Leonid Shuster

CSE 121

Lab Section 1C

11/18/19

Lab Report 5

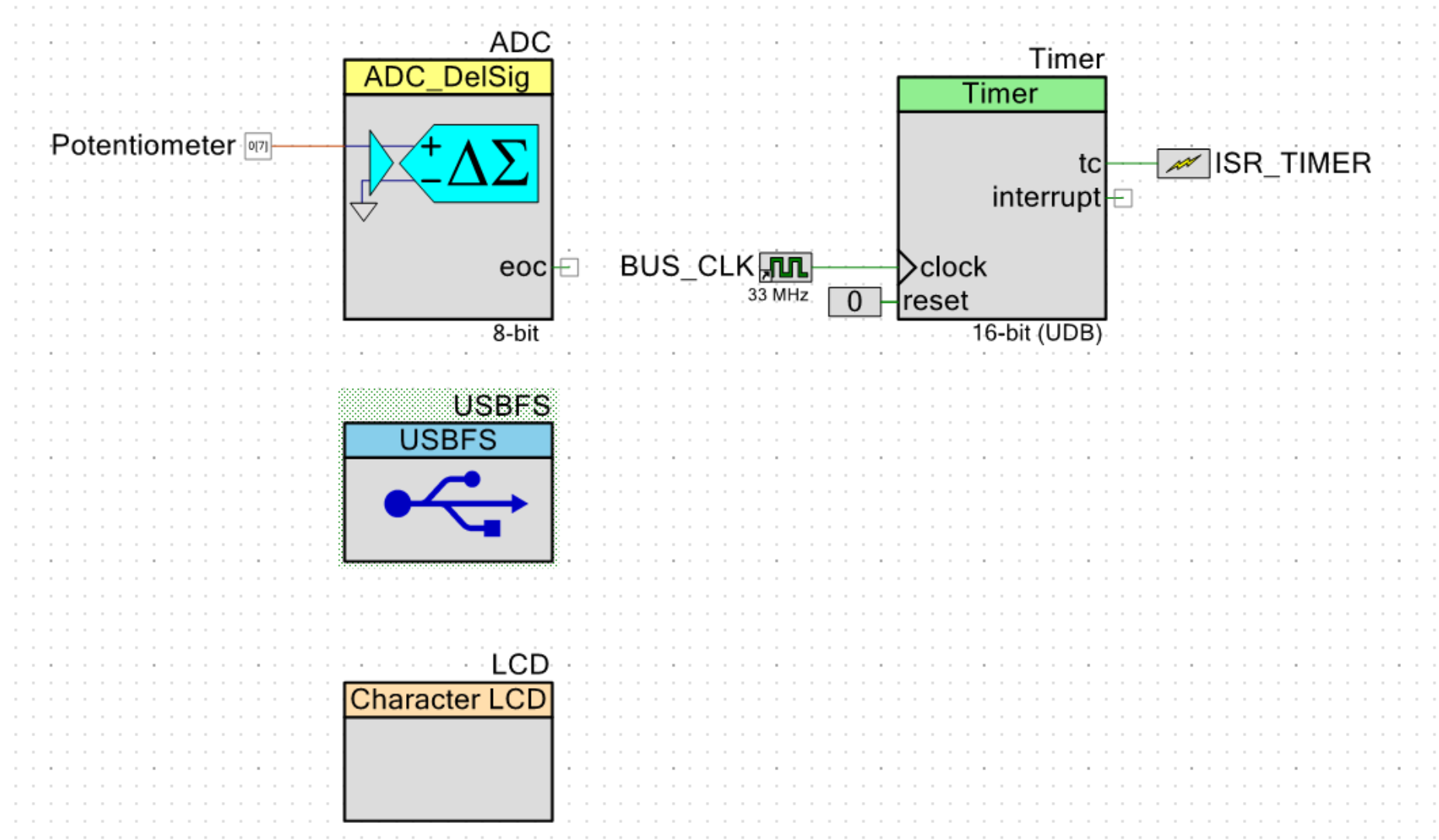
**Introduction**

In this lab, we were tasked with learning how to transfer data between the PSoC 5 and Raspberry Pi using a USB 2.0 link. This lab was the same as the previous one, except now we had to use a USB instead of a UART. Just like before, we started by connecting a potentiometer to the PSoC which was used to control the brightness of an LED that was connected to the Raspberry Pi. And in the second part, we sampled an analog signal with the PSoC and sent it over to the Raspberry Pi which sent it back to the PSoC and reproduced the original signal.

**Part 1: Remote Control of LED brightness with PWM**

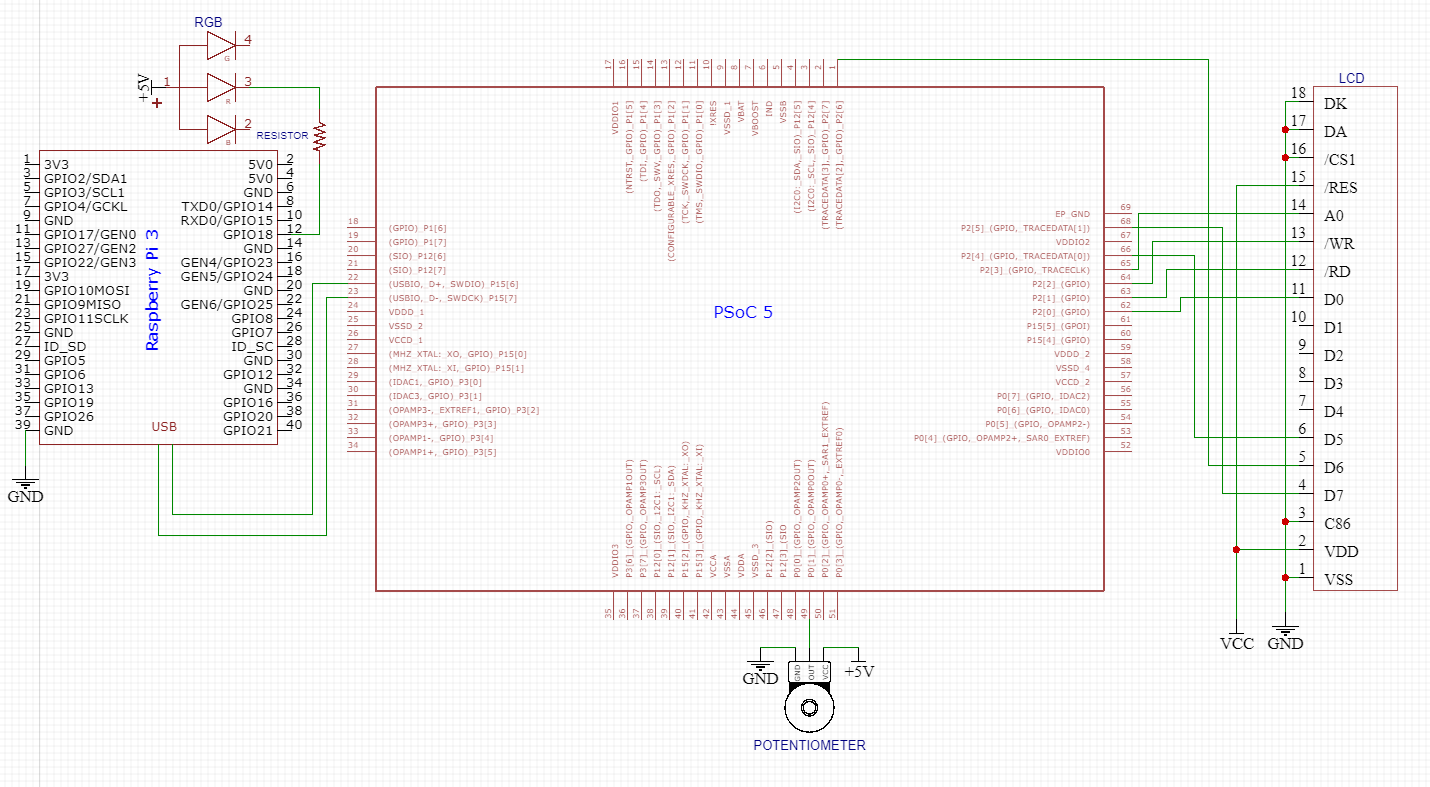
The first part of the lab was the same design as the first part of the previous lab, except now we had to use a USB to transfer data rather than a UART. Again, we were told to use a potentiometer to control the brightness of an LED connected to the Raspberry Pi. However, because the Raspberry Pi does not have any built-in analog blocks and could not receive any analog input, we had to use the PSoC to read in data from the potentiometer, convert it to a digital value using an ADC, and use a USB link to send the data over to the Raspberry Pi which would then read in the data from the USB and a PWM to control the brightness of the LED.

I worked on the PSoC side first, and in my top design, I had a USB component with a single endpoint descriptor configured as going into the host as a bulk transfer, an ADC connected to an analog input pin, a timer that went off every millisecond and triggered an interrupt that would be used to sample the ADC, and an LCD (see Figure 1).



*Figure 1: Part 1 Top Design*

On the outside of the PSoC and Raspberry Pi, I connected a potentiometer to the PSoC input pin going to the ADC, an external wire from the PSoC micro-USB port to the Raspberry Pi USB port, an LED to the PWM output pin of the Pi, and an LCD to the PSoC (see Figure 2).



*Figure 2: Part 1 External Schematic*

In main.c of the PSoC, I set up the timer interrupt to read a 16-bit value from the ADC into a 16-bit integer, checked if the number was less than 0 or greater than 255 and kept it in those bounds, and then stored that value into an unsigned 8-bit integer to be sent over the USB. In the main part of the program, I started all the components and stayed in an idle loop until the USB was configured. Next, I entered an endless loop that checked and got the configuration of the USB if it changed, and then waited until the USB In Buffer was empty to then load in the value from the ADC.

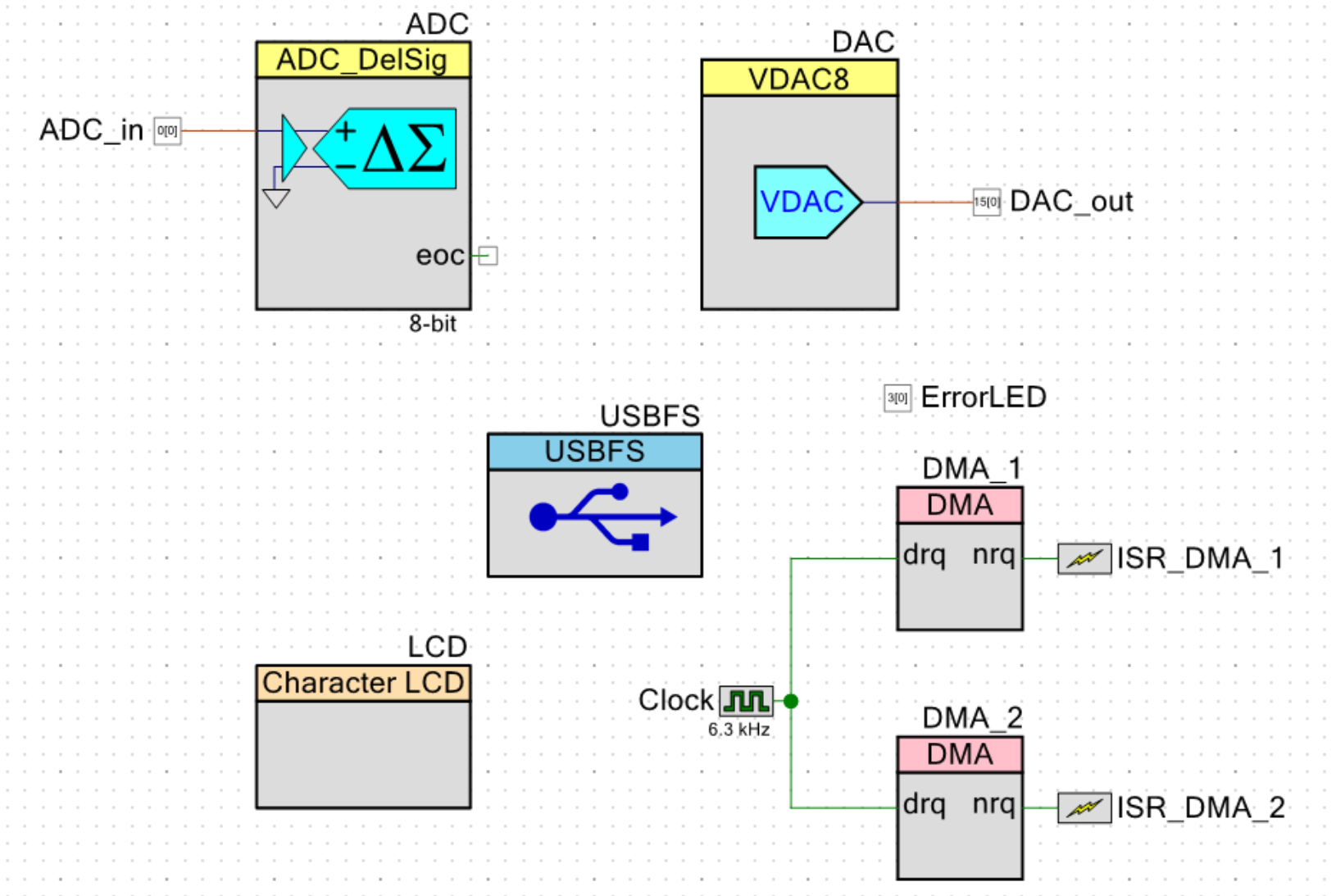
Meanwhile, in main.c of the Raspberry Pi, I used the example code provided by the professor to set up reading from the USB, error checking, and wiring and pin configurations, and used an endless for loop to read from the USB into an unsigned 8-bit integer which would then be scaled and written to the PWM and set the brightness of the LED.

When I first ran the program, I was having problems with my potentiometer. Initially, I was trying to read a 16-bit value from the ADC into a 16-bit integer, and send that value over the USB to the Pi. However, the USB only loads in unsigned 8-bit integers, so I couldn’t read the ADC into a 16-bit integer and put that in the USB, and when I tried reading into an unsigned 8-bit integer, the value would overflow when it passed 255. The solution to this was to use two separate variables, one 16-bit integer that would read from the ADC and bound its value between 0-255, and then another unsigned 8-bit integer that would be equated to the bounded value and be sent over the USB. Thus, I was able to send over the correct values over the USB. Another issue that came up was that the starting and ending values of my potentiometer kept changing, and my LED would sometimes not turn off. To fix this, I increased the value I was writing to the Pi’s PWM by a number large enough that would cover this discrepancy. Finally, one thing that I forgot to implement was an error check in the Pi that checked whether there was an error reading from the USB, which could be done by checking if the return value of the USB bulk transfer function call returned -1.

**Part 2: Analog Loopback through the Raspberry Pi**

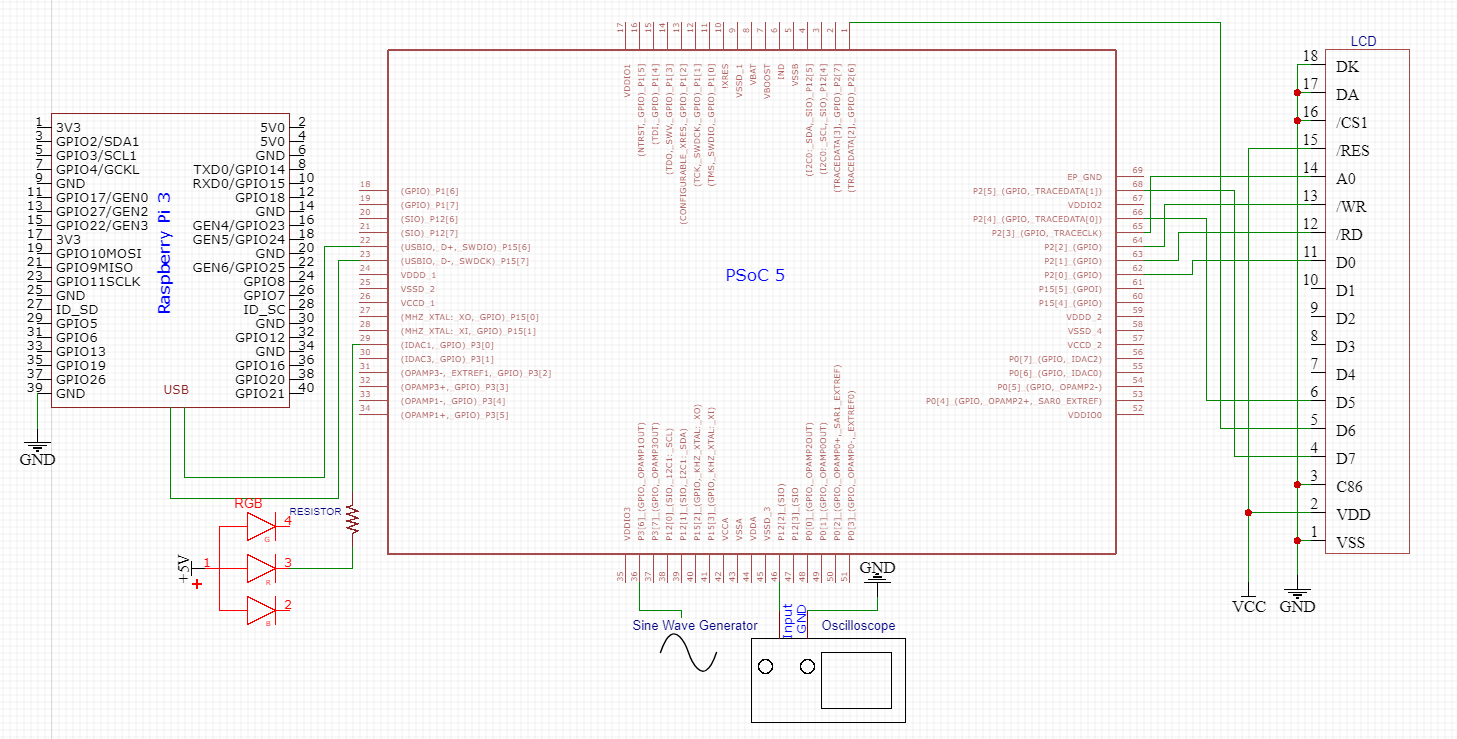
The final part of the lab was again the same as the final part of the previous lab, but now we had to implement it using USB. Just like before, we were told to sample an analog signal with the PSoC, convert the data to digital using an ADC, send it over to the Raspberry Pi, and then send it back to the PSoC, which would then convert the data back to analog using a DAC and display the original analog signal. Because we had to process, send, and receive a continuous stream of data, we also had to make use of ping-pong buffers and DMA transfers to ensure that no data was lost when transmitting and receiving the data.

I again started on the PSoC side, and in my top design, I had a USB component with two endpoint descriptors configured for bulk transfers going in and out of the host, an ADC connected to an analog input pin, two DMAs that used the same clock for their hardware requests and their own interrupts when a transfer was completed, a DAC connected to an output pin, an output pin for an error LED, and an LCD (see Figure 3).



*Figure 3: Part 2 Top Design*

On the outside of the PSoC and Raspberry Pi, I connected the AD2 pattern generator to the PSoC input pin going to the ADC, an external wire from the PSoC micro-USB port to the Raspberry Pi USB port, an LED to a PSoC output pin, an oscilloscope to another PSoC output pin, and the LCD to the PSoC (see Figure 4).



*Figure 4: Part 2 External Schematic*

In main.c of the PSoC, I set up two DMAs and two pairs of 64-byte ping-pong buffers similarly to last lab’s. The goal was to continuously process data without losing any of it, which could be done using a DMA and two ping-pong buffers so that one buffer received data while the other was being read from. The USB didn’t allow us to directly use a DMA to transfer data to or from the USB because it used its own internal DMA, so we had to first use a DMA to transfer data to one of the buffers before giving it to the USB, or wait until the USB wrote to one of the buffers and then use a DMA to transfer from it. Thus, while the first DMA and pair of buffers were used to transfer data from the ADC to the first buffers, the second DMA and pair of buffers were used to transfer data from the second buffers to the DAC. Each DMA was configured with two descriptors in a loop such that each descriptor would transfer 64 bytes and then trigger an interrupt when it was done, which was then used to set a flag that would be used for the USB to decide which buffer to read from or write to. In the main part of the program, I started all the components, and stayed in an idle loop until the USB was configured. Next, I entered an endless loop that checked and got the configuration of the USB if it changed, and then checked if the USB Out Buffer was full. If it was, I checked which ping-pong buffer the USB should read from according to one of the DMA interrupts, and then read from it. Afterwards, I stayed in another idle loop that waited until the USB Out Buffer was not full, and then entered another idle loop that waited until the USB In Buffer was empty. Once it was, I checked which ping-pong buffer the USB should write to according to the other DMA interrupt, and then wrote to it.

Meanwhile, in main.c of the Raspberry Pi, I again used the example code provided by the professor to set up reading from and writing to the USB, error checking, and wiring and pin configurations, and used an endless for loop to read from the USB into a 64-byte array and then immediately write to the USB using the same array.

When I first ran the program, I confirmed that the PSoC’s DMAs’ interrupts were being triggered and that the Pi was receiving and sending 64 bytes, but I was not getting the right signal from the PSoC DAC. However, when I changed my input signal to have an offset of 1 V, all of a sudden I had the right output. Another issue I ran into was that the output signal from the DAC had glitches when the clock frequency I had connected to both DMAs was lower than 6 kHz, but those glitches went away when I increased the frequency. Finally, I again forgot to implement an error check in the Pi that checked whether there was an error reading from or writing to the USB, which could be done by checking if the return value of either USB bulk transfer function call returned -1.

When inputting signals higher than 100 Hz, reproducing the signal began gradually resulting in glitches on the waveform. In order to fix this, I believe that the DMAs’ frequency and sample rate of the ADC should be raised in order to accommodate for this. However, at some point, I believe that one of them would not be able to physically run at such a high frequency, and so the signal could not be reproduced faithfully.

**Conclusion**

After having gone through the lab, I feel like I fundamentally understand how to use a USB to transfer data between devices. From sampling a potentiometer’s value with the PSoC and sending its value to the Pi to light up an LED in the first part and reading in an analog signal with the PSoC, sending it to the Pi, and then back to the PSoC to reproduce it in the second part, I feel confident that I can effectively use USBs to serially communicate between devices. If I were to do this lab again, I would implement more tests for error checking on both the PSoC and Raspberry Pi to ensure that the data sent and received across the USB transfers without any errors.