Project Report - Dual-channel Digital Oscilloscope

Brandon Hayame

December 2019

1 Introduction

In the final project of the quarter, we are tasked with designing and building a dual-channel oscilloscope using the PSoC 5 as a slave device and Raspberry Pi as the master device. The application consists of the PSoC, which is used to sample the analog channel data, convert it to digital, and send it in packets via USB to the Raspberry Pi, as well as control the waveforms vertically via two potentiometers which communicate via I2C connection. The Raspberry Pi is used to process parameter commands given to the program and send them to the PSoC, and handle the data processing and graphics display, as well as trigger detection.

The mode (free or trigger), trigger channel, trigger level, trigger slope, sample rate, x scale, and y scale are all parameters that are determined by the user at setup and processed by the Raspberry Pi. Each of these parameters have default argument values and may be left blank, in which case the default value is used.

2 PSoC Slave Application

The PSoC side of the application was the most logical place to begin work on this project, and was where I began. Before any work could be done on the software side of the PSoC, some modifications had to be made to the board itself to allow it to work alongside the Raspberry Pi. To configure the PSoC with a VDDIO of 3.3 V rather than 5 V, the 0 Ohm resistor R15 was desoldered from the board and VDDIO pin was connected to the Raspberry Pi's 3.3V pin. These changes allow the I2C link between the two to function properly.

The PSoC uses two 8-bit SAR ADC's to sample the data from the two channels of the oscilloscope, as they possess the highest sampling rates. The ADCs are sampled 600,000 times per second, and the values from these samples are loaded via DMA transfer into two arrays, using the ping-pong buffering method in Labs 4 and 5. Once one of these arrays fills and the TD of the DMA transfer is complete, an interrupt triggers that switches the current array being emptied to the other array. The array that is determined by this interrupt is then polled in the main loop to be transferred by the USBFS component. This ping-pong buffer occurs twice, for each channel of the oscilloscope.

The USBFS component is set up with two endpoints, 1 and 2, corresponding to Channel 1 and 2 of the oscilloscope and ADC. The component transfers the 64-byte blocks to the Raspberry Pi, to be processed with the `libusb` library later.

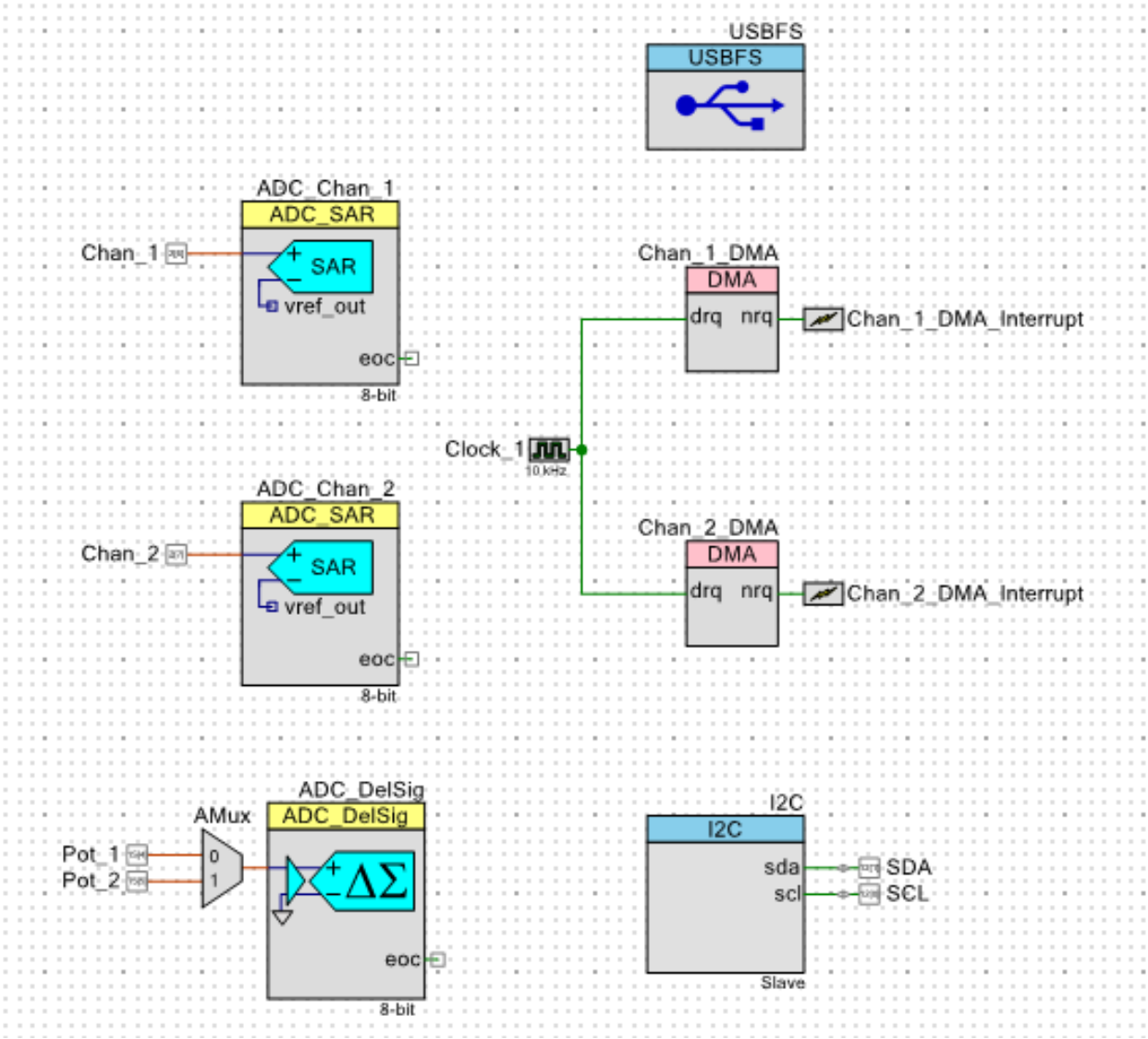

Figure 1: TopDesign for the PSoC Slave application.

3 I2C Communication

The PSoC application also consists of a third Delta-Sigma ADC, which is responsible for sampling data from both of the potentiometers for the purpose of shifting the waveforms left and right on the Raspberry Pi display. This is done with one ADC for both potentiometers, by using the Analog Mux component to multiplex the pin inputs of the potentiometers. The bytes from the ADCs are sent via the I2C slave component, whose SDA and SCL outputs are connected to pins of the PSoC and then wired to the SDA and SCL inputs of the Raspberry Pi. Two pull-up resistors connected to 3.3 V are also connected to the wires to ensure they are able to change state.

Within the main loop of code, there exist two `char` variables for I2C communication. The `command_reg` buffer is one byte wide, and contains a byte pertaining to a parameter of the oscilloscope to be set by the PSoC, which will be covered later. There is also a two-byte `read_buffer`, which is responsible for sending the data from the potentiometers. On every iteration of the main loop, if the I2C slave component is free to send or receive, it sends the byte from the ADC using the first potentiometer, then the `AMux_FastSelect()` function is used to switch to the other potentiometer and the next byte is sent in `read_buffer`. The same occurs for receiving a byte into `command_reg` from the Raspberry Pi. On the Raspberry Pi side of the application, these bytes are received or transferred via the `<wiringPiI2C.h>` library.

To receive data from the Raspberry Pi, the PSoC side of the application has pre-defined byte codes pertaining to whatever action the Raspberry Pi requests. For example, to change the sampling rate of the PSoC, the Pi sends a byte whose value is determined by a define to mean the sampling rate to be changed to, and both sides of the application have this defined. On receiving the byte, the PSoC checks against these byte codes to determine the action to be done.

4 Command Argument Parsing

When the Raspberry Pi side of the application is run, there are optional command line arguments that may be entered by the user to control various parameters of the oscilloscope. These commands are parsed and interrupted by the `cmdParse()` function in `cmdargs.c`. This function is then called at the start of the main function of the program.

The `cmdParse()` function works by using the `getopt()` function included in the `unistd.h` library. This function takes the argument count (`argc`) and argument vector (`argv[]`) from the main function as arguments, as well as an `OPTION_LIST` string, consisting of all of the possible options we wish to check for, followed by colons. The colons let the function know that any option that appears in the command line interface by the user must be followed with an argument. The `getopt()` function and the following executions run in a while loop until `getopt()` no longer equals -1, signalling the end of parsing.

Within the parsing loop, there is a switch statement corresponding to the return value of `getopt()`. Depending on the option chosen, the argument of that option is saved into a defined `scopeParameters` struct. This struct contains integer values pertaining to all of the possible parameters, and the option argument of each option is stored in its corresponding

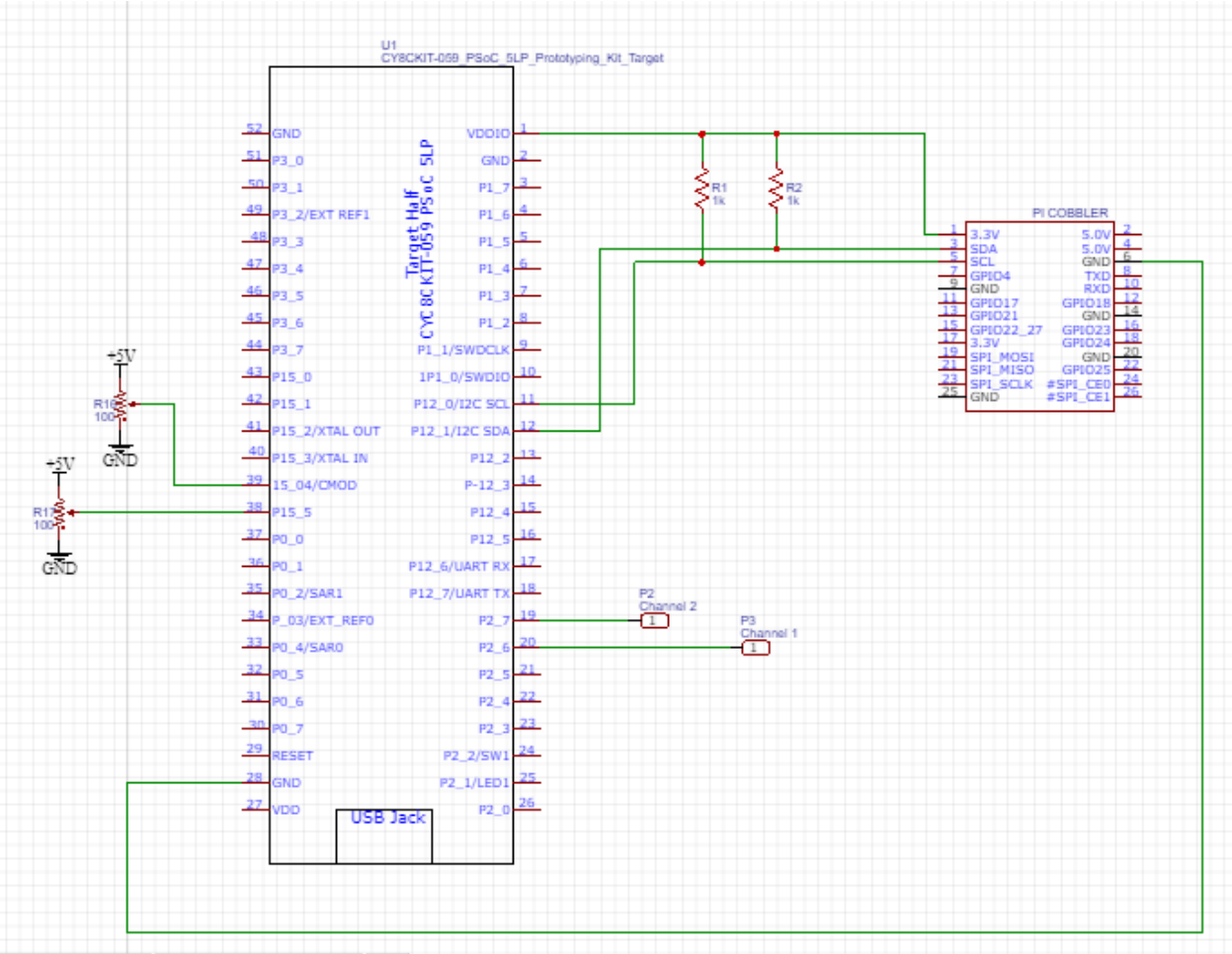

Figure 2: Hardware Schematic for the PSoC and Raspberry Pi, showing the I2C connection with pull-up resistors and potentiometer setup. Note that the two devices are also connected via USB. Made with EasyEDA

parameter entry. For entries such as `mode` and `triggerSlope`, a string i s stored instead pertaining to the mode or slope. There is also input checking to ensure that each argument is within the legal range defined in the lab manual, as well as error messages unique to each check. Lastly, the default values of the struct are assigned before `getopt()` is called so any uncalled arguments retain their default value.

This struct is defined within `scope.h` and an instance is declared and used within the main function. At the start of the main function, the `cmdParse()` function is called with an instance of the struct as an argument, as well as `argc` and `argv[]`. When the parsing has completed, the instance of this struct will contain all of the parameters the user has entered, as well as the default parameters for any the user has not entered. The values of this struct are then sent to their according functions later in the main loop.

5 USB Communication

The `usbcomm.c` library was created to facilitate USB setup and transfers within the main function into single functions. There are two functions within this file; `USB_Start()` and `USB_GetBlock()`. `USB_Start()` is responsible for initializing and configuring the USB link in the main function and `USB_GetBlock()` performs a 64-byte bulk transfer and stores it in an array specified by one of the arguments.

Within `USB_Start()`, the `libusb` library is initialized, and the USB device is opened and reset. The configuration of the USB device is set and the interface is claimed. This is all in line with how the USB would be initialized in the USB example code from Lab 5.

`USB_GetBlock()` calls the `libusb_bulk_transfer` function depending on which channel number it is given as an argument; it is capable of reading from both endpoints. If no errors occur, the function returns the array of bytes into an array that has also been specified as an argument.

Both of the functions take a double pointer to a `libusb_device_handle` as an argument. This allows them to communicate with the device properly as the functions are called. This, however, led to a lot of issues debugging, as it was difficult to keep track of the pointers within the main function. While debugging, I also ran into an issue with the USB halting at runtime and not transferring any more bytes. This problem was fixed by introducing a 1 ms timeout to the `libusb_bulk_transfer` function as it is called. This has the unwanted side effect of `libusb_bulk_transfer` returning an error code of -7 `(LIBUSB_ERROR_TIMEOUT)` all of the time, but the USB communication now works regardless of this.

6 Trigger Detection

To detect the triggers from a waveform, I defined and used two functions, `freeSweep()` and `triggerSweep()`. These functions are used according to which mode the oscilloscope is in, free or trigger.

The purpose of both of these functions is to populate a 100,000 byte array with the number of samples needed to fill one screen through USB bulk transfers, based on the oscilloscope mode. The functions take as arguments the `libusb_device_handle`, the channel

number that the data corresponds to, and the `samples_per_screen` variable, as well as an array through which the outputted samples will be placed into. For `triggerSweep()`, the trigger channel, trigger value, and trigger slope are also taken as arguments. Within both of these, the `USB_GetBlock()` function is called in a loop until the `samples_per_screen` limit is reached. These bytes received are used to fill the given array over and over. This how the `freeSweep()` function works, without any trigger detection.

The `triggerSweep()` function works similarly, with some code to detect a trigger. It takes in the trigger value in mV, and converts it back to the 8-bit value that corresponds to this value from the ADC. On each call of `USB_GetBlock()`, if any of the received values are equal to this value, and the other trigger conditions are met, then the array that holds the samples is filled starting at this point. These trigger conditions consist that the last sample is either above or below the sample, depending on the trigger slope, and that the next sample after the trigger sample is also above or below.

7 Graphics Library

The graphics rendering on the Raspberry Pi is done through the `openVG` library. These functions are demonstrated in the `scopedemo.c` file. Of the functions included, the ones needed for oscilloscope display are the `grid()`, `drawBackground()`, `printScaleSettings()`, `processSamples()`, and `plotWave()` functions. `grid()` prints the horizontal and vertical gridlines, `drawBackground()` displays the background and margins, `printScaleSettings()` prints the yscale and xscale values in the top right corner of the screen, `processSamples()` converts an array of unmodified values from the USB bulk transfer to a `data_point` array that has been scaled to the size of the screen, and `plotWave()` plots the `data_point` array using connecting lines and handles the offset due to the potentiometers.

I wrote my own functions by first studying the functions in the example and rewriting them one-by-one. The method through which these functions operate are similar to the ones in `scopedemo.c`, with the exception of a few changes. For instance in some lines of some functions, scaling of the waveforms was changed to better reflect the size of the screen. There is now a `samples_per_screen` variable in the main code that is calculated by determining the sample rate, x scale, and width of the screen to determine the number of samples that are needed to fill the screen. This value is used in many of these functions. Things such as font, font size, and colors were kept the same between the two programs because I did not know how to properly change these into a color I wanted.

For whatever reason, the `waituntil()` function used to escape from the graphics window would not work on my device, so I would exit my program using `Ctrl+C`.

8 Makefile and Scope

The makefile used to generate the executable is fairly similar to the example one posted on Canvas. Within my makefile, every file is compiled using the line

```
gcc -c FILE.c -I /opt/vc/include -I /usr/include -I/usr/include/libusb-1.0 -lusb-1.0
```

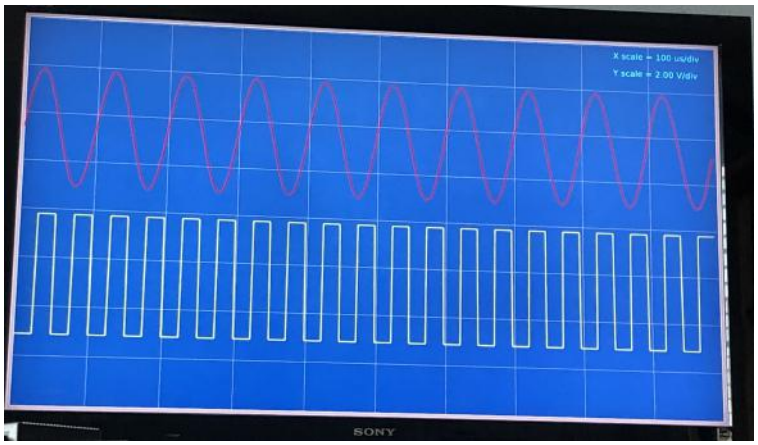

Figure 3: Example of oscilloscope display when the main program is executed.

The reason that libusb must be included in every compile is because of how my `scope.h` file is set up. Within this file, the struct and function definitions used for the other files are defined such that every file is able to use them; this is necessary for parts such as command argument parsing, where the struct must be defined globally, and using the `VGfloat` structure.

9 Main loop

The main function works by calling all of the other functions established in the other sections. It first creates variables to store parameters of the display such as width, height, margin size, and number of divisions, as well as two 100,000-byte arrays for holding the screen samples for Channels 1 and 2. These arrays may not necessarily hold 100,000 bytes; rather they are set up to accept `samples_per_screen` number of bytes, with 100,000 being the max possible value of this. Variables that control the color of certain channels or text color are also defined.

From here, the `cmdParse()` function is called and the results are stored in an instance of a `scopeParameters` struct called `userParameters`. The I2C and USB devices are then set up and begun, through the `wiringPiI2CSetup()` and `USB_Start()` functions. The `samples_per_screen` variable is also calculated by multiplying `userParameters.xscale` by the number of x divisions. The program now enters the main loop.

In the main loop, the background and scale settings are printed with the `drawBackground()` and `printScaleSettings()` functions. From here, the value of `userParameters.mode` is tested to see if it is set to `free` or `trigger`. Depending on which one is the case, the `freeSweep()` or `triggerSweep` function is called on both channels, respectively. This creates an array of size `samples_per_screen` populated by multiple `USB_GetBlock()` calls. The values of the potentiometers are then read through the `wiringPiI2CRead()` functions and stored in two variables. The `processSamples()` function is called on both data arrays to scale them to the size of the screen, and the `plotWave()` function generates lines for both of these waves, as well as their offsets using the values of the potentiometers.

On running the program, I would notice that the graphics would take a long time to initially begin and readjust. Turning the potentiometers would result in a change in the waveform after about 10 to 15 seconds. Also, sometimes the program would refuse to close, even after `Ctrl+C` is used. This would require me to reboot the Pi entirely. I suspect that this is just due to the sheer amount of CPU power being used to generate the graphics, and the command not registering through to the terminal because of this. Sometimes, the command would work, but only after several minutes of waiting and several different attempts at closing. Most of the time, however, this wasn't really a problem and the program would close.

10 Conclusion

All in all, this final project was a great way to apply all of the skills we learned about microcontrollers, the PSoC, and Raspberry Pi over the last 10 weeks. The debugging involved in this lab was much more straightforward than in other labs, mostly due to the freedom I had in designing my project. This allowed me to come up with solutions to a problem that I would not normally implement due to design constraints. I was also able to learn about some very important techniques that I will definitely use for projects in the future, such as makefiles, command argument parsing, and graphics libraries. This was the first project of its kind and size that I had ever taken up; before this most class projects were not on the level of complexity as this one. Although I was not able to finish the entire project at checkoff time, I am very satisfied and proud of the work that I was able to do, and how far I was able to get.