ECE341

**Lab9 - PWM Controlling a DC Motor**

Report

**Seth Cram**

**11/30/2021**

**Introduction:**

Goal:

The goal is to generate a signal with Pulse Width Modulation (PWM) using the Output Compare Module to control a CD motor’s speed.

Background Information:

The PWM is a useful control signal. Some characteristics of it include: the pulse rises at a fixed interval, and the falling edge varies. Varying the duty cycle of a PWM changes the amount of time the pulse is high for, per period. So, a higher duty cycle results in a higher amount of energy delivered to the load. Corresponding to this pulse width, during the lab we’ll use the PWM to deliver energy to the DC motor. Therefore, a higher duty cycle should result in a faster motor.

The ‘H-bridge’ allows current to flow in two different directions so the DC motor can rotate in different directions. When no energy is being delivered from the PWM signal to the DC motor, the motor is slowed down by friction. Due to the high frequency that the PWM pulse is delivered at, it's likely we won’t see much of the friction being put into play, unless we totally power off the motor with a 0% duty cycle for long enough.

To create the PWM signal, the output compare module uses the output compare register (OCxR), as well as a Timer, its period register, a comparator, and a counter. The timer creates sawtooth waveforms, while the value loaded into the OCxR determines the pulse of the output PWM signal. We use a 16-bit timer in the lab, but a 32-bit timer could be used for higher resolution. The higher the value loaded into PRx is, the higher the possible resolution, since we could get PWM duty cycles of smaller and smaller differences. But increasing the resolution has the side-effect of decreasing the data rate transferred. A wider PWM signal corresponds to a higher value loaded into OCxR. This is because the value loaded into OCxR corresponds to the number of timer cycles the pulse width is high for.

The reason the rising edge of the PWM signal is always constant is because the PRx value doesn’t change at runtime. The output comparator is clocked by the peripheral bus clock, operating at 10 Mhz. This operating frequency in turn limits our maximum data rate. In addition to OCxR, there’s another register used by the output compare module for OCxRS. The value loaded into this register is determined by our software, and more uniquely, can be changed anytime during runtime. This register serves as an in-between to the OCxR register, since loading that register requires a clock edge to synchronize.

We’ll be using a Half H-Bridge Pmod to drive the motor. It acts as an intermediary between the PIC32 and the DC motor. Two signals come from and go to the PIC32: the desired direction and enable line for the DC motor. Since the DC motor is an inductive load, we can’t create or destroy its current immediately. Clamping diodes are put into place to ensure safety concerns are met. PWM is essentially a cheap ‘DAC’ or Digital to analog converter.

*The background information was found in lecture notes and the lab 9 handout.*

Plan:

First, I’ll create a function for initializing the PWM. This function will take in the desired duty cycle and cycle frequency. We’ll verify these values are within range, and compute period register 2 and output compare register 3 using formulas derived in-class. I’ll then open timer 2 with the same setup as in lab 5, and the previously calculated PR2.

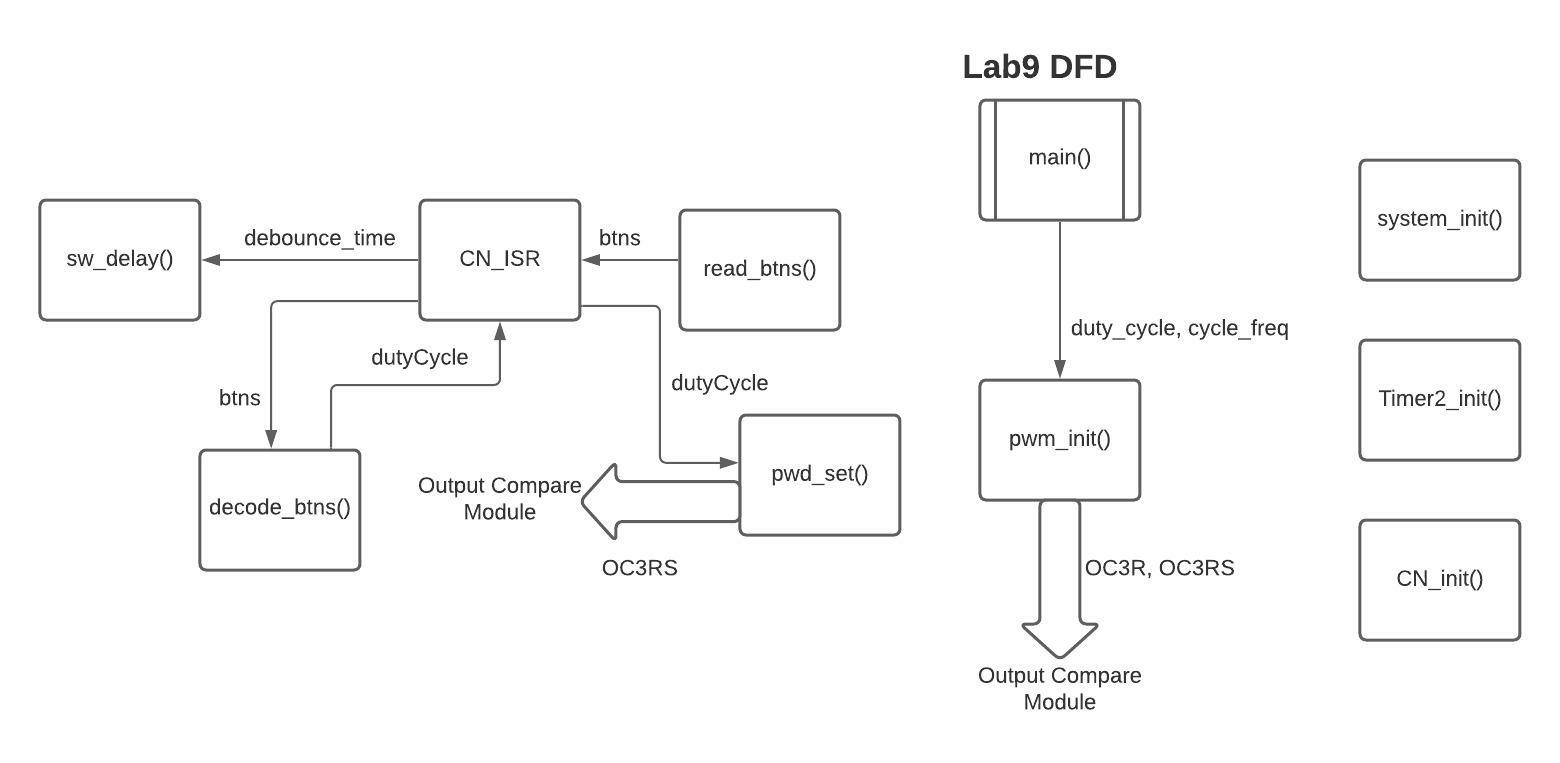
In my main(), I’ll initialize my system by setting up the buttons and LEDs properly and enabling interrupts. Then I’ll call the pwm setup function, and initialize both timer 2 and change notice Interrupt Service Routines (ISRs).

The Timer 2 initialize function will just set its priority to 2 and enable it as an interrupt, since it's already open. The change notice initialization will stay the same from lab 5. The Timer 2 ISR will just toggle LEDA every ms and clear its flag. Although, the change notice ISR will delay to debounce the buttons, read the buttons, decode them using the table presented in the lab handout, and output the determined duty cycle to the set the PWM as. Then, we’ll output the changed duty cycle to the LCD. At the start and end of the function, we’ll also set and clear LEDB for instrumentation. Finally, we’ll clear the change notice flag.

To set the PWM duty cycle during runtime, we’ll need to use a separate function. This function will calculate the OC3RS value just the same as we did for the OC3R value when initializing. Then, it’ll utilize a different PLIB function and verify it was set correctly. Decoding the buttons will use a switch statement for the passed in button-combination, but will instead set the duty cycle variable and pass it back to wherever it was called.

**Implementation Discussion:**

Before implementation, I designed a data flow diagram to get a visual of what files I’d need to incorporate and their overarching functioning in the grand scheme.



As seen above, we’re going to need to use both the change notice from previous labs and Timer2, which is incredibly similar to Timer1 from previous labs. We’ll rehash their ISRs. Not shown above, the Timer2 ISR will simply be used for instrumentation and flip an LED every ms to verify timing.

**Listing 1. pwm.h:**

First, we’ll delve into the constants needed for this lab. All the prototypes included can be inferred from the DFD shown above.

*/\* Software timer const \*/*

*#define COUNTS\_PER\_MS 8889*

*//Debounce btns const:*

*#define DEBOUNCE\_TIME 20 //debounce btns for 20ms*

*//PWM consts:*

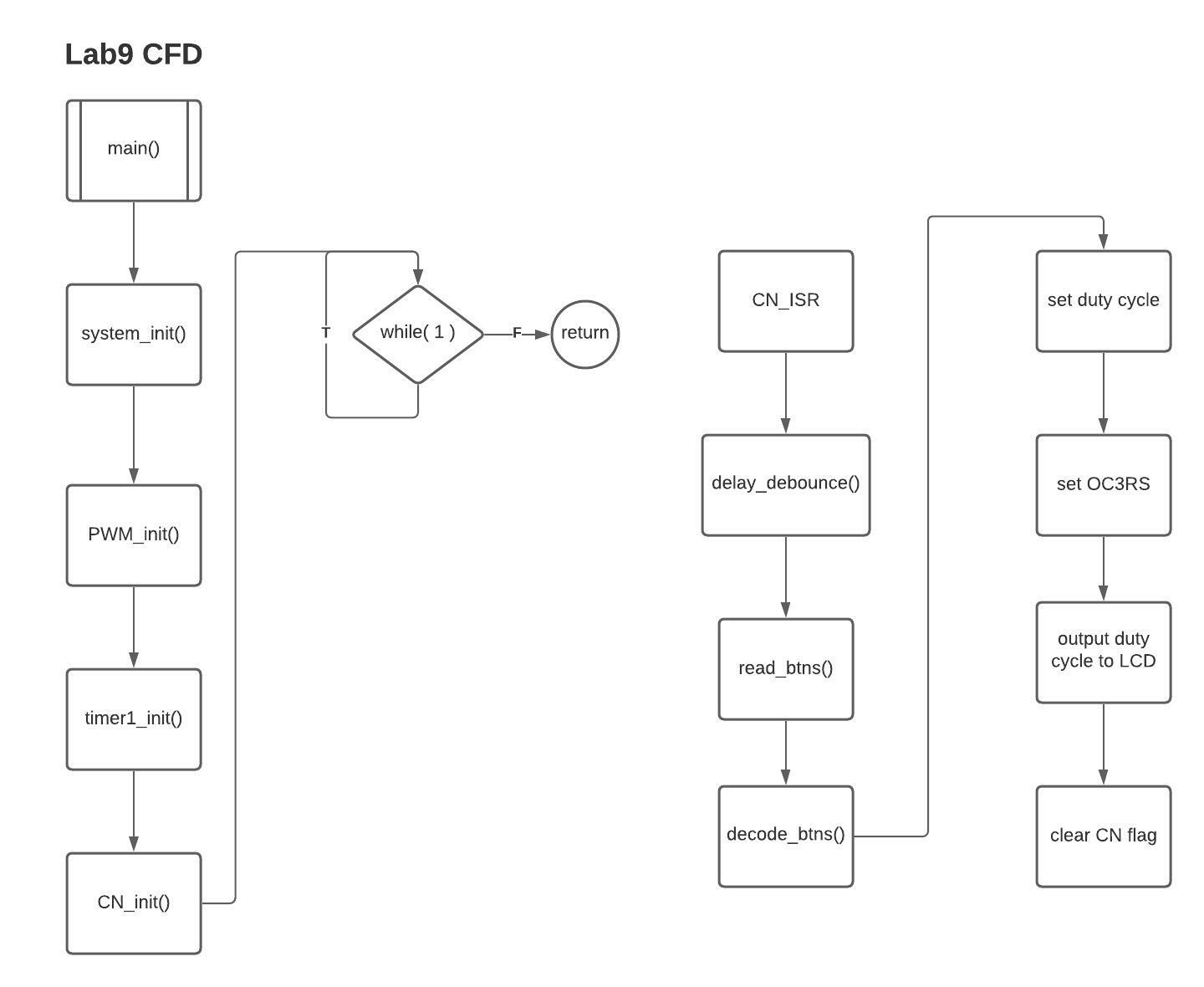
*#define CYCLE\_FREQ 1000 //set for 1ms period*

*#define JIM\_PR2 ( FPB / CYCLE\_FREQ ) - 1*

The reason the calculation for JIM\_PR2 doesn’t include the prescaler for timer 2 is because including it threw errors, and for this lab the prescaler was 1, so I just decided to leave it out. Initially, I incorrectly named JIM\_PR2 as PR2, which conflicted with the name of the Special Function Register, PR2. The error being thrown by this conflict occurred when I tried to open timer 2, so it was incredibly hard to track down. After some help from Dr. J, this problem was remedied by renaming the constant. The other constants are fairly self explanatory. A macro is included for CYCLE\_FREQ for the sole reason that we use it to calculate JIM\_PR2. We need JIM\_PR2 for the pwm\_set() function, as we’ll see below.

**Listing 2. test\_proj9.c:**

The expected headers included in this file were the plib, Cerebot, and pwm header files. In addition to those, I also included the LCDlib header for instrumentation. From then forward, I initialized the duty cycle and error message I wanted output in-case of failure. Next, I pretty much followed the left side of my CFD to a tee.



*system\_init();*

*initLCD(); //for instrumentation*

*//For this lab, the PWM should be initialized with a 40% duty cycle*

*int errorCode = pwm\_init( dutyCycle, CYCLE\_FREQ );*

*if( errorCode != 0 )*

*{*

*LCD\_puts( errorMsg ); //if error, output*

*}*

*t2\_intr\_init();*

*cn\_intr\_init();*

*while( 1 )*

*{*

*//empty, no fg tasks*

*}*

The main difference was that I added in error checking that returned from pwm\_init() for a more robust driver and output to the LCD if an error was detected.

**Listing 3. system\_init():**

Delving into the function contained within pwm.c, the first one we’ll analyze is system\_init(). I normally don’t include this function very in-depth in my reports, but this lab’s required more setup than usual. First, I called Cerebot setup, then:

*//set btn1 and btn2 to digital inputs:*

*PORTSetPinsDigitalIn(IOPORT\_G, BTN1 | BTN2);*

*unsigned int LEDAtoD = LEDA | LEDB | LEDC | LEDD;*

*//set LEDA-D as outputs:*

*PORTSetPinsDigitalOut(IOPORT\_B, LEDAtoD );*

*LATBCLR = LEDAtoD ; /\* Turn off LEDA through LEDH \*/*

*//setup for next proj:*

*PORTSetPinsDigitalOut(IOPORT\_D, BIT\_7 ); //for RD1 (dir)*

*LATDCLR = BIT\_7;*

*PORTSetPinsDigitalIn(IOPORT\_D, BIT\_9 | BIT\_10 ); //for RD3 and 10 (tachometer inputs)*

I enabled the buttons as inputs to later affect the duty cycle of the PWM. I then set LEDA-LEDD to outputs for instrumentation purposes and cleared them. I then set up RD1, RD9, and RD10 in preparation for the next lab. Not shown above is how I enabled multi-vectored and global interrupts after all the other setup.

**Listing 4. pwm\_init():**

This function used the passed in duty cycle and desired cycle frequency to open timer 2 and open the output compare module. First, I did some error checking on the input.

*//if duty cycle isnt 0 to 100:*

*if( dutyCycle < 0 || 100 < dutyCycle ) return 1; //failure*

*//if cycle freq is negative:*

*else if( cycleFrequency <= 0 ) return -1; //failure*

Now that we know our input’s valid, I’ll calculate the PR2 and OC3R value using formulas derived in class and found in the lab handouts.

*//supposed to multiply cycle freq by timer 2 prescaler, but throws error*

*unsigned int pr2 = ( FPB / cycleFrequency ) - 1; //should be 9999 for first run*

*unsigned int oc3r = dutyCycle \* ( pr2 + 1 ) / 100;*

I then use the calculated pr2 to open timer 2 at the correct frequency, and both of them to open the output compare module properly.

*//open timer2:*

*OpenTimer2( ( T2\_ON | T2\_SOURCE\_INT | T2\_PS\_1\_1 ) , pr2 );*

*//open OC3R: (settings), (OCxRS), (OCxR)*

*OpenOC3( ( OC\_ON | OC\_TIMER\_MODE16 | OC\_TIMER2\_SRC | OC\_PWM\_FAULT\_PIN\_DISABLE ), oc3r, oc3r ) ;*

The other values passed to timer2 and OC are their operation settings. I then return 0 at the bottom to signify proper execution of the function.

**Listing 5. pwm\_set():**

This function is for setting the output compare register during runtime. We have to first load a value into OC3RS before it is transferred to OC3R on the rising edge of the clock for synchronous operation. But before all that, I do similar error checking to above upon the input passed to this function.

*//if duty cycle isnt 0 to 100:*

*if( dutyCycle < 0 || 100 < dutyCycle ) return 1; //failure*

Looking back, it would have been wise to make this into a helper function since I use the same code more than once. After that, I calculate the OC3RS value the same as above in pwm\_init(), except I use the JIM\_PR2 macro instead of calculating it here. I don’t calculate it here because the lab report specified that this function only took in the argument of the duty cycle, and in order to calculate the PR value, I needed the cycle frequency.

*unsigned int oc3rs = dutyCycle \* ( JIM\_PR2 + 1 ) / 100;*

*//set new PWM duty cycle:*

*SetDCOC3PWM( oc3rs );*

I then return 0 for success at the bottom.

**Listing 6. CN\_ISR:**

First, I declare some important variables at the top. I then debounce the buttons and read them. Decoding the buttons in this lab works differently than in previous labs for the stepper motor. I pass it the read-in buttons, the direction, and dutyCycle variables. For this lab, we don’t set or change the direction but I expect we might during future labs.

*unsigned int btns, dir, dutyCycle; //local var*

*//debounce the btns for 20 ms:*

*sw\_msDelay( DEBOUNCE\_TIME );*

*btns = read\_buttons();*

*//decode btns into correct duty cycle for PWM:*

*decode\_buttons( btns, &dir, &dutyCycle );*

From there we use the newly set dutyCycle based on the button combination to set the PWM duty cycle. If an error was returned from trying to set the duty cycle, we output an error message to the LCD. Finally, we clear LEDB and the change notice interrupt flag.

*int errorCode = pwm\_set( dutyCycle );*

*//set PWM to desired duty cycle:*

*if( errorCode != 0 ) //error setting duty cycle*

*LCD\_puts( errorMsg ); //if error, output*

*LATBCLR = LEDB; //end of CN ISR*

*mCNClearIntFlag();*

**Listing 7. decode\_buttons():**

Just like in previous labs, I used a switch statement for the buttons and set the variables by dereferencing them based on the button combination. I ended up hardcoding the LCD verification messages within each case since it made the code significantly easier to write.

*switch( buttons )*

*{*

*case BTN1:*

*\*dir = CW;*

*\*dutyCycle = 65;*

*LCD\_puts("duty cycle: 65");*

*break;*

*case BTN2:*

*\*dir = CW;*

*\*dutyCycle = 80;*

*LCD\_puts("duty cycle: 80");*

*break;*

*case BTN1 | BTN2: //both btns pressed*

*\*dir = CW;*

*\*dutyCycle = 95;*

*LCD\_puts("duty cycle: 95");*

*break;*

*default: //neither btn pressed, or more than 2 pressed*

*\*dir = CW;*

*\*dutyCycle = 40;*

*LCD\_puts("duty cycle: 40");*

*break;*

*}*

Some possible issues with my approach is if we were to incorporate more buttons, scalability would be more difficult since I use hardcoding for some of it.

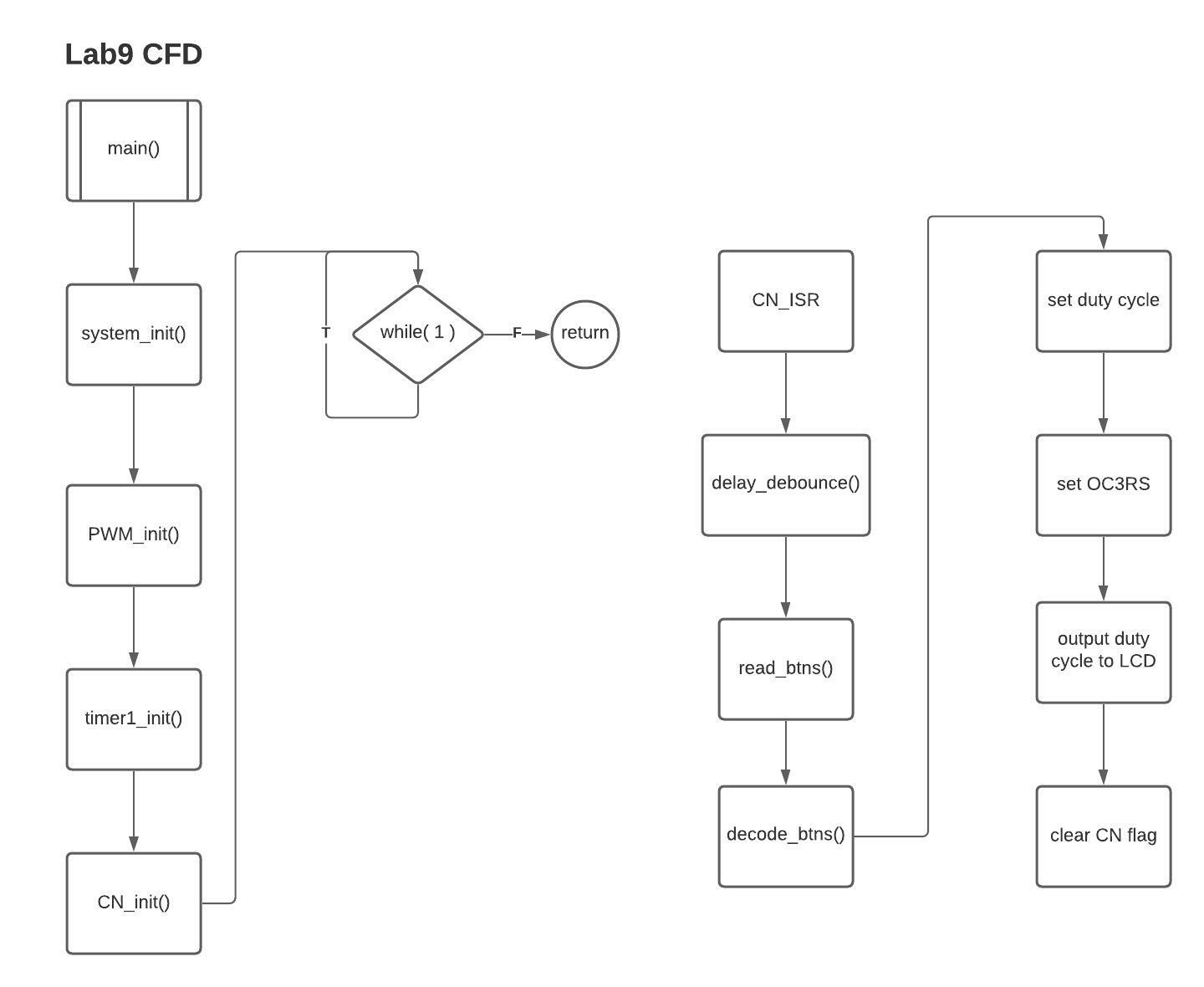
Now, I’ll discuss the other functions not previously mentioned, but held within pwm.c.

Timer2\_init() just sets timer 2’s priority to 2, subgroup priority to 0, and enables it. ChangeNotice\_init() merely does what it did in lab5. There were no changes made to it.

sw\_msDelay() was also not changed from previous labs.

read\_buttons() was, again, imported unchanged.

As seen previously, my control flow diagram models the behavior of the above specified listings:

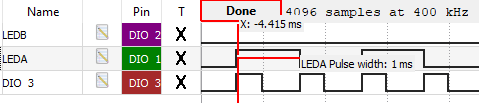


**Testing and Validation:**

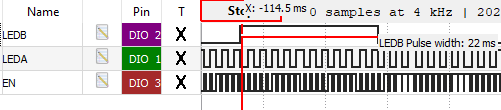
To test my program for the TAs, I showed them the DC motor changing speeds based on each button combination, and the appropriate message being displayed to the LCD screen.

*Include oscilloscope captures of the 1 ms cycle period, the button detect ISR, and the duty cycle for each button combination. Use the measurement menu on the oscilloscope to measure the duty cycle.*

Timer 2 ISR:

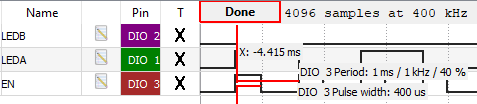


Button Detect ISR:

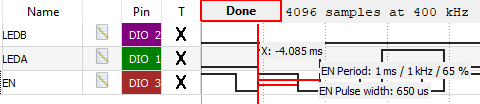


(20 ms taken to debounce the buttons)

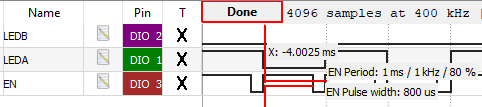
Duty cycle for BTN1, BTN2 = off:



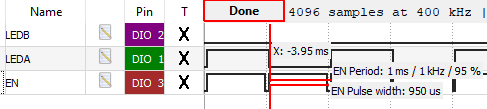
Duty cycle for BTN1 = on, BTN2 = off:



Duty cycle for BTN1 = off, BTN2 = on:



Duty cycle for BTN1, BTN2 = on:



*– Derive an expression that approximates the number of bits required by a conventional DAC to achieve the same resolution as the PWM. Assume a 10MHz peripheral bus clock.*

The resolution of the PWM output compared to an equivalent DAC is the inverse of PWM\_CYCLE\_COUNT. An N-bit DAC where N = log\_based2( PWM\_CYCLE\_COUNT ) rounded up to the nearest integer. This equation assume PWM\_CYCLE\_COUNT = PBCLK / (T2\_PRESCALE \* PWM\_CYCLE\_FREQUENCY), so a 10MHz peripheral bus block results in PWM\_CYCLE\_COUNT = 10,000,000 / (T2\_PRESCALE \* PWM\_CYCLE\_FREQUENCY) .

Combining all this, we get an N-bit DAC where

**N = log\_based2( 10,000,000 / (T2\_PRESCALE \* PWM\_CYCLE\_FREQUENCY) ).**

*– What is the relationship between the PWM cycle period (the carrier period) and the PWM duty cycle resolution (the “fineness”)?*

The duty cycle resolution = 1 / (PR + 1)

Tpwm = (PR + 1) / f\_timer

So, as long as f\_timer doesn’t change, Tpwm is proportional to the duty cycle resolution. To increase the duty cycle resolution, we’d just increase the PWM period to get a smaller possible interval.

*– What is the purpose of the shadow register in the PWM peripheral (OCxRS)?*

OCxRS stores the value we wish to assign OCxR at runtime, and at the end of the current clock cycle moves the value into OCxR. We need this secondary register, since loading OCxR needs to be synchronous in order to occur at the end of a clock cycle instead of in the middle of one. Otherwise, the PWM would be messed up.

*– The PWM output is a digital signal that has essentially two values: VDD and GND. How could a design use a PWM signal to generate a continuous signal (such as a sine wave)?*

By constantly changing the ratio of VDD to GND, or duty cycle, in each period of time. So, the larger portion of a given time period that corresponds to VDD, also corresponds to a higher point on the continuous signal/sine wave. To get a good approximation of a sine-wave, we’d need a large frequency for closer points, which would decrease how many different duty cycles/total points we could have between the peak and trough of the sin-wave. So, the resulting continuous wave would be a rough approximation.

*– What is the effect of changing ONLY the timer prescale value (the PR value and the OCxR value are constant)?*

PR = f periph bus block / (f\_desired \* timer\_prescaler) - 1

The desired frequency would need to decrease to account for a higher timer prescaler.

Tpwm = (PR + 1) /f\_timer

f\_timer = f\_pbclk / prescaler

So, as the prescaler increases, the frequency of the timer decreases, which results in a longer PWM period.

Pulse width = OC3R / f\_timer

As seen above, since f\_timer would decrease with a higher prescaler, the pulse width would also increase.

**Conclusion:**

In conclusion, we learned how to generate a PWM signal to power a DC motor. The PWM is a form of Digital to Analog converter, so we practiced translating signals from digital 1’s and 0’s to continuous, real-world sine wave signals. We also learned how a DC motor operates and about its internals. The output compare module greatly assisted in our setting and changing of the generated PWM signal.

Some limitations of my design include the before-mentioned issue with expanding the number of buttons as inputs. Other limitations include the fact we haven’t yet implemented how to change the direction of the DC motor, just its speed. So, the pwm device driver code we wrote isn’t very complete since we can’t fully control all aspects of the DC motor. Another limitation is that as we increase the frequency of data transfer, the number of different duty cycles settable decreases, and vise-versa. This problem is a problem specific to using PWM.