Leonid Shuster

CSE 121

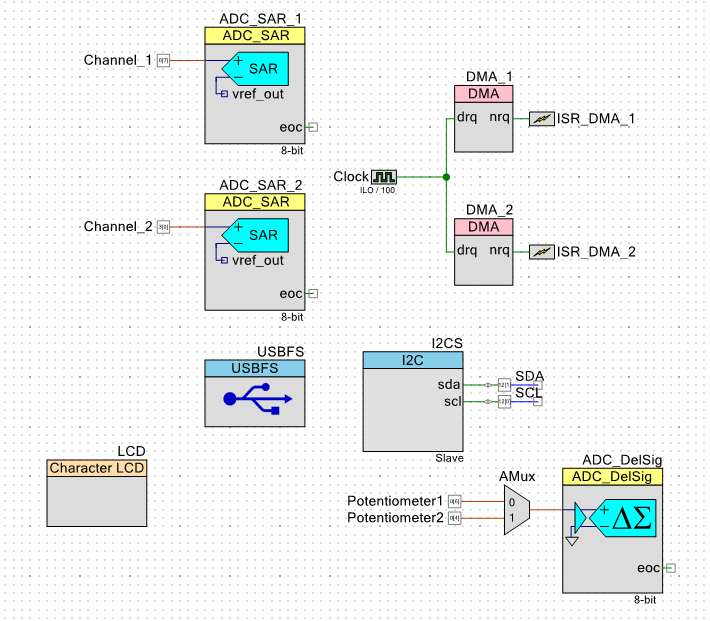
Lab Section 1C

12/10/19

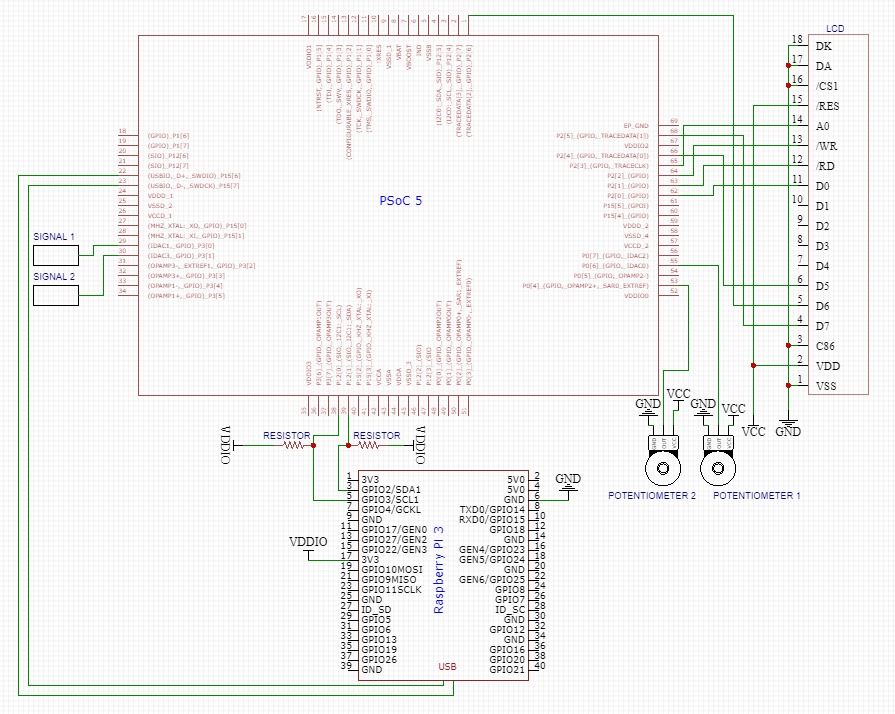
Lab Project Report

**Introduction**

In this lab, we were tasked with creating a dual-channel oscilloscope using the PSoC 5 and Raspberry Pi. The Pi was to take in command line arguments to set certain parameters for the waveforms, and then send a signal to the PSoC over the I2C bus to start sampling two analog signals at a certain sample rate and reading from two potentiometers. The PSoC would then sample the signals and send their data over the USB back to the Pi and also read from the potentiometers and send their data back over the I2C bus. The Pi would then use the data from the signals and the command line parameters provided to plot the waveforms and use the potentiometers to shift the waveforms up and down.

****

*PSoC Block Diagram*

**

*External Schematic*

**Description of Features**

To start the program, the user runs the executable on the Raspberry Pi with possible optional command-line arguments, which are then parsed and processed. If no command-line arguments are provided, the program runs with default values of mode = trigger, trigger level = 2500, trigger slope = pos, sample rate = 1, trigger channel = 1, xscale = 1000, and yscale = 1000. Otherwise, the user can change any of these arguments by putting in valid arguments on the command-line. Once the program starts running, the Pi sends the sample rate and a start signal to the PSoC over the I2C bus to start sampling the analog signals. The PSoC begins sampling at the specified sample rate, and uses two DMAs that transfer data from the SAR ADCs connected to the analog signals to two sets of two ping-pong buffers that switch off every time their DMA’s interrupt triggers, with each set of ping-pong buffers then being used to load data into its own separate endpoint descriptor in the USB to be sent to the Pi. The PSoC also starts reading from two potentiometers that are connected to an analog mux that chooses which potentiometer gets fed into a Sigma-Delta ADC to be converted and then sent over the I2C bus. In the meantime, the Pi processes the command-line arguments, and calculates the number of samples and scales the yscale accordingly. Then, the Pi sets up the background and scaling, and in a continuous loop, reads from the USB from each endpoint into two buffers, calculates the starting point of the waveforms according to the trigger level and trigger slope, reads the potentiometer values coming from the I2C, and then plots the waveforms accordingly.

**System Description**

The PSoC was connected to two analog signals coming the AD2 signal generator, two potentiometers, an LCD display, two wires coming into the SDA and SCL pins coming from the Pi’s SDA and SCL pins, and a USB that connected to the Pi. To connect the SDA and SCL pins between the PSoC and the Pi, I had to configure the PSoC to have a VDDIO of 3.3V which was done by soldering off the R15 resistor from the back of it and supplying 3.3V to VDDIO from the Pi, and then I had to pull up SDA and SCL to 3.3V with resistors since they were bidirectional.

**Hardware Design**

In the top design of the PSoC, I added an I2C Slave with its sda and scl bidirectional pins connected to the pins for SDA and SCL communication that would receive the start signal and sample rate from the Pi and send the values of the potentiometers back over. I also had two SAR ADCs that took in the analog signals to convert to digital as an input, and set them to an 8-bit resolution with the maximum conversion rate. Next, I had two DMAs with their hardware requests connected to a single clock with a divider value that would be changed based on the sample rate received from the Pi. I also had a USB configured with two IN endpoint descriptors for bulk transfers that would send the converted data from the analog signals to the Pi.

Next, I had two analog input pins for the potentiometers going to an analog mux that chose which potentiometer went to the Sigma-Delta ADC which was set to an 8-bit resolution with a conversion rate of 10000 to convert the analog input to digital. Finally, an LCD for error checking.

**Software Design**

For the software in the PSoC, I had two sets of two ping-pong buffers in which the two buffers would switch off which would be filled with data from the SAR ADCs while the other would read from the USB. I configured two DMAs to each transfer data from their own SAR ADC to one of the ping-pong buffers by using transcription descriptors in a loop. I had two flags for each DMA that would change when the DMA finished transferring data to one buffer and switched to the other that would determine which buffer should be read from the USB. I also had another flag for the analog mux that would switch everytime the mux selected one of its options. In the main loop, I initialized and started the I2C, and waited in an idle loop until the I2C bus transferred the starting signal and sample rate, which I then used to set the frequency of the clock connected to the DMAs. Next, I initialized all the other components in order to start sampling the analog signals, and in an endless for loop, I checked whether the I2C slave buffer and read buffer was read, and if so, cleared the read status, disabled interrupts, and checked what the mux flag was set to in order to select one of the potentiometers and read from the Sigma-Delta ADC and read it into a variable that would then be written into the I2C bus. After that, I checked the state of the USB endpoints to see whether their in buffers were empty, checked what each DMA flag was set to, and loaded in the proper buffer for the USB to read from.

For the software in the Pi, I tried to organize my code into different files based on their functionalities, so I had one file for reading in the arguments from the command-line, one for reading data from and writing data to the I2C bus, one for reading data from the USB, one for processing the command-line arguments and adjusting accordingly, one for processing the trigger level and slope, one for plotting the waveforms, and a main file to facilitate the program flow and just call functions, as well as a header file that spanned all the files.

In the file in charge of reading the command-line arguments called cmdargs.c, I set up an argument struct that would store the arguments’ values from the command-line arguments, and initialized them to those specified in the lab manual. In a function, I read from the command line argument by argument using the getopt function that would get an argument and the value next to it and then used a switch case statement to check which argument was being read. Within a single case, I checked if the value was valid and if so, changed the value within the arguments struct to the one just read, otherwise I printed an error message and exited. All this repeated within a while loop until the getopt function returned -1 meaning that it went through all the arguments.

In the file in charge of writing to and reading from the I2C called i2c.c, I had one function that set up the I2C wiring and checked if the device was not found and if so, printed an error message and exited, otherwise I sent the sample rate, also the start signal, over the I2C bus. In another function, I read from the I2C bus into two variables that would hold the potentiometer values.

In the file in charge of processing the command-line arguments called data.c, I had a single function that calculated the number of samples based on the xdivisions of the plot and the sample rate and xscales from the command-line arguments.

In the file in charge of reading data from the USB called usbcomm.c, I had one function that initialized the USB and opened it, checked if it was found, reset correctly, configured correctly, and claimed the interface, and printed an error message and exited if otherwise. In another function, I read from the USB with a while loop that ran for the number of samples calculated from the command-line arguments. In the while loop, I read from two endpoints of the USB into two buffers that shifted the starting point of the array being written into. I also checked whether either of the transfers failed at any point and if so, printed an error message and exited.

In the file in charge of processing the trigger level and slope called trigger.c, I had one function that converted the trigger level from volts to an 8-bit number, checked what the mode from the command-line arguments was set to, and then called a helper function that would go through the data buffers from the USB until the trigger condition was met. In the helper function, I had an endless for loop in which I iterated through the data buffer and checked for rising or falling edge trigger conditions. For rising edge, the previous data sample had to be less than the trigger level, the current sample had to greater than or equal to the trigger level, and the current slope had to be positive. For negative edge, the previous sample had to be greater than the trigger level, the current sample had to be less than or equal to the trigger level, and the current slope had to be negative. When one of these conditions was met, I broke out of the loop and returned the index from which the data points fulfilled the condition. For the slope, I had to take an average of 4 data points in order to accurately calculate the slope and avoid glitches.

In the file in charge of plotting the data onto a plot called graphics.c, I had one function that got the screen size, initialized the display, and started it. In another function, I calculated the pixels per volt, cleared the window, started the display again, called a helper function that drew the background, printed the scale settings onto the screen, and processed and plotted the wave using the data buffers and the starting index that satisfied the trigger.

Finally, in the main file called main.c, I called whatever function I needed. First, I got the command-line arguments and put it into a struct. I then wired the I2C and wrote the sample rate and start signal over the I2C to tell the PSoC to start sampling. Then I initialized and opened the USB. Next, I processed the command-line arguments. I then set up the plot, and in a continuous loop, got data from the USB, found the starting point based on the trigger, read from the I2C to get the values of the potentiometers, and then plotted the data.

**Testing**

I tested my program every step of the way. Every time I wrote a file that was dedicated to a certain task, I called its functions in main to check whether my program was still working. I started with reading in the command-line arguments, since I thought that was the easiest thing to start with. I wrote a simple if-else chain, and I ran my program with various outputs to see if it worked. I was getting errors with empty arguments, so I decided to use the getopt function and switch case statement that made it a lot easier to go through the arguments. After that, I set up the PSoC to read a single analog signal and send over its data to the Pi, and checked whether the Pi was receiving the right data by looking at the numbers being received to make sure they resembled the pattern of the shape I was sampling. After that, I worked on understanding how to plot waveforms onto the screen from the Pi, and then tried to plot the data I was receiving in the Pi. After messing around with the number of samples, I was able to get somewhat of a shape resembling that of the original analog signals. Next, I tried to figure out how to process the trigger, and tested my functions with the data I was receiving from the USB and printing the starting index from the trigger and eventually integrated the trigger into the plotted waveform. Next, I processed command-line arguments in order to get the number of samples, which I then integrated with reading from the USB to only read a certain amount and plotting on to the waveform. After that, I added a second waveform and adjusted in reading and writing from the USB in the PSoC and reading from the USB in the Pi. Finally, I integrated I2C in the PSoC and Pi to communicate data about starting sampling and the values of the potentiometers. When I ran my program with everything put together, my waveforms looked extremely messy, almost as if the two waveforms’s data were interacting with each other and creating data loss. I later found out this was due to continuously opening and starting the USB and I2C, which was causing major data overhead and loss. After fixing this, my data looked a lot cleaner.

**Conclusion**

After having gone through the lab, I feel like I understand how devices process data and communicate data on a fundamental level. This lab taught how to make multiple components with various timings all working together, and how to put together a project from everything we’ve learned throughout the quarter. If I were to do this lab again, I would make sure that the memory usage in my project is a lot cleaner and efficient, because I wasted a lot of time with the data corruption of the signals.