Wilson Au-Yeung

CMPE 121

Anujan Varma

October 15, 2016

Lab Report 1

Introduction:

This lab is comprised of three sections, each of which were focused on teaching us the basics of using and getting familiar with the PSoc-5LP Development kit. In order to familiarize ourselves with the Psoc5, we had to look and study various data sheets carefully to understand the configuration options. In the first section of the lab, we were told to toggle a single LED using 3 different methods. All three of which was to teach us the basics of the GPIO pins so we would be able to manipulate the input/outputs from our C program. In the second section of the lab, based on the analog value of the on board and off board potentiometers, we were told to alter the brightness of a single LED using the PWM and changing its duty cycle. And lastly in the final section of this lab, we were told to measure and output frequencies using the components of the first parts such as the ADC and the PWM, but also use new components such as the counter, timer, and interrupt.

Method:

Part 0: GPIO Pin Toggling

In this section of the lab, we were told to control the state of a LED on the Psoc5LP Board using C programming. We used three different ways to manipulate the output of a GPIO Pin such as the Per-Pin, Component API, and using the Control Register. We first had to add a digital output pin from the component catalog and assign it to Pin that was connected to the LED shown in Figure 1. It would not have a hardware input pin, but instead be controlled by the software implemented in the C programming. The Per-Pin method and Component API set up was almost the same due to it being fully software controlled and writing / clearing the Pin itself

Figure 1: Digital Output Pin

The Control Register was slightly different due to it being a hardware controlled pin. After reading the Control Register datasheet, we learned that we could write into the control register instead of into the pin itself and use the Control Register as a hardware input into the GPIO Pin. (Shown in Figure 2)

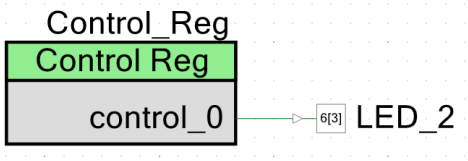
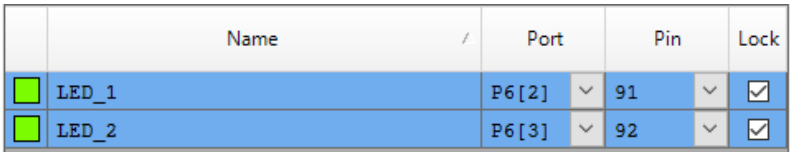


Figure 2: Hardware Controlled Output Pin

I used the Pin 6[2] for the software controlled output and Pin 6[3] for the hardware controlled output so I wouldn't have to change the software/hardware settings of the Pins every time I used this Part of the Lab. (Shown in Figure 3)



	Name	Port	Pin	Lock
	LED_1	P6[2]	91	☑
	LED_2	P6[3]	92	☑

Figure 3: Pin Assignments

Part 1: LED Brightness Control Using PWM

This section on the lab required changing the LED brightness based off the on board and off board potentiometers. Since both potentiometers are analog signals, we had to use the ADC to convert it into digital so we would output the result and also to feed it into the PWM. The PWM was to vary the brightness of the LED depending on what was inputted into the compare value. The LCD was used to output the value in the potentiometer which was useful in debugging. In this part of the lab, we had to use the ADC, PWM, LCD, and clock. (Shown in Figure 4)

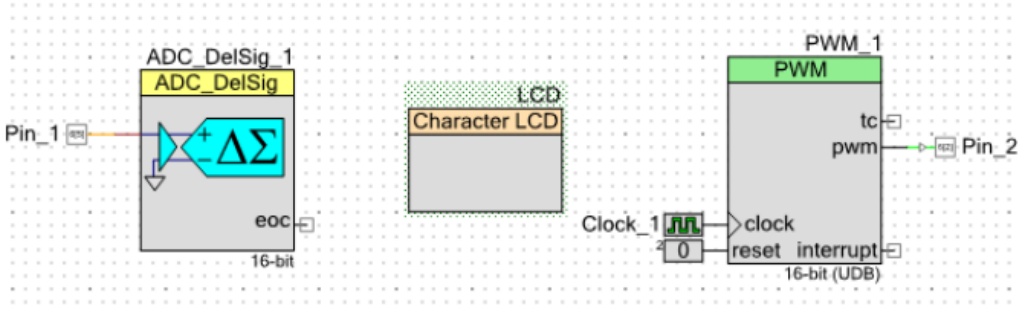


Figure 4: Part 1 Schematic

Part 2: Frequency Meter

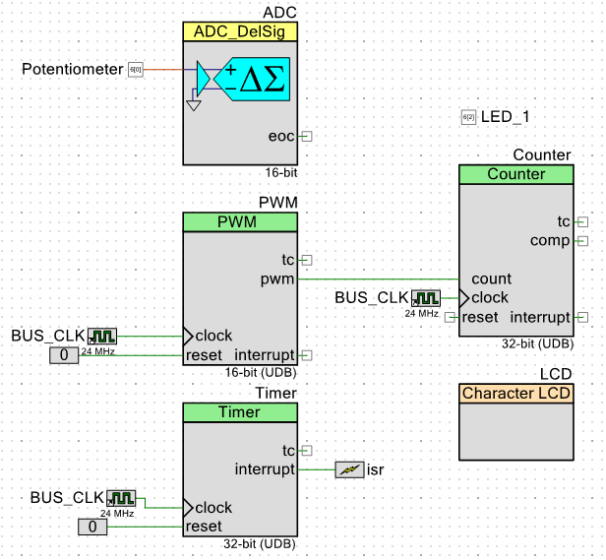


Figure 5: Part 2 Schematic

In this last part of the lab, we were told to generate a frequency that ranged from 1kHz to 12 MHz. In order to check our frequency, we used two different methods to calculate it. Our clock speed for the inputs of the timer, PWM, and Counter was 24MHz so it would only go up to 12MHz since the duty cycle will always be set to half of the PWM pulse.

$$24\text{MHz} / 2 = 12\text{MHz}$$

The ADC's purpose is to convert the Potentiometer's analog signal into a digital one so we could both output it to the LCD for debugging and to input into the PWM. The PWM would send pulses ranging from 1kHz to 12MHz which would either slow down or speed up the counter increment depending on the Potentiometer. We would then divide it by the 2ms or multiply it by

500 because that was the period of the timer. If we divide the number of increments by the period of the timer, we would be able to calculate the frequency.

Procedure

Part 0: GPIO Pin Toggling

Per-Pin API

This method required me to access the GPIO Pins directly. I would use premade functions listed in the API to Set a Pin and another function to clear the pin.

```
CyPins_SetPin(LED_1_0);
CyDelay(1000);
CyPins_ClearPin(LED_1_0);
CyDelay(1000);
```

Component API

This method also required me to access the GPIO Pins directly. This API allowed to use one function to do both setting a pin and clearing a pin. I could have set 1 or a 0 to turn it on and off, but instead I used another function to read its last setting and do the opposite of that.

```
LED_1_Write(!LED_1_Read()
CyDelay(1000);
```

Control Register

This method allowed me to directly write to the control register instead of the GPIO Pins. Then it would send the signal directly to the digital output pin. I created a int variable called count to keep track of it being a negative or positive number. Then it would either turn on the LED on every odd number and off at every even number.

```
Control_Reg_Write(count%2
CyDelay(1000);
count++;
```

Every function listed above does the exact same thing which was to turn the LED on for one second, and then off for one second. The CyDelay() function helped delay it so we would actually see the change that was transpiring. There was small differences between the Per-Pin API and the Component API. The Per-Pin used functions that individually cleared or set a pin whereas the Component API has one function to manually write the value into the Pin. The Control Register meant that the value was written into the register itself and would send the signal to the GPIO Pin.

Part 1: LED Brightness Control Using PWM

We first set the digital value obtained from the potentiometer into a variable. Then we would have to check the bounds to make sure the values wouldn't go over 0 and over 0xFFFF. If the value went under 0, set it to 0. If it went higher than 0xFFFF, set it to that value. If I did not account for overflow, the bits would go straight from 0x0000 to 0xFFFF and vice versa, and that would set the LEDs to on/off instantaneously. (Figure 6)

```
output = ADC_GetResult32();
if(output<0x0000){
    output = 0;
}
if(output > 0xFFFF){
    output = 0xFFFF;
}
```

Figure 6: Top / Bottom Thresholds

```
LCD_Position(0, 0);
LCD_PrintString("Debug: ");
LCD_PrintInt16(output);
PWM_WriteCompare(output*999/0xFFFF);
```

Figure 7: Writing into PWM

Lastly, I scaled the output of the potentiometer to 999 which is the max period inside the PWM by dividing it by 0xFFFF (Max Value of potentiometer) and multiplying it by 999 (Max Value of the PWM. I used the built in function of the PWM which was PWM_WriteCompare() to change the duty cycle to alter the brightness of the LED. (Figure 7)

Part 2: Frequency Meter

In order to make sure we had the right frequency, we were told to calculate it using two different methods. The first method was to divide 24MHz (Clock input) by the PWM at the time which is the expected frequency.

The next method was to divide the number of increments from the counter by the amount of time within the 2ms time frame. As stated before, the potentiometer was determined the frequency for the PWM. The lower the frequency, the slower the counters incremented. The higher it was, the faster the increments were. Once we had the counter value increasing, we had to add an Interrupt Service Routine out of the timer to output every 2ms. After every 2ms, it would go into the Service Routine, read the counter value, and divide it by 2ms (Multiply 500) to get the frequency.

We were told to use two implementations of the interrupts. First was to clear the counter after every calculated frequency. (Figure 8) The other implementation was to keep track of the counter during that time and subtract the new value of the counter from it to get the difference. (Figure 9)

```
    CY_ISR(Interrupt)
{
    isr_count= count;
    count = Counter_ReadCounter();
    new = count - isr_count;
    CxF = new*500;
    isr_flag = TRUE;
    //Timer_STATUS;
    //Counter_STATUS;
    Timer_ReadStatusRegister();
}
```

Figure 8: ISR Implementation 1

```
CY_ISR(Interrupt)
{
    count = Counter_ReadCounter();
    CxF = count*500;

    Counter_WriteCounter(0);
    isr_flag = TRUE;
    //Timer_STATUS;
    //Counter_STATUS;
    Timer_ReadStatusRegister();
}
```

Figure 9: ISR Implementation 2

Questions:

Part 1:

Per-Pin API

Pros: Could safely access individual pins without a performance or memory penalty

Cons: Does not take in account for different components accessing the same pins

Component API

Pros: This method allows access to all the pins with a single function call

Cons: Only works if all the pins are on a single physical port on the device

Control Register

Pros: Allows us to write into the control register instead of the pin directly

Cons: More expensive than Per-Pin and Component

Part 2:

When I read the on-board potentiometer as a 16-bit, it only went up to 0x8000. When I read it as a 32- bit, it went up to 0xFFFF. I would use the 32-Bit

Part 3:

	Frequency (track)	Frequency (clear)
0x1388	0x1391 (.18%)	0x1412 (2.76%)
0x8207	0x81FA (.16%)	0x82dc (.64%)
0x1398B	0x13941 (.09%)	0x13880 (.33%)

The method that required us to keep track of the counter is more reliable and accurate because clearing delays the process which causes more time to pass. If delayed, the counter would increment more than it was supposed to in that 2ms. I realized that the measurements at the low ends aren't very accurate and would fluctuate a lot.

Conclusion:

In this lab, I learned to familiarize myself with the Psoc5 through experimenting with different components and reading the data sheets. I enjoyed this lab due to it telling you exactly what you need to read about to really progress through the lab. I also really appreciated that it gradually built on what you learned previously and you had to really grasp the previous section to really progress on. This lab was a really good refresher for me as it combined some aspects of CE 100, CE 13, and EE 101 in one lab.