Microprocessor System Design Final Project Report

Introduction:

For the final project, I created a waveform plotter using the PSoC 5 (hereinafter PSoC) and Raspberry Pi (hereinafter "the Pi") communicating over both USB and I2C communication channels. The final design implemented all the following features prescribed to us by the project manual:

1) The system must take in command line arguments that adjust the settings on the final output. These settings include trigger mode, trigger level, trigger channel, trigger slope, sample rate, x axis scale, and y axis
2) scale.
3) The system must support all waveforms with a frequency of 1 kHz or less and a voltage of 0 to 3.3 volts.
4) The system must support a sample rate of 1 Megasamples/second
5) The system must trigger on any level in the voltage range and on either positive or negative slope
6) The final output must use the OpenVG graphics library
7) The user must be able to move the waves up and down using two potentiometers connected to the PSoC

This project had several purposes

1) To teach us how to apply I2C
2) To further our application of USB
3) To apply the OpenVG graphics library

I broke this project down into several smaller modules: PSoC Top Design, PSoC main.c, cmdargs.c, usbcomm.c, I2C, external hardware, main.c, data.c, and graphics.c. I will discuss each in detail roughly in chronological order of when I began working on them. I will discuss major changes and updates, as well as a few of the major issues I faced while developing this project.

PSOC TOP DESIGN

My final design on the PSoC went through several changes. First, I added a copy of my top design for lab 5.2. In lab 5.2 I recreated a wave from a source by taking measurements with a Delta Sigma ADC, I then used a DMA to periodically take measurements from the ADC and record it into one of two 64-byte ping pong buffers. The DMA operated by filling one of these, then trigger and interrupt, then fill the other one, then trigger the same interrupt. The DMA proceeded in this loop forever. In the ISR triggered by the DMA, I set a flag that swapped between two values depending on which buffer the DMA currently operated upon. This ensured that the rest of the program never disrupted the DMA. In the main loop I used USB to transfer the array not used by the DMA into the Pi. The Pi then echoed back the same array which the USB stored in one of two similar ping pong buffers. Finally, a second DMA then transfered one of the two buffers into a VDAC. These two buffers operated precisely the same as the input ping pong buffers in reverse. The DMA operated on one of them while the USB transferred to the other, then the DMA triggered an interrupt and they swapped. This flow successfully recreated wave forms. For further details on lab 5, see my lab 5 report.

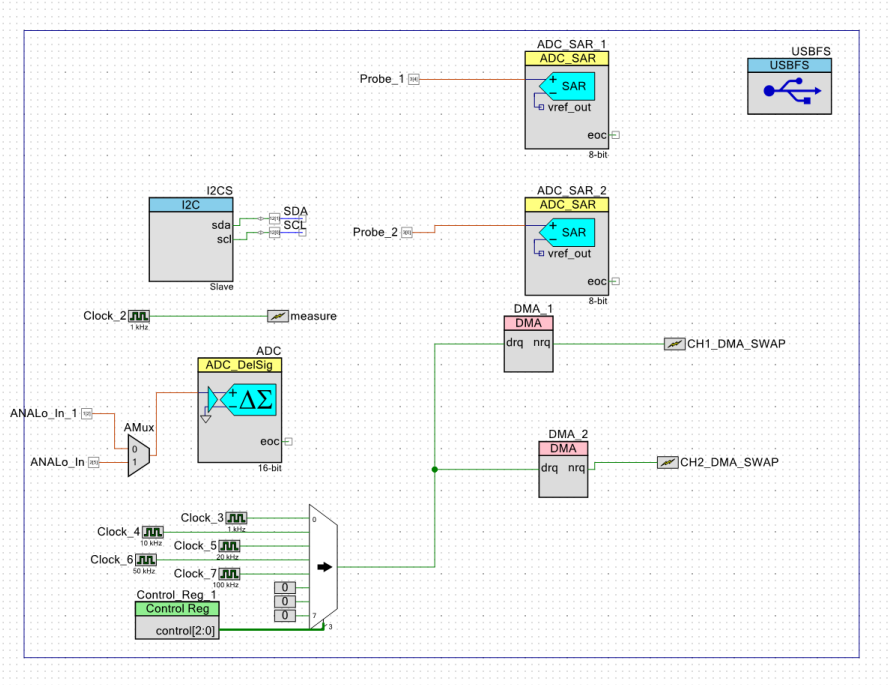
I immediately removed the analog output pins and the VDACs and replaced them with one extra analog input pin. I named these Probe 1 and Probe 2. I replaced the Delta Sigma ADC with two SAR ADCs and added a second DMA on the same clock as the first with identical setting. As with lab 5.2, I connected

both DMA nrq outputs to interrupts which controlled flags that controlled which data sets sent to the Pi over USB. I changed the USB module to instead of using one input EP and one output EP, I used two output Eps. I left the LCD char display unchanged because I wanted it around for debugging purposes. I also added an interrupt on the second DMA just like the first. I left this for a while to test and implement other parts of the project.

I used the SAR ADC because they have a higher sample rate at higher resolution. Even at 8-bit, where I planned to operate it, the SAR ADCs are the better option to support 100 kSpS. I used two output EPs on the USB module because that ensured that the function could not confuse data between channel 1 and channel 2. Even if they used the same wire, if the data transfer used different addresses, one would have to use that specific EP to access the data associated with that EP.

I came back the top design to create the I2C module. I modified the I2C module from the example code, which turned out like what I needed. I removed the LCD display module because I didn't need it anymore. I also added two analog input pins connected to a 2x1 analog mux. These were connected to the potentiometers (hereinafter "pot") which controlled where on the screen the waves appeared. The Delta Sigma ADC then measured this mux and thereby one of the two pots. I also added an interrupt called "measure" controlled by a 1 kHz clock that periodically triggered a measurement of either values of the pots.

The final major change came when I removed the static clock which controlled both DMA modules. I replaced this with five clocks at all the required sample frequencies connected to an 8x1 mux. This allowed me to select which clock I wanted to control the DMA using the selection input of the mux. I controlled this with a 3-bit control register that I controlled using software. I filled the three remaining spots with logical zero to prevent any unexpected errors from the mux. I chose to build it this way because I didn't want to mess with clock dividers in software, and this method appeared the best alternative I saw after a discussion with one of the tutors. After I finished working on this, I had no further need to change the top design.



PSOC MAIN.C

Much like my top design, my main.c file on the PSoC went through a few large changes. At first, I simply copied all the code from my lab 5.2. I removed all the code for the DMA which served as an output in lab 5.2 and replaced it with a copy of the input DMA with changes to make it operate on DMA 2 instead of DMA 1. I at first had the USB continuously transferring if the EP state returned empty on both EPs, but after some testing in the Pi, I realized that this transfered the same data repeatedly if allowed, which caused breaks in the data (see usbcomm.c). To fix this I moved the transfer code into the interrupt that previously only dealt with the DMA flags. This seemed to work, but I developed issues with "stale" data and breaks in the waveform on the Pi side (see usbcomm.c). After some discussion with a tutor, I moved the transfer code back to the main loop but added a flag for each EP that set inside the interrupt and cleared whenever the USB transferred the data.

The ISR for the I2C components triggered every millisecond based on the top design. The ISR first swapped a flag between two values representing pot number one and pot number two. This controlled an if statement that switched which value outputted from the analog mux in the top design to the ADC. I then read the value of the ADC and stored it in the variable pot1 or pot2 depending on which input the mux passed. I then broadcast both values using the I2C code given to us.

The last change I made to the I2C interrupt checked if the Pi had sent a value over to change the sample rate. If it found a new value, I used a switch statement to change the control register value, which changed which input on the DMA mux passed to the DMAs. This effectively changed the clock frequency seamlessly without needing to mess with inaccurate clock dividers. By default, nothing happened to ensure that if I2C transferred bad data, a change in the mux wouldn't disrupt the data stream. After I had this component functional, I had no further need to modify the PSoC side of the code.

I chose to do all the I2C code inside of the interrupt because it never systematically disrupted the data flow. If I had to do it again, I might move the code associated with the switch statement inside the main loop because I know it can execute quickly and doesn't need to be in the interrupt. I also might move the potentiometer variables from global variables to statically declared inside the interrupt. Notably, both are minor optimization changes that would not require a major overhaul.

CMDARGS.C

I decided to do cmdargs.c first for several reasons, primarily it comparatively had little to do with the rest of the project, and because of that I could work on it once and move onto other parts. It also introduced me to all the features the project needed to implement without having to directly work on them yet. While many of my peers chose to pass in a struct and save values directly to that, I chose to pass in numerous variables by reference because I felt this would avert future confusion inside main.c and thereby sacrifice readability inside cmdargs.c for increased readability in main.c. Since cmdargs.c didn't contain much complexity compared to main.c, I decided this was a worthwhile trade.

The file cmdargs.c implemented only one large function which, in a spurt of creativity, I named cmdargs. The function had multiple arguments, it passed the number of arguments from the command line by value, but everything else, including the array of command line arguments and all the variables representing the oscilloscope settings by reference. This allowed me to change the values of the variables inside the function and have main.c allowed to pass them into other functions by value.

The complexity in parsing the command line arguments came from the dynamic ordering and number of arguments. However, C saves the size of the command line argument array and that an indicator preceded the specification for each setting, so I could use that information instead. For example, "-x" always. The possible arguments and their identifiers were as follows:

- Trigger Mode -m
- Trigger Level -t
- Trigger Slope -s
- Sample Rate -r
- Trigger Channel -c
- X scale -x
- Y scale -y

I began this by separating out all the arguments by input type. Trigger Mode and Trigger Slope both accepted one of two acceptable strings. Trigger Level accepted any integer inside a rage. Sample Rate, Trigger Channel, X scale, and Y scale accepted integers from a few acceptable values. All the outputs to this function were integers, because all the inputs were either integers or a discrete set of possibilities. I initialized all these integers to -1 because -1 is not a valid output for any setting. After I parsed the command line input, replaced -1 with the default value.

As a general approach I parsed the entire argument array by looking at only one setting at a time. I iterated through the array looking for one identifier at a time. I did this by looking at each argument and using strcmp on it and the identifier "-x" for instance. If strcmp returned anything other than zero, I moved onto the next. If it returned zero, I checked that the next argument was something other than NULL, and only then try to parse the argument.

For inputs where I expected one of two strings, I used strcmp on the argument for both valid strings. If the string identified as one of the expected strings, I set a variable equal to a #define in the scope.h file. If the input differed from both expected strings, I printed an error message. For example, trigger slope has two valid argument, either "pos" or "neg". If the input parser found "-s" in the command argument array, it used strcmp on the next argument and both "pos" and "neg". If strcmp found it to be equal to say "pos", it set the variable equal "POS", defined to be zero in the .h file. If the argument differed from both "pos" and "neg" it printed an error message.

For inputs that accepted one of several numbers, I used atoi on the next argument. If it returned zero, I knew that the user tried to pass something other than an integer, so I printed an error message. Otherwise, I passed the result of atoi into a switch statement where each case contained one of the acceptable values. If any of the cases caught, that value set to the output value. If default caught, then I printed an error message.

Trigger Level could accept integer inside a range. It also accepted "0" value so atoi couldn't distinguiqh between a valid a non-valid input. I used strmcmp on the argument and "0" to check zero. This test singled out zero and allowed me to distinguish between a legitimate zero and an invalid string. I printed an error message if the value was outside the acceptable range. Only if the output satisfied all these conditions would I save the value into tLevel.

Once I wrote this for all settings, I needed to fix some of the referencing and dereferencing to make sure I had all the pointer logic correct. Once I did this, everything worked as intended and I never needed to open it again for changes. Theoretically, if a user inputted two conflicting settings, the parser found it acceptable and only used the first one. I could fix this by iterating through the whole array and print an

error message if it found a duplicate identifier. I chose to move onto other parts of the project over implementing this additional functionality.

USBCOMM.C

Unlike cmdargs.c, I needed to come back to usbcomm.c multiple times in order to get it working the way I wanted. I began by copying and pasting the initialization code I used in lab 5 into a function that took in no arguments and outputted the USB handle and a variable representing success or failure.

GetData took much more work to get functional. At first, this function only took in two different byte-arrays, two return values, and one "bytes received" value. All of these were outputs that the function passed by reference to store information. In the first version, this function simply transferred the most recent 64 bytes received by each endpoint into both arrays. This only allowed me to graph 64 bytes. This tested the graphics well, but I soon needed to change it to graph more than 64 data points.

I tried to increase the number of relevant data points by iterating through the data array in reverse. For each element with an index less than 64, I set this element equal to the array element preceding it by 64. This effectively shifts every array element down the buffer by 64 spots. Then I copied the data pulled by the USB into the first 64 bytes. This functioned like a FIFO buffer for each block of 64 bytes. Unfortunately, this resulted in numerous breaks in the data. I theorized that this came from the time it took to shift the data in between calls. I later found this assumption to be wrong.

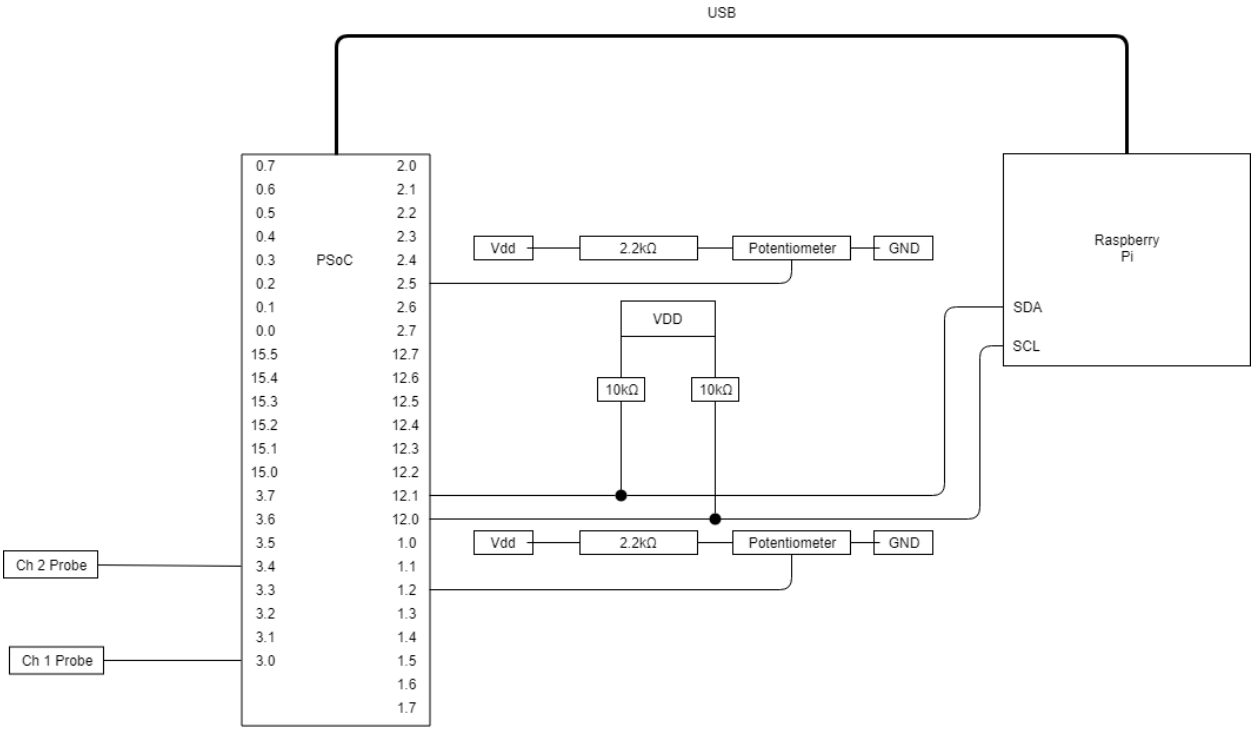
Instead of shifting the data itself, I tried to shift where the function moved the data. I rewrote the function to copy the USB data into the array starting at an index specified by a static variable declared only in usbcomm.c. After the transfer, I incremented the index variable by 64 modulo the size of the array, declared in a #define in the .h file. This didn't solve the problem.

After some discussion with the tutors and TAs, I saw two issues. First, I wrote the PSoC code to transfer data in the interrupt instead of the main loop (described in PSoC main.c). This caused the PSoC to block. I'm not sure why this affected the outcome, but it became a point of concern for the tutors and TAs helping me, so I changed it. Second, the time it took to graph the data after every transfer caused discontinuities in the data from one call to the next. To fix this, I implemented a for loop to transfer several blocks of data on every call to completely refresh the data in between calls. This removed all the discontinuities except at the first block of data. It took some discussion with the tutors to find why the first block was still discontinuous. I found that the USB EP on the PSoC attempted to transfer data continuously whether the Pi was ready to accept it or not. If the Pi didn't accept the data, the PSoC held the data in place and blocked all further data. As a result, the initial block of data transferred could be data from upwards of a second ago, but the next block would still be fresh, causing a time gap of multiple milliseconds. When main called getData, it received the first block of "stale" data then transferred the next several blocks of "fresh" data recently pulled from the ADCs. Removing the stale data removed the discontinuities because it also removed the time gap in between the stale and fresh data. I removed the stale data by first transferring a 64-byte block into a garbage array, then transferring the next several blocks into the array I graphed.

The next and final major change I made simply optimized the function for speed. Since fixing the stale data problem, I increased the size of the data arrays to allow me to plot longer periods of time (larger values of x scale). This, combined with higher sample rates, required more space in the array to store data. 100 milliseconds per division and 100 kSpS created a worst case scenario with a minimum of 45,000 data points, plus an extra 1,000 so the trigger can offset the plot if it needs to, plus the next higher multiple of 64 so the index variable stays aligned with the start of the array. I chose 51200 as the maximum array size

because it obviously satisfied all the criteria for an error-free plot, but this size significantly slowed down the program. I realized that for most values of x scale, 5120 spots sufficed, but I didn't want to dynamically declare the array size. Instead, I always declared the array as 51200 bytes, but I passed in a variable called size set to either 51200 or 5120. Usbcomm.c then only transfered 51200 bytes if it needed to and transferred 5120 bytes otherwise. This sped up the oscilloscope from about half a frame per second, to one frame per second. I didn't see any other ways to optimize the function that were non-jeopardizing to the rest of the program, so I left it for the final version.

External Hardware



I didn't have many choices on the design of my external hardware, but I did make some. The GPIO pins configure the I2C pins in open drain configuration, so I had to place a pull up resistor on each to Vdd, which the Raspberry Pi set to 3.3 volts for this project. Professor Varma recommended 10 kΩ as the resistor value, and I didn't see any need to go against his recommendation. The potentiometers behaved strangely at voltages close to 3.3, so I placed a 2.2 kΩ resistor between Vdd and the pot.

MAIN.C

I worked on my main in two parts, the initialization and the main loop. The initialization required few rewrites, but the main loop took significantly more work. I placed all the hardware initialization, command line argument parsing, finding the number of data points to plot (see data.c) and the graph initialization in the initialization because all these values never changed when calculated once.

My main loop served as my testing field until I had all parts of the project working. At first, I just had it print out all the settings repeatedly. In the next revision it used usbcomm.c to print out the data the Pi received. I decided to include the USB error checking in the main loop because I wanted an error as critical as the USB at the top level and immediately kill the program. This allowed me to kill the program at will if using Ctrl + C didn't always kill the program. When Ctrl + C failed, I had to unplug the Pi to get the graph off the monitor. This made pulling the USB cable to kill the program an indispensable feature

because it gave me a failsafe to clear the screen if Ctrl + C failed without resetting the Pi. The final revision included full graphing capability and a trigger mode.

I found that the trigger function I built in data.c caused some volatility in the graph. I deduced that this resulted from different values of the trigger offset on each measurement. I tried to solve this by measuring the trigger multiple times and averaging out over that in the main loop. I theorized that having a larger sample data resulted in fewer extremes when I went to plot the data. I'm not sure if this helped, but I kept it in because it couldn't hurt and theoretically helped. Pulling the I2C simply required two calls to wiringPiI2CRead because these gave me the most recent values transmitted by the PSoC to use as an offset.

DATA.C

Data.c served for me as a miscellaneous file for all functions that didn't belong somewhere else. For this reason, I included in data.c all code for finding trigger and the code to find the number of plot points to graph.

To find the trigger offset, I first needed to find the trigger slope. I did this by passing in an entire array, but only looking at the first and second element. I considered the slope negative if the weighted average of the first element was greater than the second. I considered the slope positive if the first element was less than or equal to the second.

To find the trigger, I built a function called findTrig which passed in the entire data array, the desired trigger level, and desired trigger slope. The function returned an integer that points to the first index of the data array that satisfied both the trigger level and trigger slope. Then in the main loop, instead of passing in the entire data array, I passed in the address of the index pointed to by findTrig. From there, graphData only graphed starting at the index pointed to by findTrig. To ensure that graphData never tried to graph more data than the amount left in the array, I forced usbcomm to record 1000 extra data points, and forced findTrig to return zero if the index satisfying the condition exceeded 1000, thus making the graph run in free mode.

I found the first index satisfying the trigger slope and level conditions by iterating through the entire array and analyzing each element in comparison to the element that came after. If the user desired negative slope, the current value had to be greater than the trigger AND next value had to be less than the trigger AND findSlope had to be negative. If the user desired positive slope the current value had to be less than the trigger AND the next value had to be greater than the trigger AND the slope had to be positive. The function then returned the number of iterations it had to make in order to satisfy this.

To find the number of data points to plot, I needed to know the samples per second and the x scale. The x scale defined how much time worth of data the graph plotted, and the samples per second defined how many data points created every second. Since we measure x scale in units of seconds per division, and sample rate in kilo samples per second, multiplying the two gave me the kilo samples per division. I multiplied this by 10 divisions per graph to give me the kilosamples per graph. I decided to implement this using a switch statement in case I needed to scale the values to account for unexpected errors. I turned out not need any scaling other than what the math expected, but I left it anyway.

GRAPHICS.C

Graphics.c had the most code and thereby most complexity. I eventually broke it down into several smaller functions, but I still had some issues figuring out what did what. Most functions operated on a struct called datapoint, which encoded both the x and y coordinates for each point. I generated these data

points in a function called processSamples which took in the number of plots to point, the data array, the margin size, the y scaling, and the dimensions of the screen. It created a set of screen coordinates by spacing all the x coordinates evenly throughout the plottable screen. The function determined the y coordinates by scaling the 8-bit value by a scalar determined via trial and error for each value of y scale. I also added the offset determined by the I2C which appeared to move the wave up and down on the screen.

To plot the wave, I originally plotted a line for each gap between the data points, but after a recommendation from a tutor, I used the function polyline to draw all the lines at once, which sped up the program substantially. Polyline took in two vectors of the same length, so I used a for loop to unpackage all the data points from the array of structs holding them. Once I implemented this function, I noticed a significant improvement in performance.

RESULTS AND CONCLUSION

Once I built all the modules, and configured them properly, the oscilloscope plotted data consistently and accurately. I could only get the graph to update about once every second, but I noticed few errors. Triggering worked well enough on the wave that caused the trigger, but the second wave didn't work nearly as well. It didn't appear to run in free mode, but the wave not triggering wasn't nearly as stable as the one triggering.

I also noticed a few glitches where the wave displayed a jump in the data. I never found the source of this, but it occurred around once every ten seconds, so I didn't invest too much time into fixing it (considering I had almost run out of time), but I considered the volatile trigger function as the most likely cause of the problem.

This project taught me several new concepts on the Pi. These included skills such as the OpenVG graphics library and I2C. I learned much more about how to dynamically transfer data over USB reliably, which wasn't a huge focus in lab 5. It became a much larger focus in this project because if I didn't quality control my data, it came out with lots of breaks and didn't represent the wave well.

If I were to do this project again, I would start with the USB and get data transferred immediately. I would then invest more energy into getting quality data transferred and less energy into getting basic functionality as soon as possible. I think I made the right choice to prioritize basic functionality since I didn't have a good idea what I was doing with the project, but I would be far less concerned about this on a second try.

I would also make a slightly more robust testing harness. I used a few #define and #ifdef statements to isolate parts of the project and test them separately, but few of these lasted more than a few hours, and I don't think I effectively maintained the ones I made. My USB test exemplifies this because it simply printed out all 64 bytes received over USB. Printing the data easily yet effectively tested the USB cable, but it didn't test data recording and processing at all or loan itself to a modification that could.

Future project ideas I had relating to this include building a logic analyzer, or an arbitrary signal generator.