Michael Peters
ESE 5190
Lab 2

Introduction and Overview:

The purpose of this lab is to setup the software needed to build runnable programs in C and SDK for the Adafruit qt py rp2040 chip. The first portion of this lab focused on downloading all the necessary software and configuring it for our unique computer. This process was documented and submitted previously on Monday October 10th in a separate document. After getting all the software properly setup a few basic examples were built and compiled to test the system. These examples included the hello_usb program and the ws2812 program. The hello_usb code consisted of simply printing a message on repeat to the serial console using the rp2040. All of the necessary files were already available in the pico_examples folder so getting this to work was a matter of configuring the Visual Studio code setup correctly.

The second example, ws2812, refers to the LED that is built into the microcontroller and involved simply trying to make the LED blink. Again most of the code was provided in the pico_examples folder however one small change needed to be made. The pin that powers the LED was off by default so a few lines of code had to be added using the “gpio_init”, “gpio_set_dir”, and “gpio_put” commands in the C program. After adding these few lines the rest of the program was already configured to blink the LED in four different patterns.

The final portion of the lab involved combining the two codes into one. This basically means having a program that can blink the LED every time a message is printed to the serial console. There are many ways to do this but the easiest I found was to use the prebuilt ws2812 code from the other example and edit it slightly. First thing was to comment out or remove most of the code that was already there for the patterns. It was unnecessary as all we needed was the LED to blink. Using the already established put_pixel command inside the main loop you could make the LED blink much simpler. After that it was just adding a printf command to the loop and enabling USB access. To enable USB access a few lines of code had to be copied from the other hello_usb CMakeLists file. This file is basically used to describe how to build the final program that runs on the chip and will be discussed in more depth later. After making these changes the program could be built, copied to the chip, and it run as expected.

Brief responses to the reading questions in 3.2.

1. Why is bit-banging impractical on your laptop, despite it having a much faster processor than the RP2040?

After reading through the documentation provided in the lab, it discusses how “bit-banging” is impractical. Bit-banging on the chip is impractical because the processor wasn’t designed with this as the main purpose. On a laptop, while it is very easy, it is again not practical because of how complicated modern laptops have gotten. There are many more layers of code that are working together to get a program to function properly. Additionally, the processor will be occupied doing the bit-banging and interrupting it can cause all kinds of problems in timing which can be very detrimental to the overall *program. If the system clock is much faster than the speed the data is coming in at then the processor will just end up spending most of the time waiting for the next instruction.*

2. **What are some cases where directly using the GPIO might be a better choice than using the PIO hardware?**

PIO state machines are very useful because they can get more things done per cycle when programmed properly compared to other GPIO pins. However, PIO state machines can't run general purpose software meaning that there are drawbacks. They are able to access the GPIO's but in a case where general purpose software needs to be run a GPIO would be a better choice. Each one has it's unique uses depending on the situation.

3. How do you get data into a PIO state machine?

Each PIO block has 32-slot instruction memory which the program gets loaded into. This memory is then used so the state machines can read it. Once it's loaded an available state machine gets told to run the program and the data gets sent to it. Both getting data in and out uses what's called a first in first out (FIFO) queue. The data goes through the state machines Tx FIFO into the output shift register.

4. How do you get data out of a PIO state machine?

To get data out, the input shift register of the state machine is used to send data into the Rx FIFO. The SM instructions are used to move the information around depending on where you want it to go. These instructions can move data between scratch registers, push it to the Rx FIFO etc.

5. How do you program a PIO state machine?

In order to program a PIO state machine you must load the program into the state machine's 32-slot instruction memory to run the program. There are 9 basic commands that are used to control what the PIO state machine does, each with a unique use. For example there is a command to push data to the FIFO, pull data from the FIFO, and jump between certain instructions based on a bit value.

6. In the example, which low-level C SDK function is directly responsible for telling the PIO to set the LED to a new color? How is this function accessed from the main "application" code?

The C SDK function that tells the PIO to set the LED color in the ws2812 example is pio_sm_put_blocking. This function is accessed through the put_pixel command, it writes a word of data to the state machine TX FIFO which is where the data gets sent to enter the state machine.

7. What role does the pioasm "assembler" play in the example, and how does this interact with CMake?

The pioasm is as the name suggests the Pio assembler. The program is known as an assembler because it translates assembly code into binary. It processes the PIO input assembly text file and creates a unique function for each program. It then outputs a configuration for the PIO state machine and makes the assembled programs ready for use. CMake uses all of this information as well as the other c codes and compiles everything together into one runnable uf2 file. The CMake files describe how all these files are built into that runnable uf2 file.

Discussion:

Looking at the files in my report, the first is the annotated code. Again this is the c code and the pio code which is converted into a header file and used in the other c code file. Both were annotated based on what I thought each line/function was doing when I first began the lab. The second file attached is the hand modeled state of the machine as it pulls in the first bit from the FIFO. This was done based on the loop of code in the pio file annotated above and trying to map it out based on the clock cycles.

Next is the excel spreadsheet which contains a few different tabs. The first tab is the 3.4 initial state of the machine when the packet is in the FIFO but has not been pulled into the output shift register. This includes the list of relevant registers in the lab, their address, offset, names, and values based on what I thought was happening initially. There are also notes to try and describe my thinking and why I put each value as I did. The second tab in the excel file contains the 3.6 Packet Transmission Flow chart. This chart was based on the hand drawn model and is meant to show the flow of each bit into the state machine and to the LED. Additionally, there are a few extra rows to try and show the value of the LED voltage, luminosity, pin voltages, and an arbitrary sensor reading the color values. These were again done to represent how I viewed the system operates as it flows through that same loop in the pio file mentioned previously. The third and final tab in the excel document contains just a pictorial representation of this packet flow from the FIFO, through the state machine, to the LED, and the reading of the sensor. I simplified it a lot but again it was still quite difficult to make as excel isn't primarily meant for this type of application. A slightly cleaner version of this diagram can also be seen below in figure 1. Lastly, a short video of my blinking LED + sending a message in PuTTy is also uploaded because I didn't get it to work until after the lab session on Friday.

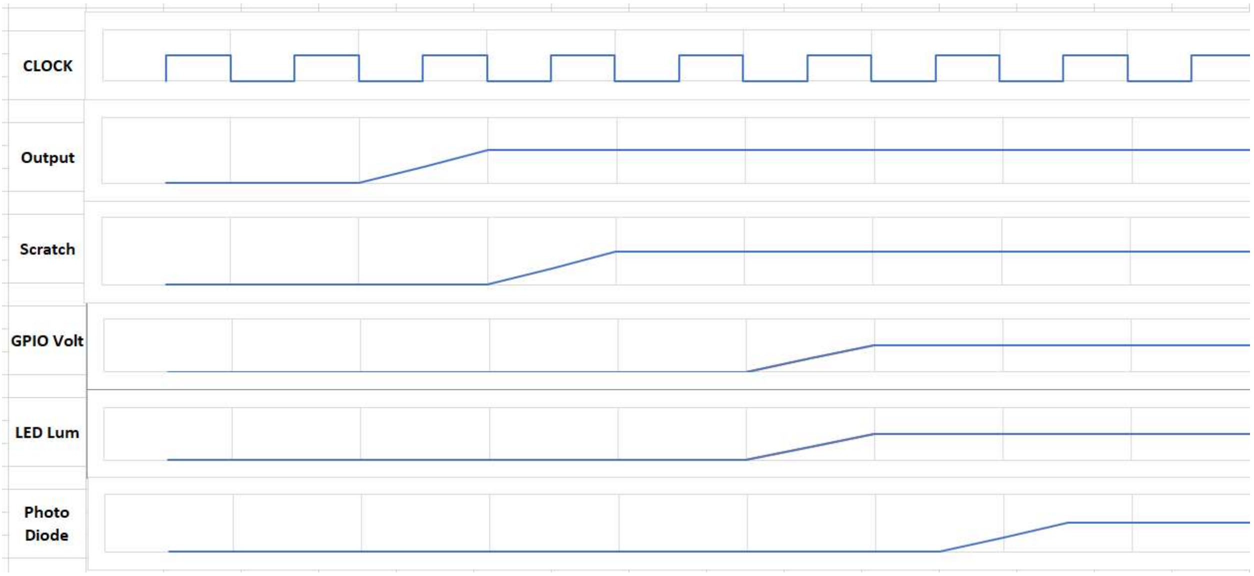

Figure 1: LED Timing Diagram

This is my first time dealing with this sort of information and trying to model a system like this. With that being said, I'm sure I made many mistakes and didn't entirely represent the flow of information how it actually works despite being able to program it properly. The first thing I was unsure about was whether the OUT command pulls the full 32 bit string from the FIFO into the X-Scratch register or if it does it one bit at a time. I figured since the process was to first model 1 bit by hand that meant it pulls the info 1 bit at a time. Next when modeling I chose to shorten the amount of rows that would be there during the clock cycles where it is executing a stall command. I wasn't sure if the stall commands meant the whole system stalls since we are setting a pin to high using the "Side" command and want it to stay on for a bit. Or if the stall command keeps the loop running but only stalls the values being output. Based on the timing of the code in the loop I assumed the first. Lastly, I had trouble finding the value of the clock cycle divider. I found in the code many places where it's called out but struggled to find where it was defined. So I used a value of what I thought it was.

The tools that were used to construct the different models by hand were not ideal. Modeling by hand was a bit annoying and didn't provide a clean look like the excel models did. On the other hand, the excel models had so much information in them that it was difficult to keep track of it all and make sure each component was acting how it would in real time. The interface to do this in excel was not great considering that's not really what excel is primarily designed to do. What made it easier though was that I have years of experience in Excel and didn't have to learn a new program if I had tried to use a different resource online.

Working with paper to annotate code wasn't bad since annotating my hand is much easier than doing it on the computer. Modeling the flow of information in the state machine by hand was annoying however. I'm new to this and doing it by hand meant it was difficult to keep track of the clock cycle and what each component was reading at that specific time.

I had the words being sent broken up into 8 bit sections because that is what I've worked with in the past at work when having to look at binary breakdowns of systems. However, even though I've worked with something similar before, it was still annoying to be because the bits being used by each register were not broken down into convenient sections. Many had 8 bit sections reserved which was convenient but for example SM0_PINCTRL registers had sections of 5 bits and 6 bits. It got slightly confusing when trying to map these and remember which bit refers to what. For the full packet transmission, there is again so much information to include between the clock cycles and the information each component is receiving that I am certain I made many mistakes along the way. I tried to be consistent in what happened in each case depending on if the bit coming out of the FIFO was a 1 or a 0. Then I varied what was happening based on the color that the bit was associated with and how the bits were flipping (1 to 0 vs 0 to 1). Working with spreadsheets definitely made this easier because you could copy and paste most values depending on the bit to make it easier to expand the model to include the entire packet. Doing the entire packet by hand would not have been possible.

This is my first time working with this type of content but I'd imagine there are many other tools that are better than excel or paper for modeling a system like this. If there are other tools besides excel that are meant for these purposes then those are almost certainly better than what we used. Additionally, making the final timing diagram was a bit of a mess so again I'd imagine there are tools and resources online that have this process in mind and make it easier to create a clean looking diagram.