Seth Cram

ECE341

Lab9 Prelab

11/08/2021

**Prelab 9: PWM Controlling a DC Motor**

Goal:

The goal is to generate a Pulse Width with 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.

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.



