Physics 335
Lab 6 - PIC Lab 2: Microcontroller based PWM

By now you've had a little introduction to the PIC both in class and in lab, and hopefully you're becoming a little more familiar with the architecture and coding of this chip. We'll use a couple of simple functions of the chip and these should remind you of earlier times when we've used similar techniques with discrete chips.

Today, we'll start using some on-board counters on the chips. To make it a little more challenging and show some of the utility of the microcontroller itself, we'll use these on-chip counters in a special way that's very useful. We'll use the built in functionality to make a simple microcontroller PWM-based waveform generator.

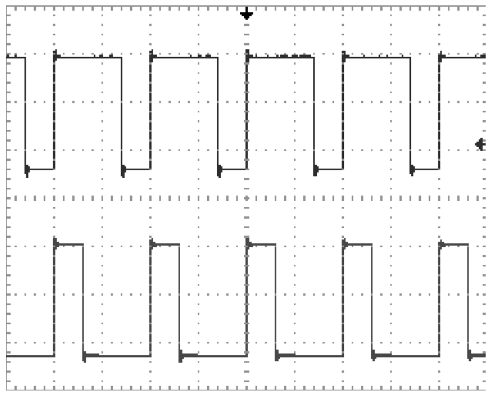
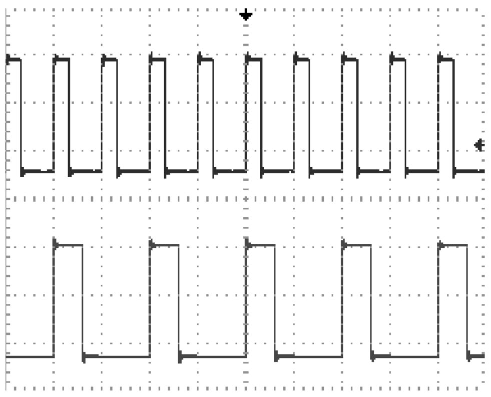
An Introduction to Pulse Width Modulation

Pulse Width Modulation (PWM) has become a very important technique throughout electronics as a method for controlling high power equipment efficiently. It's used for lighting control dimming circuits, switching power supplies and "Uninterruptible Power Supplies" (UPS's), motor and servo control, and countless other applications that used to be the domain of high power (and heavy) linear power supplies and controls.

Pulse Width Modulation is just what it sounds like: using a technique to change the width (in time) of a digital pulse. As you're well aware now, the digital world is composed entirely of ones and zeros—on and off. A digital waveform is often said to have a "period" (the time it takes for the signal to repeat itself) and a "duty cycle", which is defined as

$$\text{Duty Cycle} = \frac{\text{Time of signal high}}{\text{Period}} \times 100 \tag{1}$$

The duty cycle is usually expressed as a percent of the period. The following 'scope grabs show some examples of pulse width modulated signals. The left shows two waveforms with the same frequency but different duty cycles. The other shows the the same duty cycle but different frequencies. (Remember: frequency = 1/period)

A microcontroller makes it easy to do this and to vary the frequency or duty cycle on the fly as needed for a given load or other circuit conditions.

6-1 Port-based PWM

As a reminder of some of your programming from last week, lets do a simple exercise and make a pulse width controller with our PIC. For starters we'll do it the simplest way we can: just turn on a digital output line to high, wait an appropriate period, then set it to low. Then do it again; as they say on the shampoo bottle: "Lather, rinse, repeat."

So here's your first task, hopefully a pretty simple one: *Make a pulse-width-modulated signal with 30% duty cycle at 1kHz.*

(A) Microcontroller Clocking

Before you can do this, you need to think about about how fast our little microcontroller works. Fortunately, as we've said, it's a simple controller and that makes our life (relatively) easy.

We glossed over this fact in the last lab, but our little processor needs a clock that tells it when to execute. MicroChip was nice to us and the newer versions of these chips have an on-board clock, so all we need to do is figure out how to use it (previously, and as you found in other lab exercises, we needed to *supply* a clock signal from the outside world. Now, a few configuration bits, set correctly, are all that's needed.)

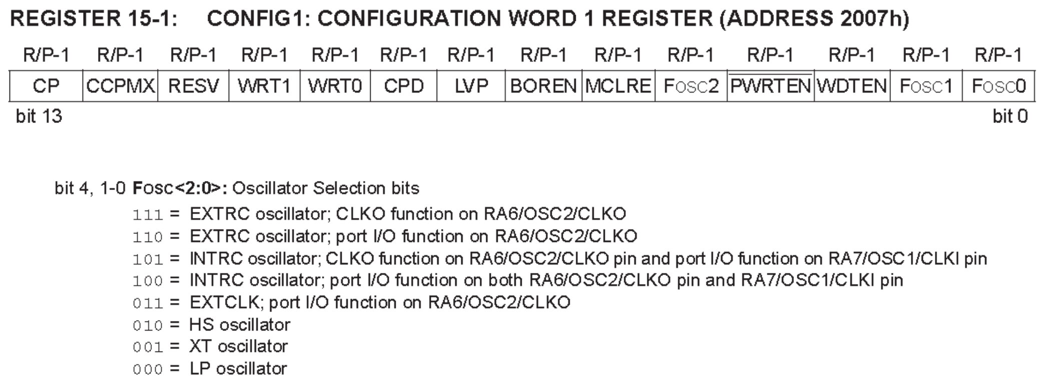
The microcontroller clocking is controlled by a number of special function registers. (The oscillator configuration is discussed in Section 4 of the PIC16F87/88 Datasheet.) Two of them concern us most:

`CONFIG1` The configuration registers `CONFIG1` and `CONFIG2` control how the microcontroller responds when it is first powered-on or reset by other events. These registers are not the same as the usual SFRs in the sense that they do not reside in the data memory space but are instead in *program memory.* Another odd aspect of them is that they are at program addresses 2007h and 2008h, which are both *above* the normal program memory address space (0000h to 1FFFh). They are only accessible during programming of the PIC, and because they are in program memory, they are 14 bits wide (like opcodes) rather than 8 bits wide.

`OSCCON` This register is a "normal" special function register, located in data memory at address 8Fh. It is used to set and control the oscillator configuration after the microcontroller is up and running.

Your first task will be to set the configuration register `CONFIG1` so that the oscillator signal (divided by four) is made available on one of the device pins. Then you will set the control register `OSCCON` to choose a different clock frequency.

The part of `CONFIG1` that concerns us are the "Oscillator Selection Bits", a.k.a. `FOSC<2:0>` which are bits 0, 1, and (curiously) 4. They are denoted `FOSC0`, `FOSC1` and `FOSC2`, respectively. Shown below is a map of `CONFIG1` and what the `FOSC<2:0>` bits do.



REGISTER 15-1: CONFIG1: CONFIGURATION WORD 1 REGISTER (ADDRESS 2007h)

R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1
CP	CCPMX	RESV	WRT1	WRT0	CPD	LVP	BOREN	MCLRE	FOSC2	PWRTEN	WDTEN	FOSC1	FOSC0
bit 13													bit 0

bit 4, 1-0 **FOSC<2:0>:** Oscillator Selection bits

111 =	EXTRC oscillator; CLKO function on RA6/OSC2/CLKO
110 =	EXTRC oscillator; port I/O function on RA6/OSC2/CLKO
101 =	INTRC oscillator; CLKO function on RA6/OSC2/CLKO pin and port I/O function on RA7/OSC1/CLKI pin
100 =	INTRC oscillator; port I/O function on both RA6/OSC2/CLKO pin and RA7/OSC1/CLKI pin
011 =	EXTCLK; port I/O function on RA6/OSC2/CLKO
010 =	HS oscillator
001 =	XT oscillator
000 =	LP oscillator

The configuration bits are set using the lines at the top of our assembly language source file. Note the following line:

```
__config _CONFIG1, _FOSC_INTOSCIO & _WDT_OFF & _LVP_OFF &
         _PWRTE_OFF & _BODEN_ON & _LVP_OFF & _CPD_OFF &
         _WRT_PROTECT_OFF & _CCP1_RB0 & _CP_OFF
```

Note also that the above is *one long line* in the source file, not broken into three lines as shown above. If you examine the labels in the line, you will be able to correlate what they are with what bits are set in `CONFIG1`. What the above long line does is to make a "bitwise-AND" of all of the numbers associated with the labels (equates in the p16F88.inc file, for example, `STATUS  EQU  H'0003'`). The result of this bitwise-AND is a bit pattern that is passed to `CONFIG1`.

Start up the MPLABX IDE as you did before. Create a new project, set the correct processor, allow it to be powered by the PICkit3, etc., just like you did in the previous lab. Grab both the `p16f88.inc` and `PIC16F88_template1.asm` files and install them in the usual spots on the tree.

Now open up the template .asm file and have a look at the top lines.

See the label `_FOSC_INTOSCIO`? That sets the `FOSC<2:0>` bits to `100`, and thus it selects the internal oscillator and sets the output pin RA6/OSC2/CLKO (pin 15 on the chip) to be a regular I/O pin. But we want to look at the clock. So in your source file replace `_FOSC_INTOSCIO` with `_FOSC_INTOSCCLK`. Be sure to type it *exactly* as shown. This will set `FOSC<2:0>` to `101`, and this will allow the clock signal to appear on pin 15 of the device.

Now, go ahead and compile the file and let it "run" on the PIC. (For example, you can run it under "debug mode.") It won't "do" anything at all, since there are no instructions other than one `GOTO`. But it will initialize the PIC, and importantly, turn on the CLKO pin.

Connect a 'scope probe up to the pin and look at the output. You should see a square wave oscillating happily away.

- What is the frequency?
- When the PIC turns on, the register `OSCCON` contains 00h. Based on the information in the Datasheet, what should the frequency of the clock be?
- How are the measured and expected clock frequency related?

Next, you should write some code to change the `OSCCON`, and see what happens. Figure out what bit pattern will make the internal oscillator run at 1 MHz. Then code in a couple of MOV instructions to load this register and watch what happens on pin 15. Below is information on `OSCCON`.

REGISTER 4-2: OSCCON: OSCILLATOR CONTROL REGISTER (ADDRESS 8Fh)

U-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0
—	IRCF2	IRCF1	IRCF0	OSTS(1)	IOFS	SCS1	SCS0
bit 7							bit 0

bit 7	**Unimplemented:** Read as '0'
bit 6-4	**IRCF<2:0>:** Internal RC Oscillator Frequency Select bits 000 = 31.25 kHz 001 = 125 kHz 010 = 250 kHz 011 = 500 kHz 100 = 1 MHz 101 = 2 MHz 110 = 4 MHz 111 = 8 MHz
bit 3	**OSTS:** Oscillator Start-up Time-out Status bit(1) 1 = Device is running from the primary system clock 0 = Device is running from T1OSC or INTRC as a secondary system clock **Note 1:** Bit resets to '0' with Two-Speed Start-up mode, and LP, XT, or HS selected as the oscillator mode.
bit 2	**IOFS:** INTOSC Frequency Stable bit 1 = Frequency is stable 0 = Frequency is not stable
bit 1-0	**SCS<1:0>:** Oscillator Mode Select bits 00 = Oscillator mode defined by FOSC<2:0> 01 = T1OSC is used for system clock 10 = Internal RC is used for system clock 11 = Reserved

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

The oscillator frequency is selected using bits `<6:4>` of the `OSCCON` register. So to select a 250 kHz clock, we'd need `OSCCON` to look something like `x010xxxx` (showing the set and reset bits corresponding to the correct frequency mask). But be careful, although you might think it might, the following code will *not quite* set the processor to use a 250 kHz clock.

```
MOVLW b'00100000'
MOVWF OSCCON
```

You need a bit more. (What? Hint: what is the address of `OSCCON`?)

If you are feeling energetic, you may want to write a loop that changes `OSCCON` to different settings and then delays for a moment so that you can see the new frequency register on the 'scope. Note, however, that your delay loop will also be affected by the oscillator frequency!

(B) Software-based PWM

Now you should be ready to tackle the problem. Here is a rough outline:

(a) Initialize the on-board oscillator so that it's fast enough for the task at hand.
(b) Initialize `PORTB<0>` as an output pin.
(c) By using bit set (`BSF`) or move functions, turn the port bit ON.
(d) Waste an appropriate amount of time.
(e) Use an appropriate function to turn the port bit OFF.
(f) Waste an appropriate amount of time.
(g) Go back to (c) and repeat.

The hard part of the above is to decide how to waste the appropriate amounts of time. To figure this out, you need to know the following: How many clock cycles does one iteration of your delay loop take? How many clock cycles should the time ON and the time OFF take? Based on this, how many iterations of the delay loop should occur for each part of the cycle?

Your lab report should include these calculations.

A couple of pointers before you go forth and code: First, when you need your processor to "waste some time" you can use `NOP` operations. Second, remember that each loop count variable must be between 0 and 255. If you want larger numbers, you will need to figure out how to "hook two variables together." (Hint: nested loop.) Finally, look at the instruction list to see how many instruction cycles each instruction takes. Most need only 1 cycle, but any that change the program counter require 2 cycles.

Now that you have done a little bit of coding, you should start using named variables and equates to make your programs clearer and easier to manage. The following example shows how to replace the literals and memory (also called "file") addresses with names. It is the same delay loop code used in the last lab—it just looks different.

```
MaxCount    EQU     H'0A'     ; Equates are good for defining literals

   CBLOCK   H'20'             ; A CBLOCK defines a sequential region
      L1Var                   ; L1Var is in location H'20'
      L2Var                   ; L2Var is in location H'21'
   ENDC                       ; ENDC ends the definition block

; Begin Delay Loop

Delay       MOVLW   MaxCount  ; Notice how above declarations
            MOVWF   L1Var     ; are used here
Loop1       MOVLW   MaxCount
            MOVWF   L2Var
Loop2       DECFSZ  L2Var
            GOTO    Loop2
            DECFSZ  L1Var
            GOTO    Loop1

; End Delay Loop
```

More information and examples are given in the MPASM User's Guide, on the course website.

OK, go forth and code. When done, prove to your TA you've created the signal described: hook your scope to the RB0/CCP1 pin (pin 6). Print out your program to include in your report.

6-2 Using the PWM on-board peripheral

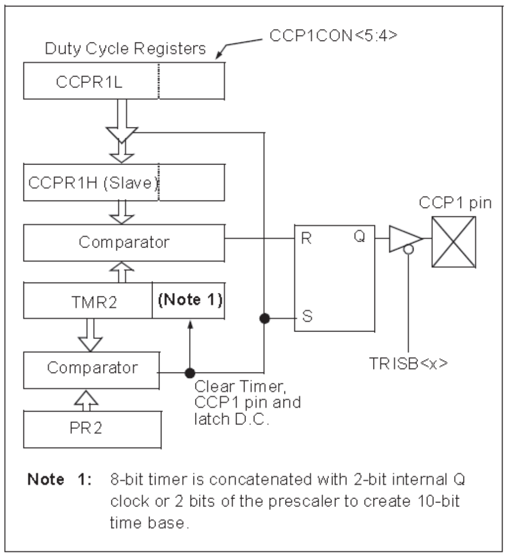
OK, so you've made a pulsing signal. So what? Well, it's useful as is, but it may have occurred to you that if you want to do anything more than just pulse that one output line you need to work that into your code. For example, you may want your processor to interact with a user in some way, say, to change the operating conditions in response to the user pressing a button. All this takes processor time, but we're using the processor timing to time our loop, so we have to figure out how long each interaction will take and plan for it.

This can get very tricky and it's clumsy to use, unless ALL you want to do is pulse the line. Because PWM signals are useful in many contexts, the manufacturer has created a "peripheral" specifically designed to handle PWM in a predictable and easier-to-code way.

The peripheral is called the Capture/Compare/PWM or CCP module. It works with another on-board peripheral, a timer/counter module called TIMER2.

The block diagram of the CCP module when it is configured to do PWM is shown below.

FIGURE 9-3: SIMPLIFIED PWM BLOCK DIAGRAM


Duty Cycle Registers
CCP1CON<5:4>
CCPR1L
CCPR1H (Slave)
Comparator
TMR2
(Note 1)
Comparator
PR2
R
Q
S
CCP1 pin
TRISB<x>
Clear Timer, CCP1 pin and latch D.C.
Note 1: 8-bit timer is concatenated with 2-bit internal Q clock or 2 bits of the prescaler to create 10-bit time base.

The way it works is as follows. The CCP1 pin is enabled by setting `TRISB<0>` to 0 ("output") and by setting the appropriate bits in the `CCP1CON` register. This turns RB0 into CCP1—same pin, different internal connection. The period of the waveform is put into the `PR2` register associated with TIMER2. The duty cycle time is put into the `CCPR1L` register and two bits of the `CCP1CON` register, `CCP1CON<5:4>`. The 8 bits from `CCP1RL` and the 2 bits from `CCP1CON<5:4>` are concatenated to form a 10 bit word. When TIMER2 starts and runs, the output latch is set (S) so that the output is HIGH, and at the same time, CCPR1L (extended by 2 bits) is latched into CCPR1H (also extended). The register `TMR2` is updated continuously, and this feeds two digital comparators that check whether the number in `TMR2` is greater than or equal to the numbers feeding into the other side of each comparator. When `TMR2` (extended) is greater than `CCPR1H` (extended), the comparator resets the latch (R) pulling the output LOW (thus ending the "duty" part of the cycle). The timer continues to run until `TMR2` matches `PR2` which then sets the output latch (S) HIGH, clears `TMR2` and latches in the next duty cycle number from `CCPR1L` and `CCP1CON<5:4>`. Whew! That is a lot to keep track of, no?

In order to make a particular PWM waveform, you need to solve a couple of equations, based on what you want the frequency and duty cycle to be. These are

$$\text{PWM Period} = [(\texttt{PR2}) + 1] \cdot 4 \cdot T_{\text{OSC}} \cdot (\text{TIMER2 prescale value}) \tag{2}$$
$$\text{PWM Duty Cycle} = (\texttt{CCPR1L} : \texttt{CCP1CON} < \texttt{5} : \texttt{4} >) \cdot T_{\text{OSC}} \cdot (\text{TIMER2 prescale value}) \tag{3}$$

The "TIMER2 Prescale value" comes from the TIMER2 module. T_{osc} is the period of the *oscillator* itself (not the instruction cycle time). The clock driving TIMER2 is first sent into a pair divide-by-4 counters that may be switched on according to 2 bits in the `T2CON` register. Thus the actual timer frequency may be reduced by 4 or by 16.

Here are some relevant details about the relevant registers from the PIC datasheet. You should read the sections of the datasheet about each of these registers thoroughly. As you've learned, electronics can be *very* unforgiving of small mistakes, so it's quite possible to have a single bit misconfigured and have that completely change what you see. Your debugging skills will be helpful here, if things don't at first work out.

First, a summary of those registers you'll need to configure for PWM

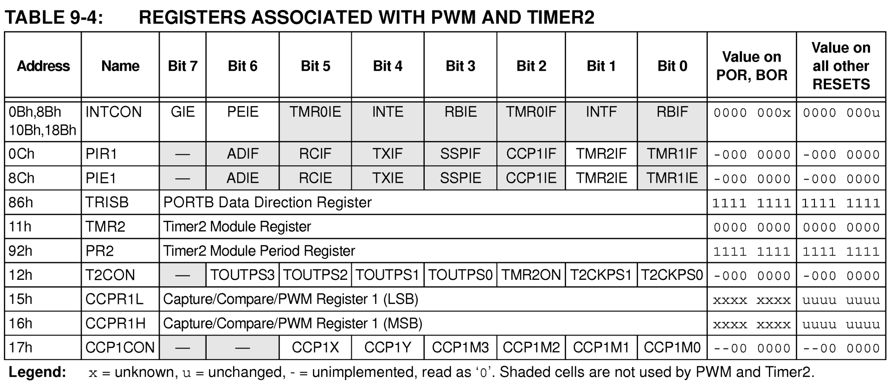
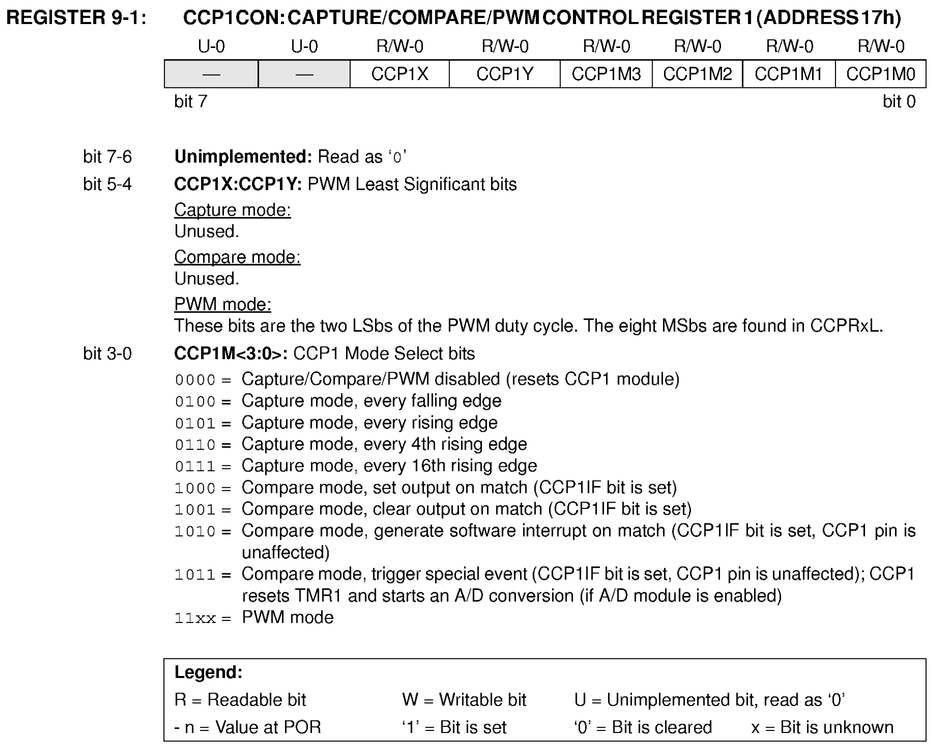


TABLE 9-4: REGISTERS ASSOCIATED WITH PWM AND TIMER2

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other RESETS
0Bh,8Bh 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	—	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	-000 0000	-000 0000
8Ch	PIE1	—	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	-000 0000	-000 0000
86h	TRISB	PORTB Data Direction Register								1111 1111	1111 1111
11h	TMR2	Timer2 Module Register								0000 0000	0000 0000
92h	PR2	Timer2 Module Period Register								1111 1111	1111 1111
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PWM and Timer2.

Of these `CCP1CON` sets the configuration of the Capture, Compare, PWM register.



REGISTER 9-1: CCP1CON: CAPTURE/COMPARE/PWM CONTROL REGISTER 1 (ADDRESS 17h)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **CCP1X:CCP1Y:** PWM Least Significant bits

Capture mode:
Unused.

Compare mode:
Unused.

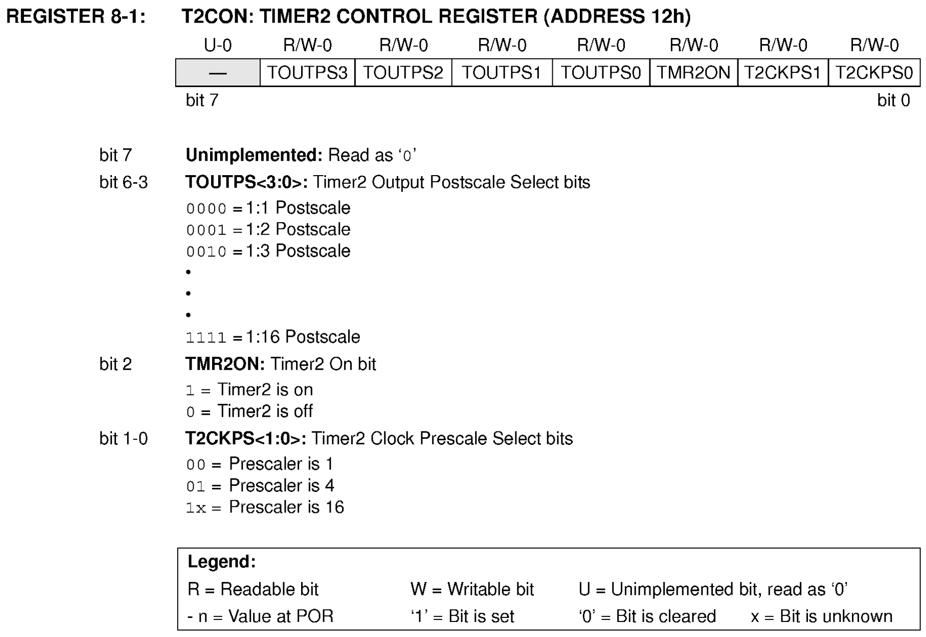
PWM mode:
These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPRxL.

bit 3-0 **CCP1M<3:0>:** CCP1 Mode Select bits

`0000` =	Capture/Compare/PWM disabled (resets CCP1 module)
`0100` =	Capture mode, every falling edge
`0101` =	Capture mode, every rising edge
`0110` =	Capture mode, every 4th rising edge
`0111` =	Capture mode, every 16th rising edge
`1000` =	Compare mode, set output on match (CCP1IF bit is set)
`1001` =	Compare mode, clear output on match (CCP1IF bit is set)
`1010` =	Compare mode, generate software interrupt on match (CCP1IF bit is set, CCP1 pin is unaffected)
`1011` =	Compare mode, trigger special event (CCP1IF bit is set, CCP1 pin is unaffected); CCP1 resets TMR1 and starts an A/D conversion (if A/D module is enabled)
`11xx` =	PWM mode

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

`T2CON` similarly sets configuration for the Timer2 Register, which is used for PWM functionality.



REGISTER 8-1: T2CON: TIMER2 CONTROL REGISTER (ADDRESS 12h)

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

bit 6-3 **TOUTPS<3:0>:** Timer2 Output Postscale Select bits

`0000` = 1:1 Postscale
`0001` = 1:2 Postscale
`0010` = 1:3 Postscale
•
•
•
`1111` = 1:16 Postscale

bit 2 **TMR2ON:** Timer2 On bit

`1` = Timer2 is on
`0` = Timer2 is off

bit 1-0 **T2CKPS<1:0>:** Timer2 Clock Prescale Select bits

`00` = Prescaler is 1
`01` = Prescaler is 4
`1x` = Prescaler is 16

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
- n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Your next task is to use the on-board CCP and TIMER2 modules to create a PWM signal with a frequency of (about) 2 kHz and 75% duty cycle.

Use the equations above to work out the values needed for `PR2` and `CCPR1L:CCP1CON<5:4>`. Show your work in your report.

Write code that carries out the following steps:

(a) Set the PWM period by writing to `PR2`.

(b) Set the PWM duty cycle time by writing to `CCPR1L` and `CCP1CON<5:4>`

(c) Enable the output of CCP1 by clearing `TRISB<0>`

(d) Set the TIMER2 prescale value and enable TIMER2 by writing to `T2CON`

(e) Enable and start PWM by writing to `CCP1CON`.

Remember: at each point, you need to ensure exactly what bit pattern must be sent (use the information given above and the datasheet), and you must be sure you are indeed writing to the correct data location—which bank is each register in?

Look at the result on the RB0/CCP1 pin and confirm that it works. Show it to the TA. Print out your (working) code to include in your report.

6-3 Using the PWM as an D/A converter

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. This makes it easy to use PWM as a "cheap" D/A converter.

Try this out: 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.)

Wire it up, connecting the input to the pin RB0/CCP1. Connect the scope to the output, and have a look. Is the voltage what you expect?

Include a circuit diagram of your RC circuit and draw/explain what you see.

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:

(a) Use a variable to hold a value for `CCPR1L`.
(b) Initialize the variable to 0.
(c) Write the variable to `CCPR1L`.
(d) Start the PWM waveform.
(e) Increment the variable by some amount.
(f) Write the variable to `CCPR1L`.
(g) Wait a sufficient time for the RC filter to settle.
(h) go back to step (e), and repeat.

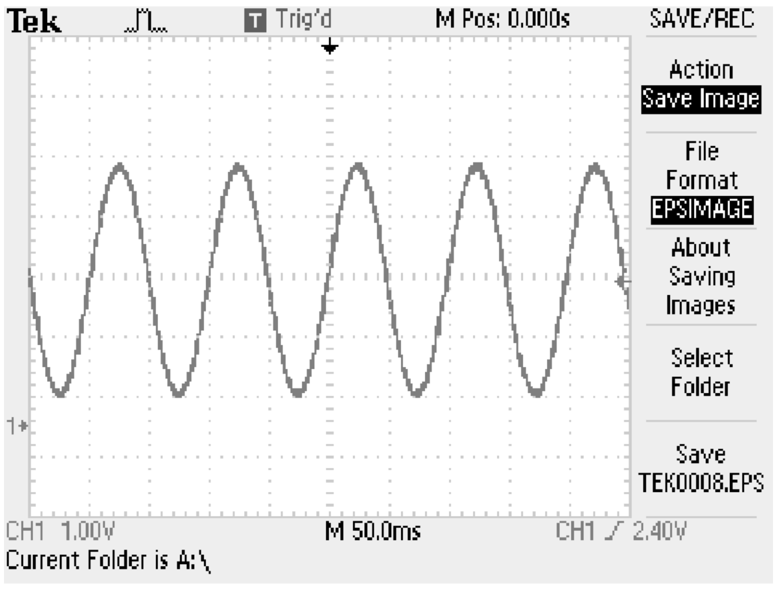
Once your variable reached FFh (255), it will roll over to zero (or a low value, depending on your increment size).

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:

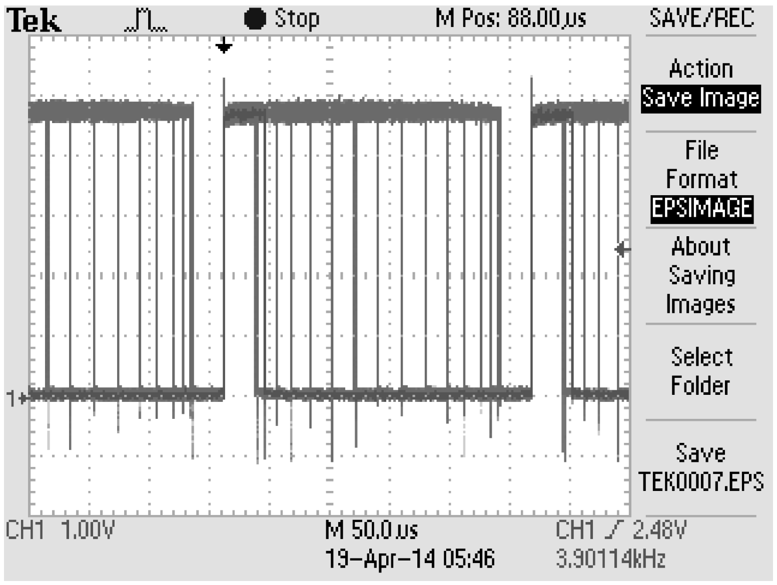
(a) Set a series of `RETLW` statements in a named subroutine. the argument of `RETLW` a literal that sits in W on return.
(b) Write the step number corresponding to which RETLW statement you want into W.
(c) Call the subroutine with the first instruction of the subroutine as: `ADDWF PCL`. This causes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

(Note: 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.)

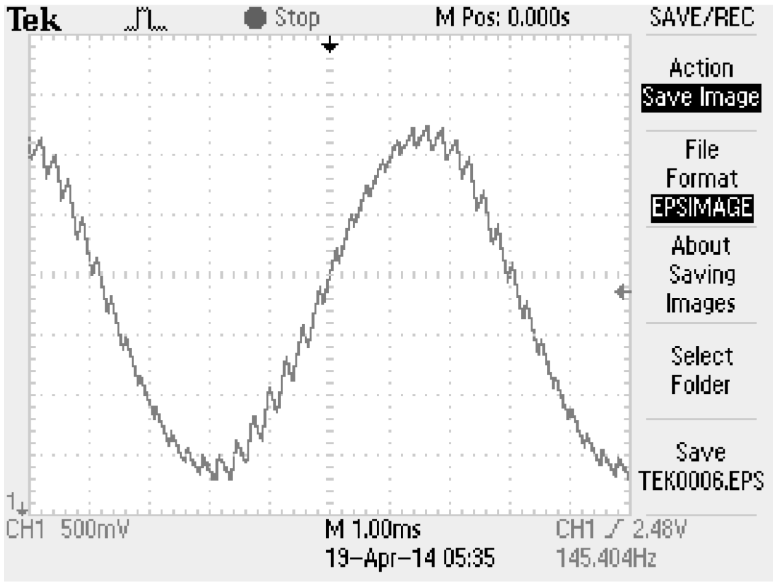
As an example, we used a look-up table to create this sine wave:



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   .0             ; Dummy table value
         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. As usual, include any code with your report.