Andrew Mikhail

CMPE 121L: Microprocessor System Design Lab

Lab Exercise 1

12 October 2016

Table of Contents:

Introduction:

The purpose of this lab exercise is to familiarize us with using the GPIO pins of the PSoC-5 microcontroller. Within this lab, we are to learn how to monitor and control the inputs and outputs (hardware) using a program (software). Moreover, this lab should teach us about pulse width modulation and its applications in circuits. Furthermore, this lab should give us a review of programming in C, in case we haven't programmed recently.

Part 0: GPIO Pin Toggling

In this section, we were instructed to configure a GPIO pin as an output port and control it from a C program. We were also told to control the state of the output pin using Per-Pin APIs (Application Programming Interface), Component APIs, and using a Control Register. The GPIO pin that we configured was pin P6[2] that was connected to LED3 on the PSoC-5 board. Then using the three methods, we toggled the output pin from a program so that the LED would blink. We implemented each of the methods in the main.c file seen in **Figure 1**.

```
for(;;)
{
    /* Place your application code here. */
    /*PART 0
    Turning on the LED*/
    LED_Write(1); /*Component API Method*/
    CyDelay(500);
    LED_Write(0);
    CyDelay(500);

    CyPins_SetPin(CYREG_PRT6_PC2); /*Per-Pin Method*/
    CyDelay(500);
    CyPins_ClearPin(CYREG_PRT6_PC2);
    CyDelay(500);

    Control_Reg_1_Write(1); /*Control Register Method*/
    CyDelay(500);
    Control_Reg_1_Write(0);
    CyDelay(500);
```

Figure 1: C code from main.c File Depicting How We Toggled the LED

This section was the easiest to implement since all the information required was available in the Application Programming Interface (API) documentation. Thus, we looked up the correct syntax for each method then toggled the LED to blink by writing the pin a digital 1 followed by a digital 0.

Each of the methods for controlling an output pin has its pros and cons. The Per-Pin Method was easy to implement but can result in undefined behavior in the physical pin if

there is another component configured to that pin by software. The Component API Method was by far the best in my opinion. It allows you to specify the output pin you are toggling to using an instance name that you can also use elsewhere in your block diagram. Moreover, Cypress recommended using Component CyPins because they automatically place and route signals for you. Though, the Component API Method lacks usability with other components on a top-level design. The Control Register Method was also simple to implement and allowed for quick pin toggling within the API. However, the downfall of this method is that there must be an initial state described for the output pin.

I would use the Control Register Method if I need to interface my output pin with other components in my design. I would use the Component API Method if I were using multiple GPIO pins in a simple program. Finally, I would use the Per-Pin Method if I were using only one GPIO pin.

Part 1: LED Brightness Control using PWM

In this section, we controlled the brightness of an LED using an onboard potentiometer. We also measured the potentiometer value and displayed it on an LCD screen. For this section, we used a PWM (Pulse-Width Modulator), a Delta Sigma ADC (Analog-to-Digital Converter), a Character LCD, and a Clock to control the brightness of the LED. We needed the Delta Sigma ADC because the potentiometer is an analog device so we had to convert its values to digital measurements before using it. We also needed the LCD display so we can visually see the values of the potentiometer. Moreover, the PWM and the Clock were added so we could control the signal that would be outputted to the LED. Combining all these elements, we implemented the top-level schematic in **Figure 2**.

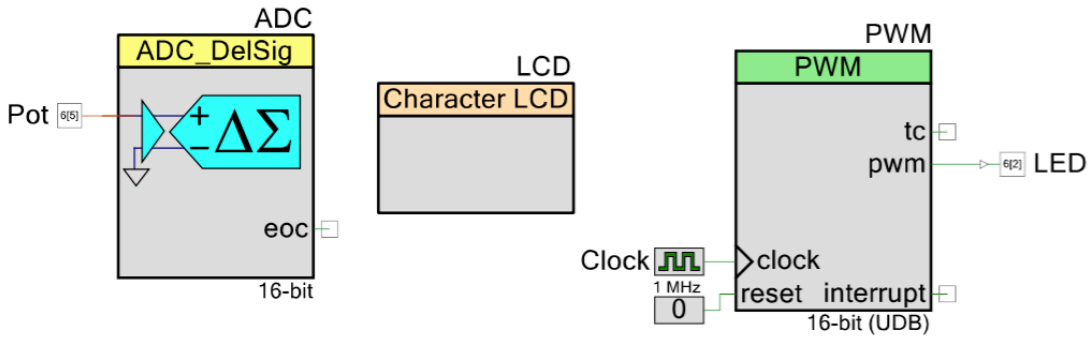

Figure 2: Top-Level Schematic for Controlling LED Brightness Using PWM

More specifically, we configured the ADC to be single-ended with an output resolution of 16 bits. Thus, our output ranged from 0x0000 to 0xFFFF. We also manually set the input range from VDDA (analog voltage) to VSSA (analog ground) since we were using an analog device. Furthermore, we added a 1 MHz clock and set the period of the PWM

signal to 999. Therefore, dividing our PWM period by our clock frequency resulted in a 1 kHz output signal as seen below.

$$\frac{1\ kHz}{1\ MHz} = 1\ kHz$$

We also configured the analog input of the ADC (Pot) to pin P6[5], which was assigned to the onboard potentiometer. Then, we configured the output of the PWM (LED) to LED3 (Pin P6[2] from earlier). To control the brightness of the LED using the potentiometer, we needed to continuously read the ADC output and adjust the duty cycle accordingly. The duty cycle was adjusted by changing the ON interval using the PWM_WriteCompare() command. The write compare command checked if the value of the potentiometer was between 0x0000 and 0xFFFF and wrote a value from 1 to 999 to the PWM. The digital value of the potentiometer was also displayed on the LCD display using the LCD_PrintInt16() command as seen in the main.c code in **Figure 3**.

```
/*PART 1 */
CyDelay(200);
LCD_ClearDisplay();
ans = ADC_Read16(); //Reading Pot value as 16 bit number
if(ans >= 0xFFFF) ans=0xFFFF;
if(ans <= 0x0000) ans=0x0000;
LCD_PrintInt16(ans);

                        Page 1 of 2
```

```
                        main.c
PWM_WriteCompare(ans * 999 / ((1 << 16)));

LCD_Position(1,0);
ans1 = ADC_Read32(); //Reading Pot value as 32 bit number
if(ans1 >= 0xFFFF) ans1=0xFFFF;
if(ans1 <= 0x0000) ans1=0x0000;
LCD_PrintInt32(ans1);
CyDelay(200);
```

Figure 3: C code from main.c File Implementing Part 1: LED Brightness Control

Achieving an accurate potentiometer (pot) reading was the most challenging part of this section. First, we read the potentiometer value as a 16 bit unsigned int type. However, we ran into an issue. The values of the potentiometer were not linear and wrapped around. That is, turning the pot to the leftmost value displayed 0xFFFF then turning it slightly right dropped the measurement to 0x0000. Moreover, turning the pot to the rightmost value displayed F8F4. Thus, we changed the pot reading to 16 bit signed int type, which

resolved the count wrapping issue. However, this left us with another issue. Turning the potentiometer to the very right made our display drop to 0x0000. This was due to the fact that we only had 15 bits to use since the 16^{th} bit is reserved for representing whether the integer is positive or negative. Therefore, we read and displayed the potentiometer as a 32-bit signed int value. This resolved all of our previous issues and displayed correct readings.

We also went on to replace the on board potentiometer with a pot from the parts kit instead and assigned it a GPIO pin. It worked the same as the onboard potentiometer with the exception of a slower linear increase in reading. However, I had trouble holding a reading constant on the display as it changed very often. Thus, I used the onboard pot instead for the check-off.

Part 2: Frequency Meter

In this section, we used the onboard potentiometer to generate a variable frequency signal then we measured that frequency using a timer and a counter. So, based on the value of the potentiometer, the PWM generated a square wave with a frequency in the range of 1 kHz to 12 MHz. We had to adjust the period and the ON interval of the PWM to accomplish this. The top level schematic that implemented this function can be seen in **Figure 4**.

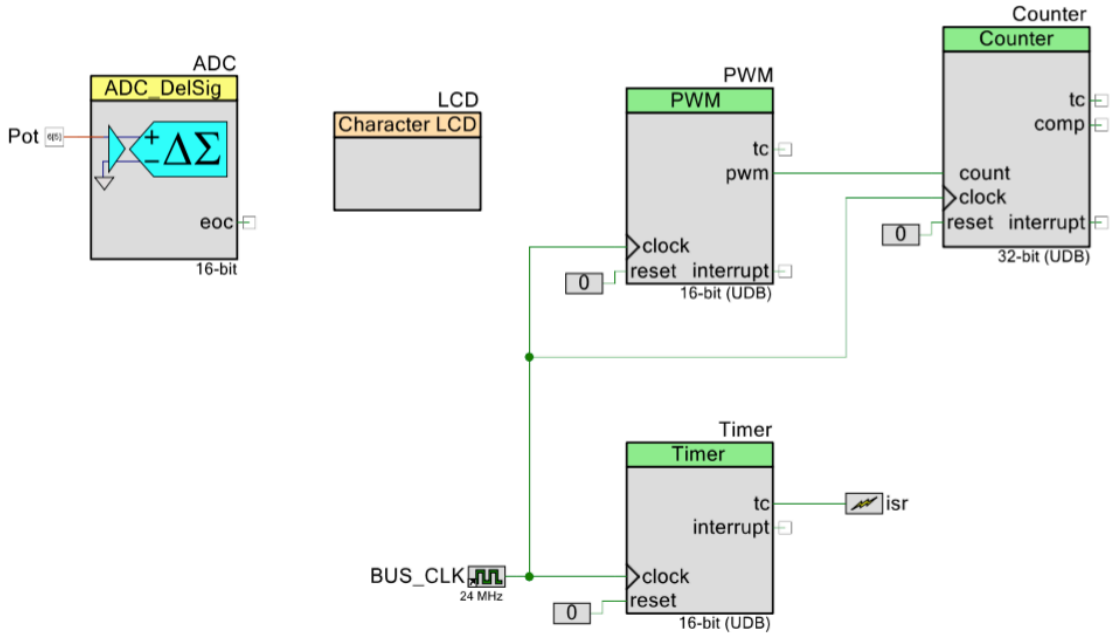

Figure 4: Top-Level Design for Part 2: Frequency Meter

Our program worked in the following manner: we read the value of the potentiometer from the ADC then checked to see if it was turned all the way to the right (0xFFFF) or all the way to the left (0x0000). If the pot was turned to its rightmost (0xFFFF), we would write a period of 1 to the PWM. This caused the PWM to output a signal with a period of 83.333µs, which is equivalent to a frequency of 12 MHz. Moreover, if the pot was turned to its leftmost (0x0000), we wrote a period of 24,000 to the PWM as seen in main.c code

in **Figure 5**. Since we had a 24 MHz clock: 24,000/24MHz produced a 1 kHz output signal.

```
for(;;)
{
    /* Place your application code here. */
    /*PART 2*/
```


main.c

```
    LCD_ClearDisplay();
    LCD_PrintInt32(freq);
    ans = ADC_Read32(); //Get pot value
    if( ans > 0xFFF0){ //Pot turned to the rightmost
        period = 1;
        PWM_WritePeriod(period);
        PWM_WriteCompare(period);
    }else if(ans < 0x0005){ //Pot turned to the leftmost
        period = 24000;
        PWM_WritePeriod(period);
        PWM_WriteCompare(period);
    }else{
        period = 24000-(ans*0.36621);
        PWM_WritePeriod(period/4);
        PWM_WriteCompare(period/4);
    }
    LCD_Position(1,0);
    LCD_PrintInt32(24000000/(1+PWM_ReadPeriod())); //Expected frequency
    CyDelay(300);
}
```

Figure 5: C code from main.c Implementing Part 2: Frequency Meter of this Lab

We had the PWM output connected to the input of the counter. Thus, the frequency of our output signal lied within the counter value. To display the frequency on the LCD screen, we created an interrupt service routine (ISR) that would interrupt every two

milliseconds that would calculate the current frequency based on the potentiometer reading seen in **Figure 6**.

```
CY_ISR(InterruptHandler){
    /*count = Counter_ReadCounter(); //Clear Counter after reading
    timerPeriod = .002;
    freq = count/timerPeriod;
    Timer_ReadStatusRegister();
    Counter_WriteCounter(0x0000);
    */

    count = Counter_ReadCounter(); //Remember previous count
    diff = count - previous;
    previous = count;
    timerPeriod = .002;
    freq = diff/timerPeriod;
    Timer_ReadStatusRegister();

}
```

Figure 6: Interrupt Service Routine Written to Calculate the Frequency of Our Output Signal on Every 2ms Interrupt

Within the ISR, we read the current counter value and determined our calculated frequency. Our calculated frequency was simply the counter value divided by the period of the timer (2 ms).

To measure the frequency continuously, we had the option of either clearing the counter or remembering its current count value before returning from the ISR as seen in **Figure 6**. I thought that clearing the counter would produce more accurate results since no bits would be lost due to any algebra required in determining the count. However, I was wrong; and remembering the previous count value turned out to produce more precise results as seen in **Table 1**.

Frequency Calculated before Clearing the Counter	Frequency Calculated after Remembering Count Value	Percent Difference	Expected Frequency
0x000003E8	0x000003E8	0 %	0x000003E7
0x00000FA0	0x00000FA0	0 %	0x00000FB8
0x00001194	0x00001194	0 %	0x00001284
0x00002CEC	0x00002CEC	0 %	0x00002CEB
0x000053FC	0x00005208	2.33 %	0x00005368
0x00006D60	0x00006D60	0 %	0x00006E4B
0x00017CDC	0x000182B8	1.57 %	0x00018365
0x00021534	0x00021EF8	1.83 %	0x00021DE8
0x00113774	0x00111F58	0.55 %	0x00113579
0x00B44664	0x00B71B00	1.57 %	0x00B71B00

Table 1: Various Frequency Measurements Performed By Either Clearing the Counter or Remembering the Count Value Before Returning From the ISR

The data from **Table 1** depicts that the frequency calculations at the lower end were fairly accurate using either method. However, clearing the counter before returning from the ISR proved to be less accurate as the frequency is increased. Nonetheless, the percent difference between using either method remained low. Although, if we observed inaccurate measurements at the low end range of frequencies, changing the frequency to a double or float data type would improve the precision when the frequency is low.

Conclusion:

I thought this lab was very helpful and informational. This lab provided a complete introduction to using the PSoC-5 microcontroller board and the I/O pins on board. This lab also provided us an opportunity to review programming in C. Furthermore, we learned about pulse-width modulation to control the duty cycle and frequency of a component in a circuit. Moreover, we learned about how to manage multiple digital blocks on a schematic and how to synchronize them using the same clock. We also learned about managing analog inputs within a digital environment by implementing analog-to-digital converters.