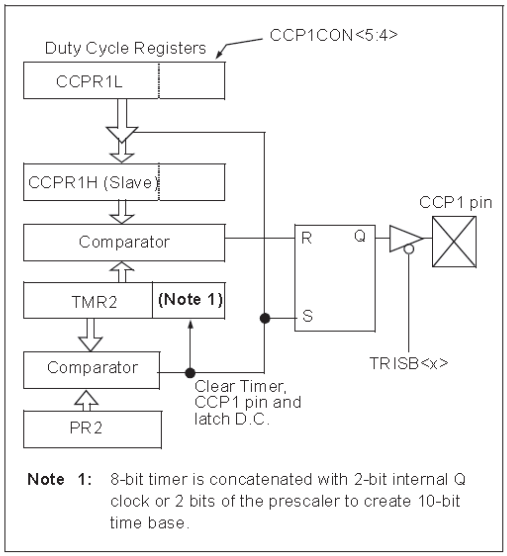
Physics 335
Lab 7 - Microcontroller PWM Waveform Generation

In the previous lab you learned how to setup the PWM module and create a pulse-width modulated digital signal with a specific period and duty cycle. Because the duty cycle in such signals is directly proportional to the average voltage at the output pin, it is possible to vary the power provided by a device that outputs a PWM signal. One can directly use the PWM signal (possibly amplified by a MOSFET switch) to drive a variety of loads, like lamps, motors, etc. And, by adding a low-pass filter, the PWM ups-and-downs can be smoothed out into a continuously variable voltage—it's a quick and easy way to make a digital-to-analog conversion.

This lab will continue the use of PWM signals to show how to do useful things with them. You will also explore a wider range of programming problems.

Recall the structure of the PWM module. It uses both the CCP module and the Timer2 module, as shown below.

FIGURE 9-3: SIMPLIFIED PWM BLOCK DIAGRAM



Recall that this works by running Timer2 into two comparators. One comparator controls the duty cycle part of the output and compares Timer2 to the 10 bit register composed of `CCPR1H`, extended by 2 bits. The output is set HIGH as long as Timer2 is below this number; when it exceeds it, the output goes LOW, and Timer2 continues to run until the other comparator shows that it exceeds the number in `PR2`. At that point, Timer2 is reset, the output is set HIGH again, and a new value for the duty cycle is pulled from `CCPR1L:CCP1CON<5:4>`, and the cycle starts again.

The equations which control the timing are given by

$$\text{PWM Period} = [(\texttt{PR2}) + 1] \cdot 4 \cdot T_{\text{OSC}} \cdot (\text{TIMER2 prescale value}) \quad (1)$$

$$\text{PWM Duty Cycle} = (\texttt{CCPR1L} : \texttt{CCP1CON} < 5 : 4 >) \cdot T_{\text{OSC}} \cdot (\text{TIMER2 prescale value}) \quad (2)$$

To run the CCP module in PWM mode, your code needs to carry out the following steps:

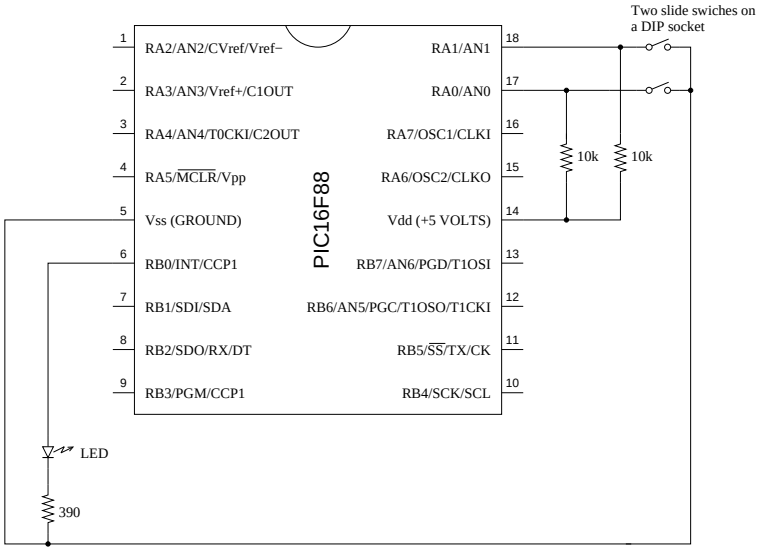
1. Set up the internal oscillator by writing an appropriate value to `OSCCON`.

2. Set the PWM period by writing to `PR2`.
3. Set the PWM duty cycle time by writing to `CCPR1L` and `CCP1CON<5:4>`
4. Enable the output of CCP1 by clearing `TRISB<0>`
5. Set the TIMER2 prescale value and enable TIMER2 by writing to `T2CON`
6. Enable and start PWM by writing to `CCP1CON`.

Remember: the special function registers live in different banks. `OSCCON`, `PR2` and `TRISB` are in bank 1, while `CCPR1L`, `CCP1CON`, and `T2CON` are in bank 0. You will want to have a copy of the PIC16F87/88 Data Sheet handy to make sure you are writing to the correct bits and registers.

7-1. A simple dimmer circuit

We will start "using" a PWM signal to make a simple dimmer circuit. Because the power output capabilities of the PIC are pretty low, we will only drive an LED. Connect up the circuit shown below. The goal is to make the LED intensity vary among four different brightnesses depending on the switch settings.



The switches are two elements from a line of tiny slide switches that sit on a DIP socket. You could also use the standard toggle switches, but the ones on the DIP socket will make a cleaner layout.

Next, make a new PIC project as you have done before. Don't forget to include the `p16f88.inc` file and to modify the configuration to take power from the PICkit3 programming device.

Use your file from the previous lab that set up and ran the CCP module to make a PWM waveform with the 2 kHz frequency and 75% duty cycle. This file will be your template for the projects in today's lab.

Before you start coding, compile and run your code from the previous lab. You should be able to see the LED light up when it runs. If you don't, troubleshoot this problem first, otherwise nothing else will work.

If all is OK, proceed to implement the following algorithm:

1. Set up PORTA<1:0> as a digital input. Remember, you need to deal with `TRISA` and `ANSEL` do do this.
2. Set up and start a PWM waveform using a 500 kHz clock and a period of 256 instruction cycles (`PR2` = 255).
3. Set up a loop that does the following:
   (a) Read PORTA<1:0> into a variable.
   (b) Test the bit pattern received.
   (c) Depending on the pattern, turn the LED off, 1/3 on, 3/3 on, or fully on by loading an appropriate value into `CCPR1L`.
   (d) Go back to (a) and repeat.

The tricky part of this loop is testing the bit pattern. You can use the `BTFSS` functions on each bit in the variable read from PORTA, or you could make an operation with the entire number in that variable and test the STATUS register for, say, the zero bit. For example, notice the `XORWF` instruction. It make a bitwise exclusive-OR operation between the W register and a memory register. If the two registers contain the same bit pattern, the result will be all 0's, which sets the zero flag in STATUS. The following code snippet shows how to choose among different bit patterns:

```
Mainloop   MOVF     PORTA, W
           MOVWF    PortTest

Test0      MOVLW    H'00'
           XORWF    PortTest, W
           BTFSC    STATUS, Z
           GOTO     HandleZeroCase
Test1      MOVLW    H'01'
           XORWF    PortTest, W
           BTFSC    STATUS, Z
           GOTO     HandleOneCase
           .
           .
           .
```

In the above, `HandleZeroCase`, `HandleOneCase`, etc., are the labels of code blocks that execute the code needed for the different cases. Note that by setting W as the destination of the XOR operation leaves the bit pattern in `PortTest` unchanged. Eventually, all code blocks should GOTO back to `Mainloop`.

For your report. Demonstrate your code to the TA and print it out (with comments) to include with your report.

7-2. Using the PWM as an D/A converter

As we noted, if you were to *average* the PWM signal over time, you would get a voltage that was proportional to the duty cycle: 0% duty cycle would give 0 V, 100% duty cycle would give 5 V and 50% duty cycle would give 2.5 V. One can get an average by running the signal into a simple *RC* low-pass filter; the capacitor charges up during the high part of the cycle and discharges during the low part. If the RC time constant is longer (enough) than the PWM wave's period, the charging and discharging curves will quickly reach a steady-state value.

For your report. What value of *RC* would be between 10 and 50 times your PWM period? Figure this out and select the parts to make such a low-pass filter. (Note: you will want to use resistor values above 1k, or else the current draw from the PIC may become too large.)

Remove the LED and its resistor, but leave the switch circuit in place. Wire up the *RC* low-pass filter to the RB0/CCP1 pin. Hook up the scope to filter's output, and have a look. Change the switch settings and look at the voltage on the scope. Is the voltage what you expect?

For your report. Include a circuit diagram of your RC circuit. No need to label all of the pins on the PIC, just the relevant ones. Draw/explain what you see on the scope.

OK, now that you've done that, lets take it a step further. Let's vary the duty cycle, and thereby vary the output voltage. This is really kind of cute, because we're using a single digital line to create an arbitrarily shaped waveform.

The easiest type of varying output waveform to make would be a "sawtooth" wave: a ramp that goes from low to high, and then starts over.

Your code should do the following. Unlike with the LED dimmer, you need to make a delay loop for this program to let the output settle ate each value of `CCPR1L`:

1. Use a variable to hold a value for `CCPR1L`.
2. Initialize the variable to 0.
3. Write the variable to `CCPR1L`.
4. Start the PWM waveform.
5. Increment the variable by some amount.
6. Write the variable to `CCPR1L`.
7. Wait a sufficient time for the RC filter to settle.
8. Go back to step 5, and repeat.

Once your variable reached FFh (255), it will roll over to zero (or a low value, depending on your increment size).

For your report. Demonstrate the result to the TA and print out the code. You know the drill!

7-3. A sine-wave generator

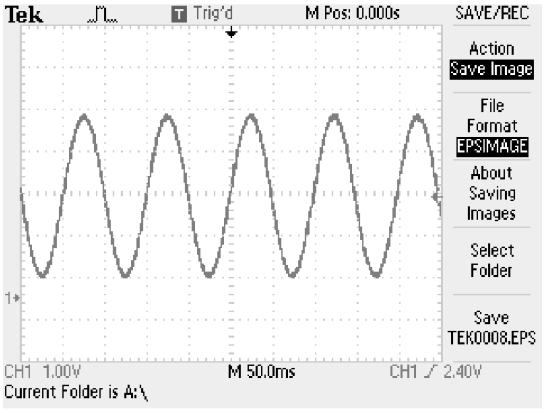
To make an arbitrarily shaped waveform, the value of `CCPR1L` must be varied in an arbitrary way. One way to do this is to use a "look-up table" It works as follows:

1. Set a series of `RETLW` statements in a named subroutine. the argument of `RETLW` a literal that sits in W on return.
2. Write the step number corresponding to which RETLW statement you want into W.
3. Call the subroutine with the first instruction of the subroutine as: `ADDWF PCL`. This causes the program counter to immediately jump forward W steps to the desired `RETLW` statement.
4. Whence the routine immediately returns from the subroutine with the desired value of the duty cycle in W.
5. Write the value to `CCPR1L`, and proceed as before.

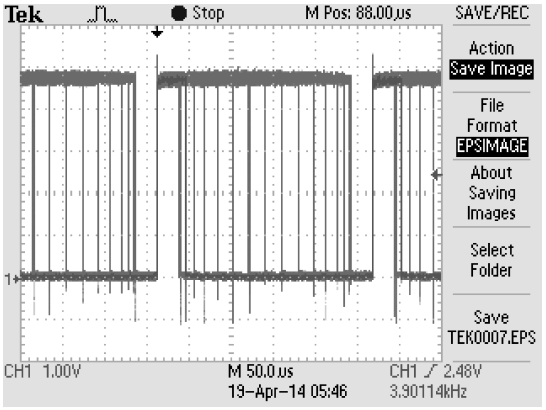
You will need to keep track of your step variable so that it resets appropriately when you reach the end of the look-up table. If you fail to handle this correctly, your program counter may wind up in some place where there is no code!

(Also: in creating your code, you can use the `DT` directive in the assembler. This tells the assembler to create a series of `RETLW` statements as a data table for just this purpose. See the assembler documentation for more details.)

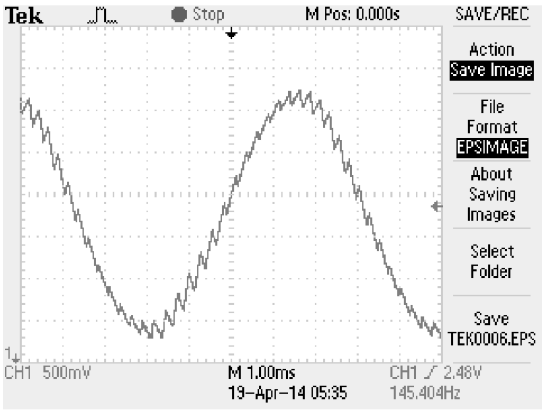
As an example, we used a look-up table to create this sine wave:



This comes from the smear of overlain pulses show below:



It's made by simply changing the pulse width modulation appropriately and filtering. Choosing an RC filter is an example of an engineering compromise. In this case, a compromise between speed and smoothing. If you choose a smaller time constant it may well respond more quickly but at the expense of seeing more of the PWM coming through the filter, which will give something like:



This is the essence of switched power supplies, which maintain a constant voltage by (very quickly) switching on and off and varying the PWM to match the attached load. Obviously, high load = high duty cycle and low load = low duty cycle. This is also the basis for most solid state motor drive systems today. Just as PWM switching on a RC circuit gives a smoothed voltage waveform (voltage across a capacitor changes slowly), PWM on a motor (an inductor) gives a smoothed current waveform.

The look-up table that makes this is given here:

```
Sinedata   ADDWF    PCL, F       ; Increment into table
           RETLW   .128          ; 0 degree, 2.5 volt
           RETLW   .148
           RETLW   .167
           RETLW   .185
           RETLW   .200
           RETLW   .213
           RETLW   .222
           RETLW   .228
           RETLW   .230          ; 90 degree, 4.5 volt
           RETLW   .228
           RETLW   .222
           RETLW   .213
           RETLW   .200
           RETLW   .185
           RETLW   .167
           RETLW   .148
           RETLW   .128          ; 180 degree, 2.5 volt
           RETLW   .108
           RETLW   .89
           RETLW   .71
           RETLW   .56
           RETLW   .43
           RETLW   .34
           RETLW   .28
           RETLW   .26           ; 270 degree, 0.5 volt
           RETLW   .28
           RETLW   .34
           RETLW   .43
           RETLW   .56
           RETLW   .71
           RETLW   .89
           RETLW   .109
```

OK, now time for you to try.

For your report. As usual, demonstrate the result to your TA and print the code for your report.

Challenge problem

Can you use your switch handling code to alter the sinewave output in terms of amplitude? How about frequency?