CMPE121 L

Xingyu Zhou

Anujan Varma

2019/6/9

**Final Project: Dual Channel Oscilloscope**

**Introduction:**

This is the last lab of the class and this time we were asked to build a dual channel oscilloscope using the PSoC 5 and the raspberry Pi. It was a combination of basically all of our previous labs since it involved the use of DMA, USB as well as I2C. This lab was consisted of two parts: the first part was to deal with the PSoC. All we needed to do is to convert the analog signal to digital signal and send them to the raspberry Pi using USB. The second part was the raspberry Pi. After receiving the signal, we need to process the data on the Pi side and display the waveform on the screen using the open-VG library. Here’s a list of things you can do with the Dual Channel Oscilloscope:

* Mode : Free running or Trigger. This is set by typing “-m trigger” or “-m free\_running”. The default mode is trigger.
* Trigger level: the voltage of trigger in millivolts. The range is from 0 to 5000 with a step of 100. This is set my typing “-t somenumber”. The default is 2500;
* Trigger slope: default is positive. Determine if you want to trigger upward or downward. This is set by typing “-s pos” or “-s neg”;
* Sample rate: Determine how fast you want to sample two ADCs. This is set my typing “-r somenumber”. The sampling rate can be 1k , 10k, 20k, 50k, 100k.
* Trigger channel: channel 1 is the default channel. Can be set by typing “-c 1 or 2”.
* X and Y scale: This is set by typing “-x 1000” and “-y 1000”. These parameters are in microseconds and in millivolts accordingly.
* The position of the waveforms can be controlled by the potentiometers.

The following lab reports will be broken into two parts: **PSoC 5** and **Raspberry Pi.** I will talk more about the project in details in the following paragraphs. The external wiring diagram is attached to the end of the report.

**Part 1: Configuring the PSoC5 (Hardware & Software)**

A close up of a map

Description generated with high confidence

**Figure 1: Top Design of the PSoC 5**

The PSoC 5 board in this project serves as an analog front end of the system. The job of the PSoC is to capture the analog signals for the two channels, sample the signals, convert it to digital values and then send them to the raspberry Pi through USB. Similar to Lab 5 and Lab 4, I set up two ping pong buffers using separate DMA channels so that I could move the data from each ADC separately. The USB was also configured to two IN transfer instead of one IN and one OUT since we did not need to receive data from the raspberry Pi. I used SAR ADCs since they had the highest sampling rate according to the lab manual and the configuration was set as follows: 8 bit resolution, free-running sample mode with the input range of Vssa – Vdda. The input mode was set to single ended since a differential out put would not be needed. The set up process of the DMA and USB was basically the same as lab 5 so I will skip them for now.

A new component was introduced to us during the lab project-I2C. The I2C was served to receive commands from the Pi, as well as to send data of two potentiometers to Pi. According to the lab manual, Pi should read the potentiometer setting through I2C around every 100 milliseconds. This was achieved by using a timer. The timer interrupt was configured to trigger every 100 milliseconds and the data was read by Pi inside my timer interrupt. The I2C component was taken directly from the I2C example code provided by the professor.

As you may notice in my top design, a component called AMux\_1 is included. The purpose of the mux was to switch between my two potentiometers, since we needed to control the two waveforms separately. A function called Mux\_FastSelect() was used to fulfill this purpose.

Since the Raspberry Pi runs on a max voltage of 3.3 volts and the PSoC is at the default of 5 volts for the digital signaling, I removed the zero-ohm resistor R15 and connect the VDDIO and GND of the PSoC directly to the VDD and GND of the Raspberry Pi.

**Testing the PSoC design:**

To make sure that each potentiometer was working independently, I modified the provided testing file called “i2ctest.c” so that it would print out values of both potentiometers. I also used the “usb\_bulk\_transfer.c” to make sure that I was transmitting data and print out what I got on the Pi side. In order to minimize the errors, I implemented two blocking calls(check if the IN buffer is empty) before I sent the data to Pi. Both testing were success and I was ready to move onto the Raspberry Pi.

**Part 2: Configuring the Raspberry Pi (Software)**

The Raspberry Pi was responsible for collecting the data from the PSoC, process it and then draw the waveforms on the screen. A library called OpenVG was used to draw graphics on the screen. Moreover, we were also asked to set various parameters to alternate the waveforms as well as to interacting with the PSoC. In stead of having a huge trunk of code that did everything, I broke it into several header files as well as c files and I will talk about what each of them does individually.

1. **cmdargs.c / cmdparse.h**

This source file serves to parse the command line argument so that I can change the settings of my oscilloscope. Inside my header file I had a struct that contains all the settings that can be changed (mode, trigger levels, etc). I also had two enums that hold the trigger slope as well as the mode. Here’s how I declared my function:

int parseCommand (struct parsingCommand \*Command, int argc, char \*argv[])

I used atoi() in most of my parsing actions because it was easier to use rather than comparing each input arguments individually. For example the trigger level. The range of the trigger level was from 0 to 5000 and if I were to compare each number individually, it would take a lot of time. What I did was to assign the return value of atoi() to an variable and then compare it with other conditions.

I also checked for invalid inputs in my parsing function and returned with an error message when something went wrong. Something to notice was that the atoi() would return a 0 if the input argument was not numerical, which meant my trigger level would be set to 0 if the input argument was a letter. In order to deal with that, I added a condition saying that the trigger level will only be 0, if both and input number is 0 and the return value of atoi() is 0.

1. **data.c/data.h**

This source file serves to take in the data transferred from PSoC using USB, and then fill up the channel data buffer according to the sample size. The sample size is determined by this formula :. I got this formula from a fellow classmate. Since the USB can only transfer 64 byte data at a time, using this data to draw directly would result in an incomplete waveform. The way I chose to use was to set up a simple for loop. When the sample size was determined, the program would keep putting the data coming from the USB into the channel data buffer, which would be use to draw the waveform, until the sample size is reached. To avoid any errors, I had a second counter that kept track of the index of the USB data buffer. Once the value was equal or greater than 64, it would be reset to 0. This program took in an integer, which was sample size, and two char arrays, which were USB data buffer and the channel data buffer as it’s arguments.

1. **usbcomm.c/usbcomm.h**

This source file was similar to what I had in lab 5. Its purpose was to initialize both my i2c and USB, and process the data accordingly. For the USB part, I used my USB data transfer code in lab 5 and broke it into two functions. One was called usbinit() and the other was usbread(). Usbinit() didn’t take in any arguments and all it did is to set up the device and stop the program if any errors are found. The usbread() took two char arrays as its arguments, and what it did is to take the data coming from the PSoC and store it into two arrays.

For the I2C part, I used the example code (i2ctest.c) provided and again broke it into i2cinit() and i2cread(). The only thing different was that the i2cread() took an integer called command and two pointers (for both potentiometer readings).

1. **graphics.c/graphics.h**

This source file handled all the drawing part. I couldn’t find other methods to draw the waveform so this entire drawing functions were taken directly from the example code to ensure maximum accuracy.

1. **Main.c**

Even though I had several source files that served different purpose, I still needed something that can connect all of them together. This was when the main.c came in. The first thing I did was to call the function parseCommand(). This gave me the access to the current scope settings so that I could change them and print them out to the console. Then it was the variable declaration section. Some of the variable declarations were taken from the scopedemo.c provided by the professor. The sample size mentioned in data.c section was also declared in main. One thing to mention was that the size of my channel data buffer was set to the same as my sample size. This allowed my code to allocate the memory dynamically. The set up code before the infinite while loop was the same as the scopedemo.c.

Now that the program entered the infinite loop, the first thing it did was called i2cinit() and usbinit(). After both devices had been initialized, I then moved the data from USB to my own buffer. This step was performed twice—I needed to skip the first set of data that was being passed in since it contained too many data losses. After that I would filled up the channel data buffer with the data collected from USB according to the sample size that was determined at the very beginning. The data was now ready for process. The function processSamples() was called and the data would be converted to coordinates. After that these coordinates would be used to draw out the waveform using the function plotWave(). In the function plotWave(), the argument yoffset was replaced with my potentiometer readings so that I could move the waveform up and down.

There are still one more thing—**the trigger detection**. I did the trigger detection inside my main so that I didn’t need new sours files. Here’s how I did it:

**first of all,** I declared two new channel data buffer that had the size of the sample size and expand the size of the original channel data arrays to 3 times of the sample size. The purpose was to make sure that I had enough space for the code to detect the trigger.

**Second,** I set up multiple if loops to check the current scope mode such as what channel and trigger slope I was at. The formula for trigger detection was already provided on the lab manual so I just transcribed it into codes. The trigger value is obtained using this formula:

**Third,** while the trigger hasn’t been detected, a variable (here I called it b) would keep increasing to keep track of the current index of the channel data buffer.

**Fourth,** After the trigger has been detected, the index would increase once more, and then break out of the loop. In my own opinion, the waveform should be drawn after the trigger, that was why I’m adding 1 to the current index, because I was treating the current index as the “trigger”.

**Finally,** I moved all the data from the old channel data buffer to the new channel data buffer. This process was done in a for loop. While the index hasn’t reach the sample size, I would keep putting the data from the old channel, plus the index b that I obtain from step 3 (in other words, the data after the trigger ) into the new channel data buffer. After this process was done, the new data buffer was used to plot the waveform with the trigger enabled.

**Testing with the Raspberry Pi and the PSoC:**

Once the system was built, I tested the system by applying a 1k Hz sine wave to both the first and second channel and set up my scope using the specs provided in the lab check off plan manual. One thing to notice was that I didn’t implement the trigger function until I make sure everything else is working.

When I first started testing, I was getting a lot of data loss on the waveform even after I skipped the first set of data. My theory was that my sampling rate for DMA was too fast that the data loss happened during the process of switching between Ping-Pong buffers. However by changing the sampling rate using the “-r” command from the Pi, the data loss still happened. I then moved to the PSoC side to alternate the clock that was connected to my DMAs. After trying different clock frequencies, I noticed that the waveform was most stable at 10.65k Hz. So instead of changing the sampling rate using Pi, I had it set to a fixed frequency.

Here was when the strange things happened. After I made sure everything is working correctly, I implemented the trigger detection function and ran the test again. This time I was not getting any waveforms on the screen. I commented out everything except the declaration of the new channel data buffer since I didn’t think it matters. After running the test again and still got nothing, I commented out the declarations as well—this time the waveforms showed up. I suspected it has something to do with memory allocation so I tried using malloc, but it didn’t work. I went back to my other source files and changed some method that might be using too much memory but still it didn’t work. Every time I declared these two new data buffers (just declaring the buffer, not doing anything else), the waveform would disappeared. I had tried every method that I can think of but none of them works so I still included the trigger code inside (commented) and got ready for the check off.

I was using a different computer at the time for check off and after I got the first checked off, I uncommented out the trigger detection code and ran again. This time I was getting a steady waveform with working trigger on the display. I had no idea what was going on because the code was literally the same, all I did was pull it from gitlab. The trigger slope was still not working properly since I couldn’t debug it if there was no image, but at least I got the second check off with the partially working trigger.

**Conclusions, Lessons learned**

The lab duration was fairly long since we had 3 weeks to build it. One lesson I learned was to work incrementally, break the whole assignment into many small pieces and finish them one at a time. The PSoC side was fairly easy to implement so most of my focus was on the Pi side. Another lesson I learned was that don’t put too much attention on the details at first. Make sure that the project is working at a minimum basis, then start to add more functionalities to it.

If I had more time (that is, if I didn’t wait until the last week to start), I would have pay more attention to the data processing part on my Pi. Since this was the part that I suspected to be the reason of the disappearing waveforms.

This is a fun project, I enjoying working on it and I learned a lot about cooperating with two different systems. I had a deeper understanding of the Raspberry Pi and PSoC 5 as well as other components such as ADCs, USB, DMA and I2C. I also learned a bit about how the oscilloscope works. I’m planning on fixing the trigger code of this project and make it work this summer.

All in all, this is a fun class and thanks for the professor, TAs and tutors for being nice and helpful.

A close up of text on a white background

Description generated with very high confidence

**Figure 2: Final Project External Diagram**