Nelson Yeap
Lab Project
12/10/17

Lab Project Report

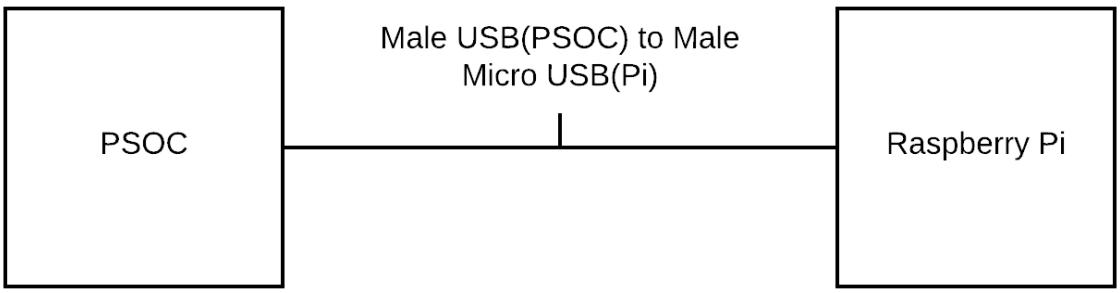
Introduction

This lab is a project which builds on the concepts we have learned this semester to build an oscilloscope and a logic analyzer out of the PSOC and the Raspberry Pi provided to us in the lab kit. In this project we will be using the PSOC to take in analog signals, convert it to digital data, send it to the Raspberry PI, and then have the Raspberry Pi output that data onto the screen using the openVg library. That will be what is needed for the oscilloscope, but the logic analyzer is slightly different. For the logic analyzer we will need the PSOC to take in data from the Raspberry PI, analyze the data coming in from the input signal, gather the data needed for the output, send it to the Pi, and then have the Pi output that data using the OpenVg library.

Methods

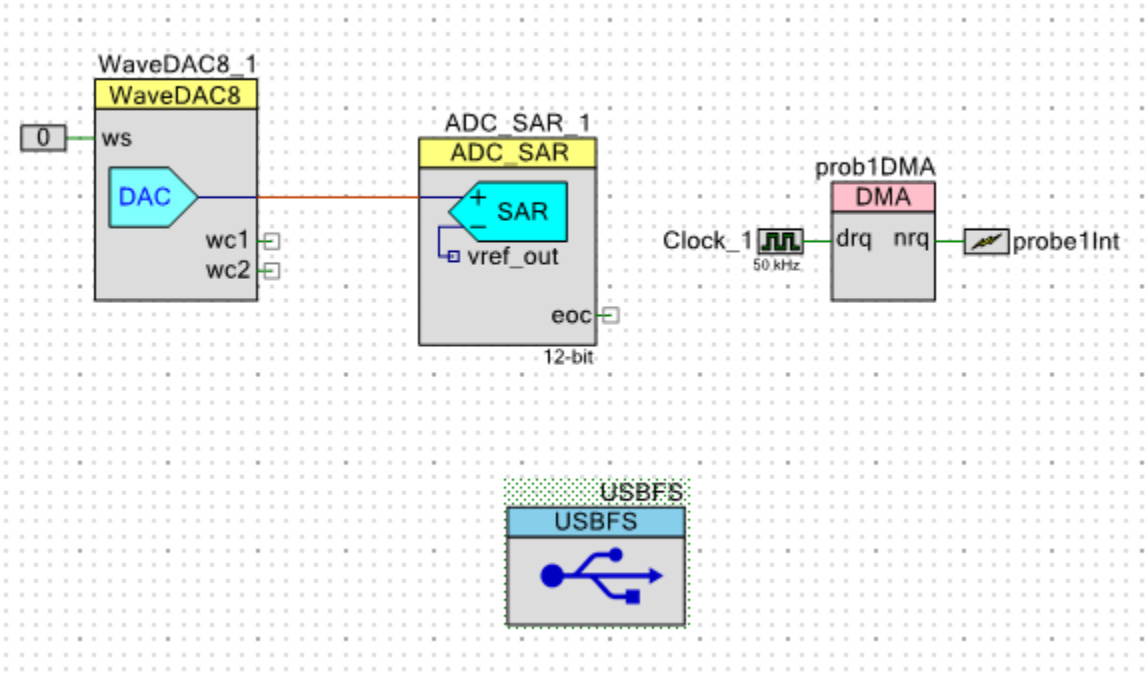
Oscilloscope

For this part of the lab we seek to make an oscilloscope out of the PSOC and the Raspberry Pi. The PSOC is supposed to take in two analog signals, convert it to digital, and then send it to the Pi for screen outputs. The schematic below shows the link between the PSOC and the Pi and shows what cables are needed for them to communicate.



To start with we will work on the PSOC side of the oscilloscope. Rather than go right to producing two waves, I instead chose to work with one wave, get all the settings and then work on getting the second wave to work. So, for the purpose of working on one wave, the PSOC will need a waveDAC, a SAR ADC, a DMA, and a USBFS block. For the PSOC, we are using the waveDAC as a test signal. The PSOC will read the waveDAC's analog signal and then treat it like any signal that an oscilloscope could read. The SAR ADC, is a faster ADC that we will be using to convert the signal from an analog signal to a digital signal. There are a couple of reasons we are using a SAR ADC. First the SAR ADC is a lot faster than the ADC DelSig block. The Successive Approximation Register ADC is an ADC that find its output through a binary search function. The normal ADC Delsig finds it data through a stepping function and is not as fast as the SAR ADC. Another reason we are using the SAR ADC is that the PSOC only has one ADC DelSig available

in its hardware, but has two SAR ADCs at its disposal. So, for these reasons we are going to use a SAR ADC to convert the analog signal to digital with the PSOC. After we convert the signal to digital we are going to put into SRAM with the DMA. The DMA will use ping-pong buffers to transfer that memory so that we are able to both send and receive the oscilloscopes data at the same time. What this means is that we will have two memory arrays for a single signal on the PSCO side. While memory array 1 is filling up with memory the PSOC will send the data in memory array 2 and when memory array 2 is being filled with data the PSOC will be sending the data in memory array 1. This allows the PSOC to be as efficient and quick as possible and will allows us a higher limit in terms of speed when sending data to the Pi. The top design for the setup on the PSCO for a single channel's worth of data will look as follows.



What we see on the left is the waveDAC with is output being fed into the SAR ADC. The waveDAC is our test signal which will output a sin wave, and the SAR ADC will see these analog values and then convert them into digital values. Then the DMA to the right of the SAR ADC will transfer theses digital values from the SAR ADC to memory arrays that we create in the PSOC flash memory through its interrupt. The DMA interrupts will also serve to send the digital values to the Pi through the USB connection. As mentioned above in the ping-pong buffer. This will allow the PSOC to read and send memory to the same time.

Now that the PSCO side of a single wave is complete we can now move on to setting up the Raspberry Pi to be able to draw on the screen using the openVg library. So, the Raspberry Pi needs to install the openVg library by downloading the git repository and then installing the files. After openVg is finished installing we can then move on to writing the code for the oscilloscope.

For this lab I first chose to setup the arguments on the Pi. This can be accomplished by using the getopt function and looping through the arguments and having switch cases statements for each of the arguments. This way, any argument that is inputted will have their values changed accordingly in the program, but any arguments that are left out will be set to their default values. The getopt function

For the oscilloscope the Raspberry Pi is responsible for receiving the digital data from the PSOC and then outputting it to the screen. So, to begin with, we will try to get the background and grid on the screen. We can draw a background on the screen by calling the Background() function from the openVG library. The background function takes in three arguments, Red, Green, Blue. It will then draw the background the color that is input from the RGB color scale. Then we need to draw the grid for the oscilloscope. To do this I created a separate function that simply calls the Line function many times across the screen. The Line function takes in four arguments, X1, Y1, X2, Y2 and then draws a line between those two points. And now with this we have a background and a grid. Now what we need to do is receive the data from the PSOC and draw it on the screen.

To being with we need to receive the data from the PSOC. To do this we first need to figure out how much data we need to take in from the PSOC. What I decided to do was to take in the maximum number of samples needed from the PSOC which ends up being (sampleSec/10 * xDivisions) *xScale. This means that for any xScale I take in enough data such that it the xScale is its maximum number, I still have enough samples to output onto the screen. So now that I know how many samples to take in, I need to actually receive them through the USB link. To do this we simply call the libusb_bulk_transfer() function and specify the array to receive in. And now that we have the data we need to print it out. This can be done with a for loop and calling the Line function in the for loop. The loop will loop through all the data and map one data point to one pixel on the screen. So, with this simply loop we have it will draw width amount of samples on the screen where width is the width in pixels of the screen. But since our loop loops through all the data points, it will end up drawing data off the screen. So, to prevent this from happening and making our oscilloscope even slower, we will create an if stamen that breaks out of the loop when we reach the width of the screen. But with this, we should have a wave on the screen and we can move on to the other parts of the lab.

Now that we have a wave on the screen we can now move on to the xScale and the yScale portions of the oscilloscope. To do this we have to understand what we need to do. The xScale needs to scale up or down the amount of waves per division to reflect the user input. The oscilloscope shows voltage over time and the xScale input determines the time per division. An xScale of 1000 means 1000 µs/division while and xScale of 100 means µs/division. What we need to do now is to figure out how many samples are needed on the screen based on the user's input of xScale. With our current set up we have enough samples to fit on the screen at the maximum xScale, but we are only able to draw at an xScale of 1000. We need to find out how to draw samples on the screen based on the user input. This can be by doing an analysis through dimensional analysis. To analyze this correctly we need to figure out what dimensions we are beginning with and trying to end with. In this case, we are trying to map samples to pixels. So, we begin with samples and figure out how to get it to the pixels unit. This can be done as follows.

$$sample * \frac{seconds}{sample} * \frac{divisions}{seconds} * \frac{pixels}{divisions}$$

What we have done is converted samples to seconds, seconds, to divisions, and divisions to pixels. This should give us the samples in terms of pixels. We can also get each of these values by observing our setup on the PSOC side and user inputs. We can observe seconds/sample by looking at our samples/seconds or our DMA frequency. By taking the frequency and dividing 1/frequency, we can get the value for seconds/sample. For divisions/second, we can observe that the xScale input is in

nanosecond/division. By dividing 1/xScale we can get the value for divisions/seconds. And finally, pixels /divisions is a simple math equation of width/xDivisions where width is the width of the screen in pixels. So, what we can now do is write a function in which we do the following equation for each sample.

$$xResult = sample * \frac{1}{sampling\ frequency} * \frac{1}{\frac{xScale}{100000}} * \frac{width}{xDivisions}$$

This equation maps each sample to a pixel on the screen depending on what xScale is. Then when we draw lines between each of these samples, the lines will be drawn closer together or farther away resulting in more/less samples on the screen depending on the xScale input. Note that in the third term xScale must be divided by 100000 because the xScale is in terms of nanoseconds. To do the dimensional analysis correctly, it must first be converted to seconds. Now to use this, you simply call the function that we do this math in on the x coordinates of the line functions that are called when drawing the wave. If this is done correctly, xScale should be implemented and working.

With xScale done, we can move on to the yScale. yScale will do something very similar in that we will do dimensional analysis in a separate function to map our values to different pixels on the screen. For yScale the equation will look as follows.

$$dValue * \frac{yDivisions}{dValue} * \frac{pixels}{yDivisions}$$

This equation converts the digital value(dValue) into divisions and then into pixels. This equation does something very similar to xScale in that it will map a certain voltage to a certain pixel height. The function we write would have to depend on yScale and the height of the monitor and would do the following calculation

$$yResult = dValue * \frac{1}{\frac{yScale}{18.5}} * \frac{height}{yDivisions}$$

What we see in the second term is that yScale must be divided by 18.5 because that is the value in pixels/volts that is needed to get the wave to the correct height in terms of pixels. We can then use this function in the same way we use the xScale function, we simply call it on the Y coordinates in the Line function used to draw the waves.

With the xScale and the yScale complete we can now move on to the trigger level and the trigger slope. These two properties must be done at about the same time as they are related in triggering the wave at the correct place. The trigger level is a value specified by the user in which the wave should begin drawing at. If the user inputs 1000. Then that means the wave should begin to be drawn when its voltage first reaches 1V. However, in a single period of a sing wave, there are two points at which it reaches 1V, which is why we must do the trigger level and the trigger slope at the same time. We will begin with explaining how to find the trigger level on the Raspberry Pi. What I chose to do here was a ratio function. I looped through my samples and found that my maximum digital value fort a 4V sin wave came out to be 430. This means that the digital value 430 corresponds to 4V from the analog signal. With this information I can then do a ratio to figure out what the value for any given trigger level should be.

$$\frac{430}{4V} = \frac{x}{\frac{triggerLevel}{1000}}$$

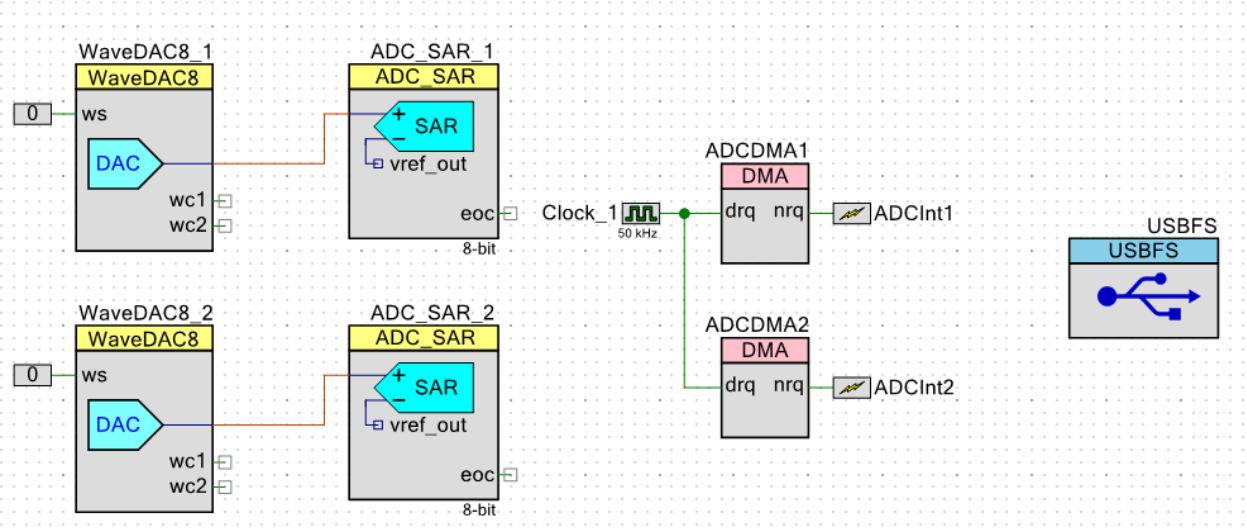
The equation above is a ratio in which the numerator is a digital value and the denominator is an analog voltage value. The triggerLevel variable must be divided by 1000 to correctly match the units of voltage on the left side of the question. But what we can observe is how the equation will work. We want to solve for x which means that isolating x would give us the following equation.

$$x = \frac{\frac{430}{4V}}{\frac{triggerLevel}{1000}}$$

And so, what we see is that our digital value is multiplied by the value of 4/triggerLevel/1000 or 4V divided by our desired triggerLevel. This means that x is equal to 430 multiplied by the ratio of 4/triggerLevel/1000 which should be a digital Value corresponding to our desired trigger voltage. Now that we have a digital value for out trigger level we can then just set a flag to only begin drawing the line when we find this trigger value. However, if we do this we observe a problem that will appear. The trigger level can happen at any time so what we observe is that the wave will begin to draw from some random point on the screen. What we want is for the wave to be drawn at the trigger point from the start of the screen, so we have to find a way to fix the is problem. One way to fix it is to simply iterate through values until we find this trigger value, and for every time we don't find it, we decrement where we will begin to print on the screen. So, it we find the trigger value at the 1000th value what we want to do is have decremented where we will begin drawing by 1000 times. This means that we must have decremented the value that goes into the xTransform function by the number of samples that have passed until the trigger value is found. With a flag this is not too difficult. I create a variable called "slope" and set it to 0 to begin with. And when I find the trigger value I change it to 2. But every time I do not find the trigger value I decrement the counter used to decide where to begin drawing on the screen. This should make the wave begin printing at the trigger value all the way to the left side of the screen. Now for the slope, we all we need to do is add if statements that control when the trigger value is found and the slope it is found at. What we have done is simply add an if statement that checks three things: $sample[i-1] < triggerValue$, $sample[i] > triggerValue$, and $\frac{sample[i]-sample[i-1]}{(i-(i-1))} > 0$. The first two conditions check for the trigger value and the last condition checks to make sure that in that instance the slope is positive. This should allow the wave to trigger at the correct voltage on the positive slope. For the negative slope we simply do the same if statement but with reversed conditions: $sample[i-1] > triggerValue$, $sample[i] < triggerValue$, and $\frac{sample[i]-sample[i-1]}{(i-(i-1))} < 0$. This should allow for the trigger level and trigger slope to respond to the user input as intended.

After this all we need is to set up free mode on the oscilloscope. Free mode is when the oscilloscope simply displays the data it receives. It can/will start displaying all data that comes in without regards to the trigger level or slope. Free mode can be done simply by taking in the data and then printing it out onto the screen with the xScale and yScale modifications. With free mode there is no need to check for any kind of trigger conditions. We simply print out the information the Pi receives from the PSOC.

With all of these done and tested, we should have a single wave that has all the arguments implemented and tested. Now we can move on to getting a second wave on the screen. To do this we have to go back to the PSOC and modify our code and design a bit. The PSOC side is relatively simple in that to get a second wave on the screen all we need to do is add another WaveDAC, Sar ADC, and DMA. These components do the exact same thing as the first ones, and we simply set another endpoint on the USB so that we send our waves data on independent endpoints. One thing to look out for is that having a second wave means that we are trying to send twice as much data. The PSOC can send a single waves worth of data at about 100kHz maximum. This means that to get two waves, we need to take this value and half it. So, at its maximum, we can only send our data at 50kHz. Below is a schematic of our top design with two waves on the PSOC.

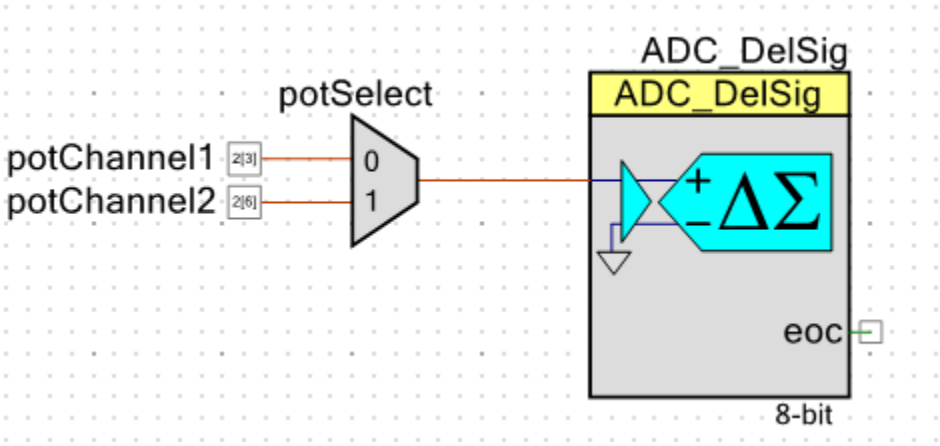


What we see is on the left we have two SAR ADC which are translating our two waves into digital data. Then our two DMAs will transfer that data into memory array in the main code, and finally we have the USB which will send both waves of data to the Raspberry PI to output onto the monitor. Note that the DMAs run at 50kHz because that is about the maximum speed they can run at with our current setup.

Now on the Pi side, for this second wave, all we need to do is print it out at the same data indices as our triggered wave. To do this you can simply create another variable that saves the index in which the triggered wave begins drawing and then begin drawing your second wave from that index. But we are not done with that just yet. From here we must set up the trigger channel argument. This allows the user to pick which channel of the oscilloscope to trigger on. With two waves, we can either trigger on our first wave or our second wave. With the implementation described above we can trigger on the first wave, but not the second wave. To trigger on the second wave, all we need to do is copy the code from the first channels trigger and reverse the order in which we do things. What this means is that we take all our trigger/slope checks and apply it to the second waves data instead of the first waves data. Then when we find where to trigger the second wave, we save the index here it triggered and then begin printing our first wave from that same index. And with this all we have left is the potentiometers.

The potentiometers will be able to move our waves vertically on the screen. This means that we will be able to control the y positions of our waves to make them overlap or separate from each other.

To do this we need to add more blocks to our PSOC side. Below is a schematic of the hardware blocks used to implement the potentiometer on the PSOC side.



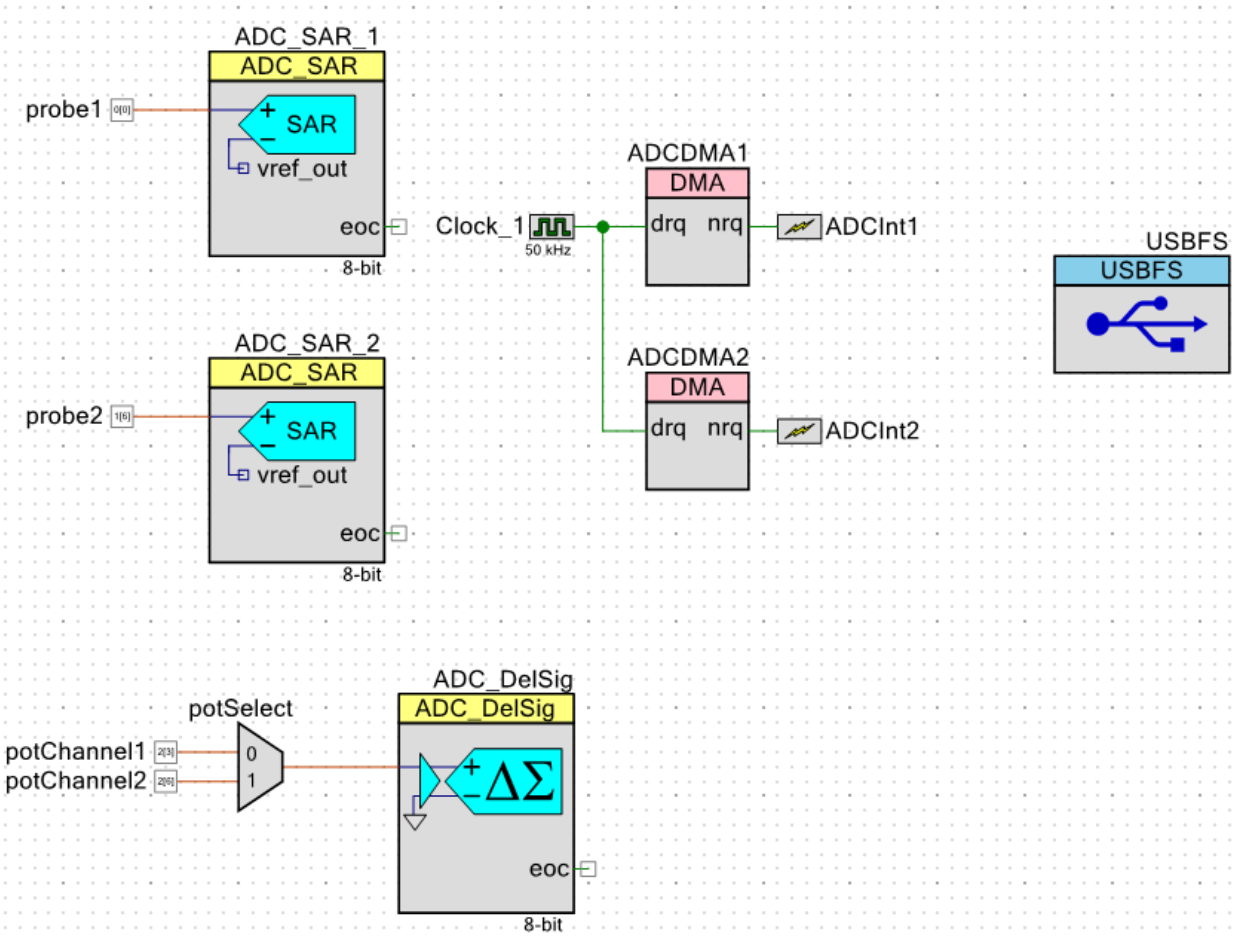
What we see is our two analog pins corresponding to our two potentiometers going into an analog multiplexer and then into an ADC DelSig. The reason we are using only one ADC DelSig is simple: the PSOC only has one ADC DelSig in its hardware. So, we our only choice is to setup our potentiometers in such a way that they can share the ADC DelSig. This is done through the analog mux. With this setup we can read out potentiometers and then send the data to the Raspberry Pi. For the purposes of this lab, I set up the potentiometer values to be sent through the DMA interrupts where the data is sent. Normally, it should not be done this way, but sending the data through a time interrupt messed up the data that was being sent from the sin waves. Thus, my only choice was to send it through the DMAs where there was no chance of interfering as I set up my transfers to be blocking transfers.

On the Pi side all we need to do is receive the potentiometer data and then scale it to the screens height. This can be done with the following equation.

$$scaledPot = pot * \frac{height}{255}$$

This equation will take any value we have for the potentiometer and then scaled to the height of the screen. This will allow our pots to move both waves the full height of the screen. To actually see this all we must do is add the scaled potentiometer values to the Y coordinates of the line function we used to draw. This should add the pots values and allow us to control them on the screen.

With all of this done and tested rigorously, we have finally completed the oscilloscope. Below is a schematic of the complete PSOC setup used for the oscilloscope.



Something to notice is the absence of the waveDACs in the first schematic. This is because the waveDACs were simply acting as test signals to make sure that the implementation of our oscilloscope worked. To truly be sure it works, we have to hook it up to a signal generator and feed that analog signal into the PSOC. Then by running he PSOC and the Raspberry Pi at the same time, we can observe the wave generated by the signal generator and be sure that the oscilloscope that we created is working as intended. Once this is done, we are finished with the oscilloscope and can move on to the next part of the project, the logic analyzer.

Logic Analyzer

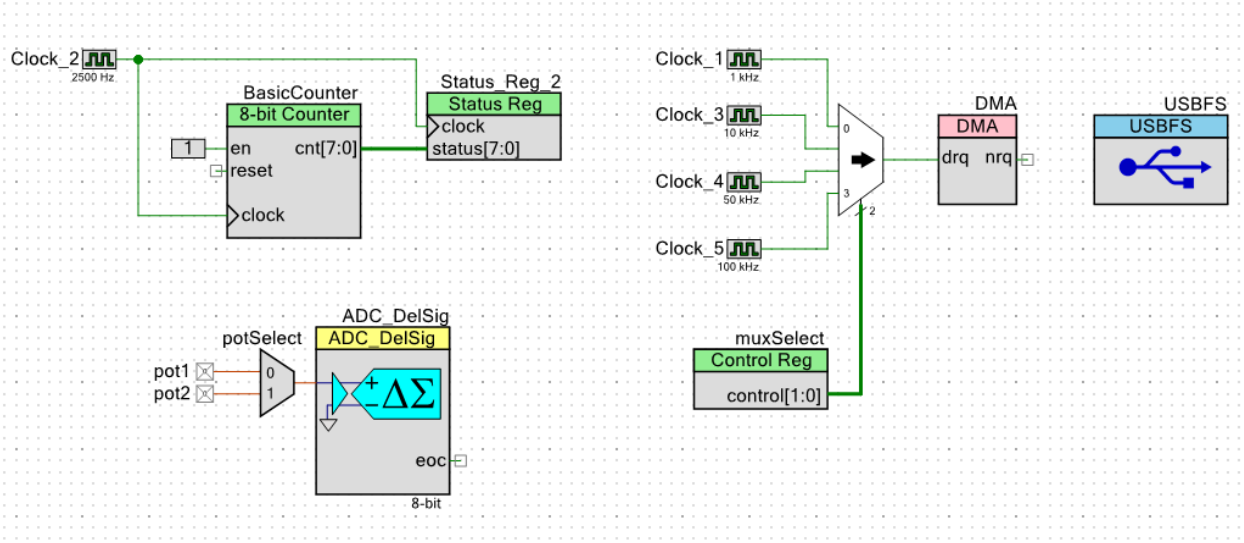
The logic analyzer is something that is slightly similar to the oscilloscope. He logic analyzer takes in a digital signal and then prints the data that is taken in when certain conditions are met. The logic analyzer trigger condition is a one-time event, so unlike the oscilloscope we do not need to take in data constantly from the PSOC. When the trigger event happens, we print out our data, starting with the trigger values. Then we should be able to see some amount of data both before and after the trigger event happened. The logic analyzer is a tool that allows us to analyze digital signals and try to understand what happened by seeing the data around a certain event that happens.

For the logic analyzer, it will be better to start with the Raspberry Pi side first, so we will go to the Raspberry Pi and do as much as we can before moving to the PSOC. The Raspberry Pi is responsible for a fair amount on this part of the project. It must read in the logic expression from the input file, and

display the data that it receives from the PSOC. So first we will do the arguments again. The arguments on the logic analyzer were done the same way as the oscilloscope. By calling the getopt function and putting it in a loop with switch cases, I can scan for all the possible arguments and deal with each one as it comes up. And with the arguments done, I can then move on to the implementation of the logic analyzer. After, reading in the logic expression from the file, I need to get a truth table for the expression. This is done by including the file from homework 1 and then calling the function we created, I can get a truth table. The problem now is what we should do with he truth table. With this truth table we can search for our trigger event, but the Raspberry Pi will not be able to easily ensure that we have enough samples before and after the trigger. This means that it would likely be best to search for the trigger on the PSOC side. This would involve sending the truth table from the Pi to the PSOC. This is not too difficult, but before doing this we should explore the other options available to the user.

We now know that we will have to send the truth table to the PSOC, but that is not the only data we have to send to the Pi. The user should be able to control the frequency, memory depth, and the trigger direction. All of this data must also be sent to the PSOC. The frequency must be sent because that is literally the speed of the DMA clock. The memory depth must be sent because we are searching for the trigger condition on the PSOC. The amount of data we analyze on the PSOC and send back to the Pi will depend on the memory depth. The direction must also be sent as that is crucial to searching for the trigger event. Now that we know what we have to send to the PSOC we can set that up the Raspberry Pi. What we can do on the Pi is simply send all this data once, and then store it in memory on the PSOC side. What this means is that we only have to receive once on the PSOC side also. This can be done on the Pi by calling the libusb_bulk_transfer function as many times as needed. In this case, we will have to call it 4 times to send all the data to the PSOC. Then on the PSOC we simply read in all that data with blocking reads and then we will have the frequency, memory depth, and trigger direction for our uses. The memory depth and trigger do not actually change anything on the PSOCs top design, but the frequency is a different story.

The frequency from the Raspberry Pi user input will change the speed of the DMA on the PSOC top design. To do this I chose to implement a 4 to 1 mux going into the DMA input. The mux's 4 inputs are four different clocks: a 1kHz, 10kHz, 50kHz, and 100kHz clock are all supported by my design. So, the raspberry pi will send a signal to the PSOC, the PSOC will receive said signal and then pick the appropriate clock speed as determined by the user input. By doing this the user can choose their frequency and the frequency argument of the logic analyzer is complete. Below is a schematic of the top design used to do the logic analyzer.



What we see on the left is the basic counter and the status register used for testing/storing the probes data. On the right is the DMA which will read the data from the status register. Note the mux that comes before it. The value of the control Register going into the mux's Select is read from the Pi and from there the corresponding frequency speed is selected. On the very right is the USB which will receive/send information to the Raspberry Pi. On the bottom is and ADC DelSig with two pots that control the cursor and the screen scrolling which will be discussed later. For now, we have the truth table and the relevant user inputs from the Pi, and now we need to search for the trigger event.

To search for the trigger event, the DMA will have to have received certain amounts of data from the status register. To do this I set up a DMA with eight TDs and each TD transfer 2500 bytes of data. Then I have the DMA interrupt on every other TD. With 8 TDs reading in 2500 bytes each, this will ensure that the PSOC has enough data before and after the trigger event occurs. When the trigger occurs, we know the PSOC will have received at least 10000 values, and after it occurs, the next time that the DMA interrupts, it will have read in 5000 values. This is enough values such that even if memory depth is 10000, we will have enough data both before and after the trigger event as per the logic analyzers specifications. Now, to search for the trigger condition on the PSOC we simply check each value from the status register to its corresponding index on the truth table. And when we see a transition from 1 to 0, and the trigger direction is positive, we have found the trigger. And when we see a transition from 0 to 1 and the rigger direction is negative, we have found the trigger. So, the trigger direction is simply another if statement that checks user's direction input and then searches for that condition. When we find the trigger event, we then break out of the DMA interrupt and let it interrupt one more time. We do this so that the PSOC will collect data after the trigger event. Then when it interrupts we disable the DMAs so that there is nothing to interfere with sending the data to the Raspberry Pi.

Now that we have the data, we can simply concatenate it into a single array with the trigger in the middle and then send it to the Raspberry Pi. The Pi will receive this information and then outputs it to the screen using the Line function, just like we did with the oscilloscope.

With our current setup all the Raspberry Pi needs to do is take in the relevant amount of data form the PSOC. In this case, we simply do a libusb_bulk_transfer() of size memory depth and then we

have all he data that the Pi needs to display on the screen. However, we cannot display the data just yet. The data is still in decimal format. This means that before we try to display we need to convert it to binary. This can be done with a for loop that first ANDs the data with a mask and then stores it into a memory array to store the decimals 8-bit data. To display this 8-buit data on the screen we do much the same process that we did with the oscilloscope. We call the line function on the data with its corresponding indices. But we also must scale our binary ones so that they will be visible. This can be done by simply multiplying the data by any value you wish that will still allow all 8-bits of data to be on screen. Then by doing this we can see data on the screen.

The data we see on the screen is not the data we are looking for though. The logic analyzer should begin with the trigger on the screen when it first print. This can be accomplished by simply telling it to begin printing from some samples before memory depth/2. In the data we received the halfway point will be the trigger event so by beginning drawing from some number of samples before that, we can begin with the trigger event on the screen. Something else we want to do is mark when the trigger even occurred. But this should be relatively trivial as it will always happen at the halfway points. All that we need to do to get a line on the trigger point is to call he Line function at the halfway index of the data.

Now that we have a wave on the screen and it begins with the trigger, what we need to do is the xScale input. The xScale input on the logic analyzer is the same modification as the oscilloscope. We simply do the exact same dimensional analysis and that will finish our xScale.

Then with our xScale finished we simply need to get the potentiometer capabilities working. There are two potentiometers and they do two different things. One potentiometer controls a cursor. It is simply a line on the screen that move as the potentiometer moves. The other potentiometer moves the screen so that we can see all the data that the Pi takes in. To set these up we first have to move to the PSOC side and set up our ADC DelSig and analog mux to be able to use them. In the PSOC code I chose to simply send the potentiometer data in the main for loop. This is not necessary, and does create some software overhead, but the amount of data that is sent is not all that much, so it is not too bad.

Now that the potentiometer data has been sent to the Pi, we need to scale the potentiometers such that it is useable. To scale the cursors value, we simply multiply do the following equation:

$$scaledPot = pot * width/255$$

This does something like the oscilloscope pots. It scaled the value of the pots to the width of the screen. So, the cursor cannot leave the screen. No with the value of the pot in memory all we need to do is display it on the screen by calling a Line function and then we can see it move around as the screen updates. The second pot is slightly different though. The second pot needs to be able to scroll through all the data. The equation to allow this would be

$$scaledPot = pot * \frac{memDepth}{255} - \frac{memDepth}{2}$$

By multiplying by the above value, it allows the pot to be a value memDepth/2 before or after the trigger event. Then by adding this potentiometer value to the Line functions x coordinate values, the screen should be able to scroll according to the potentiometers value.

Then by testing all the capabilities of the logic analyzer, we should be finished with the oscilloscope, the logic analyzer, and the project.

Conclusion

This lab was a good way to better understand the capabilities of remotes systems and how they can be used to accomplish many things. The USB is a very versatile tool that can transfer information quickly and allow other remote systems to process and output data in a relatively quick and easy fashion. The PSOC and Raspberry Pi are very small, cheap, and efficient tools in transferring data and can be used for their many capabilities. Whether it's the UART, the USB, or any of the other GPIO pins, these tools can be put to good use as long as we can think of how to implement them.