
5V
5V
GND
TX
RX
GEN1
GND
GEN4
GEN5
GND
GEN6
CE0N
CE1N
ID_SC
GND
GPIO12
GND
GPIO16
GPIO20
GPIO21

The features of the logic analyzer are as follows. This logic analyzer is designed to be configurable in order to analyze a variety of different logic commands. The first example of this configurability can be seen through the channel controls. By default, the logic analyzer captures and displays data from 8 channels, labeled A through H, simultaneously. However, using the command -n in the command line the number of channels can be set to 4 or 2 as well. In addition to the number of channels the system displays, the program also allows for the memory depth of the waves to be configured using the command -m. By default, this value is 5000, however, the depth can range anywhere from 5000 to 50000 in increments of 5000. This configurability allows for the user to determine hoe much data they want to be taken and displayed at once. The rate at which this data is sampled can be configured using the command -f. The four frequency options provided by this program are 1, 10, 50, and 100 kHz, with the default being 10 kHz. The last command that can configure the wave is -x. This command allows for the x scale to be changed from its default value of 1000 (1ms/ division) to either 10, 100, 500, 1000, 2000, 5000 or 50000. The trigger for the logic analyzer is also configurable. The command -t allows for a logic expression to be read in from an external file. This expression determines the trigger point for the wave. The direction in which the wave will trigger, positive for false to true or negative from true to false is set using the -d command. The logic analyzer will also have a cursor. This cursor can be moved left and right using one of the two potentiometers in the system. The other potentiometer moves all the waves left or right. In addition to this, the logic analyzer will also display a reference grid, scales, and trigger parameters.

The system layout involves various components outlined in the block diagram above. A PSoC receives the data from an 8-bit binary counter, one wire is responsible for each bit on the counter. These bits are stored in a status register and stored within two ping pong buffers using a DMA. The data is passed to the Raspberry Pi through a USB bulk transfer. The PSoC-5 also receives the input from two potentiometers. This data is converted from analog to digital. However, this data is relayed to the Raspberry Pi through I2C. The Raspberry Pi allows for the user to configure various setting of the oscilloscope. Once those setting are adjusted, and the program is run, the Raspberry Pi converts the data to coordinates, based on the setting given before the program is ran, to be displayed on the VGA screen. Data is constantly being loaded from the counter through the PSoC-5 to the Raspberry Pi, thus the Raspberry Pi is also constantly resetting the coordinates for the VGA display and redrawing the finalized waveform.

The hardware design is integral in the success of the logic analyzer. A PSoC 5 will be used to receive eight channels of digital information from the Analog Discovery 2 that is located at each lab station and convert that information to digital data. This data is coming from an eight bit binary counter within the Analog discovery 2. The data from these channels is stored in an eight bit status register. One DMA then moves this data from the control register to a set of ping pong buffers. The DMA is set to use two TDs in a loop in order to take full advantage of the ping pong buffers. The data within one ping pong buffer will be sent to the Raspberry Pi, through USB bulk transfer, while the other is filling up. The roles of these buffers are controlled by an interrupt generated at the end of each 64 byte DMA transfer. The Raspberry Pi will then analyze that data and display it to a monitor. Data is also sent to the Raspberry Pi from the PSoC-5 through I2C. The I2C transfer is responsible for transmitting the data given by the potentiometers. Unlike the counter data, the potentiometer data must travel through a Sigma Delta ADC. The two potentiometers are converted with the use of one Sigma Delta ADC, just as they were in the oscilloscope portion of this project. This is accomplished by only passing the data from one potentiometer to the Sigma Delta ADC at a time. This control is provided using one 2-1 MUX. I2C is also allows for the Raspberry Pi to send commands to the PSoC-5. The Raspberry Pi select the frequency at which the 8 bit binary counter is sampled. This is accomplished using a 4 to 1 MUX. The sampling frequency is set at the very beginning of the program. Once this frequency is set the PSoC will continually send data to the Raspberry Pi. In order to ensure the PSoC-5 met and did not exceed the power requirements of the Raspberry Pi during the I2C transfers the rail connecting the PSoC-5 to the Raspberry Pi needed to be isolated from the 5V power flowing into the Raspberry Pi from the USB. In order to accomplish this the R15 resistor was removed from the PSoC-5 board. The 3.3V power from the Raspberry Pi was then connected to the VDD input on that isolated rail. The ground on the Raspberry Pi was connected to the ground on this rail as well. In addition to this, the I2C pins from the PSoC-5 to the Raspberry Pi were set to and configured to be resistive pull up. Thus, the PSoC-5 would not exceed the 3.3V power limit of the Raspberry Pi. The exact layout of this can be seen in the hardware schematic.

The logic analyzer has the same basic software goals as the oscilloscope. The software in this program was to write fast, simple, well formatted and commented code. I accomplished fast code by ensuring only necessary code was implemented and running. This includes breaking the program into as small of pieces as possible so that no excess commands are ran. The code that was implemented and running was made as simple and linear as possible. This allowed all time constraints to be met and for the frequencies of the various components to be set to the maximum values outlined in the lab manual. The code was similarly formatted across all devices and commented extensively in order to ensure ease of understanding. The final goal of the code was to ensure that errors were checked in all the appropriate places. I did my best to account for as many as I could think of.

The logic analyzer has an even simpler communication protocol than the oscilloscope despite having four times as many channels. 8 bit counter data is passed into a8 bit status register. A DMA then transfers this data to a 64 byte ping pong buffer. Once this array is full is it passed to the Raspberry Pi using a 64 byte USB bulk transfer. The ping pong buffer is set up so that while one array is filling up the other is transferring. This I handled within the interrupt of the DMA. The data being received by the raspberry Pi is stored in a 64 byte array. This array is then transferred to a larger data array that will be used to display the waves for each channel.

For I2C data is taken from a Sigma Delta ADC and stored in one byte of a two byte array. The MUX channel is then switched to the next potentiometer where the other byte of data is stored in the next byte of the two byte array. This data is then transmitted through the I2C to the raspberry pi. This data is stored in two separate variables and used to move the waveforms up and down individually.

For the PSoC the program flows as follows. All variables are initialized, and all components are set up and started. The main for loop is then entered. If a command was received through I2C it stores that value in a buffer and clears the status. Depending on the value that was given to the PSoC a different MUX channel will be selected. This channel controls the sampling rate. Next the PSoC takes care of the USB Bulk transfer. The program will wait for the buffer to be empty, check which ping pong buffer to send and complete the bulk transfer. Once the transfer is finished the PSoC enters the I2C protocol. The MUX channel is set to 1. The potentiometer value is then read and passes through boundary checking. The MUX channel is then changed to two. A small delay occurs to give the mux time to change channels. The potentiometer value is then read and passes through boundary checking. These values are placed in a buffer and sent over I2C to the Raspberry Pi. At some point the DMA will generate an interrupt and switch the roles of the two buffers.

For the Raspberry Pi the program flows as follows. All libraries and structs are initialized. Then modular portions of the code are initialized and set up. Then the main loop occurs. Variables are set up and initialized to their default value. Command arguments are then dealt with. The terminal is then saved, and user input is enabled. I2C is then set up followed by USB. The main while loop is entered. The VGA screen is started. A value is written to the PSoC over I2C. The potentiometer data is read. The data is received from the USB bulk transfers and stored in a larger data array. Data is then bitwise anded with the appropriate value and stored in eight separate arrays. Thus, allowing for the 8 different bits of data to be displayed on eight separate channels. The program then checks how many channels should be displayed. It processes and prints out only the necessary amount in order to be efficient. The cursor is then printed as well as the target indicator. Various counter variables are reset to zero and the sreen transmission is ended. The loop is then reran.

Error checking was done when deemed appropriate. This included invalid command arguments, invalid bulk transfers, invalid I2C or USB setups, invalid file names and invalid logic equations.

My testing method for the logic analyzer was the same for the oscilloscope. However, here it is again. I tested this design in as small of chunks as I could in order to ensure that each change, I made along the way was current. After each implementation of each feature I would test all previously implemented features as well to ensure that I had not accidentally messed one of those up. Once the design was competed in its entirety I tested as many end cases as I could think of. This brought some issues to my attention. In order to solve these, I once again would change and test small chunks at a time. Retesting old portions as well. Once I believed to fix all the issues, I tested edge cases again. If I found more issues, I would repeat that process.

Just as in the oscilloscope I did not have any issues with this lab. Until I tried to implement the data being controlled by the logic expression to generate a trigger. I could not, and still have not figured out how to implement this logic. I have tried many different iterations of the program and have made no progress. I have tried to step through the program, test portions of the algorithm separately, and even creating a separate program just to handle this portion of the project, but I wasn't able to come up with a working solution before my turn in date. These iterations cannot be seen in the current submission because it would qualify as "dead code". This is something I had been marked down on in past lab submissions and didn't want to make the same mistake twice. Now that school is over, I plan on diving back into this project in order to implement this algorithm.

First and foremost, I would like to improve my design by finishing the implementation of my current design. However, once that is finished there are three big design changes I'd like to implement. I would like for the program to be able to handle multiple logic expressions and while the program is running be able to switch between the different expressions. After this I would like for the display to show the number that is being shown by the waves at the trigger point. I believe this could be a valuable piece of information. Lastly, I would like for the design to be configurable while the program is running. I would like this for the same reason sated when evaluating the oscilloscope, I find it very slow and inconvenient to have to exit the program in order to reconfigure the logic analyzer setting. Especially because the data can be adjusted while the program is running.

All in all, this project was a fantastic mix of both old and new material. This allowed for a challenging yet informative lab. I also found this lab to be extremely practical and interesting which I appreciated. I learned some valuable lessons, but the most important was the value of incremental testing. Incremental testing streamlined my development process and saved me a great deal of time. It wasn't for all these things the 60+ hours spent on this lab would not have been worth it. I plan on revisiting this project on my own in order to implement the improvements I discussed throughout the lab report. With that being said, I am happy with the results I was able to produce for the project. I was able to complete the oscilloscope perfectly and almost complete the implementation of logic analyzer.

References:

As stated in the introduction some code was inspired by professor Varma's examples. Those examples can be seen on the canvas website here: https://canvas.ucsc.edu/courses/18218/files. Specific files used for this project include:

usb_bulk_transfer.c, usb_discover.c, I2ctest.c, scopedemo.c, lab-project.pdf, Configuring-I2C.pdf, openVG-Install.pdf, Getting-started-with-PSoC.pdf