Timers

- A **timer** on a µC is simply a counter
- The input clock frequency to a timer can be **prescaled** so that it is some fraction of the system clock frequency.
  - This will slow down how fast the timer counts

Timer2 on the PIC18Fxx2 is an 8-bit counter.

Prescaler for TMR2 is 1:1, 1:4, 1:16 of FOSC/4

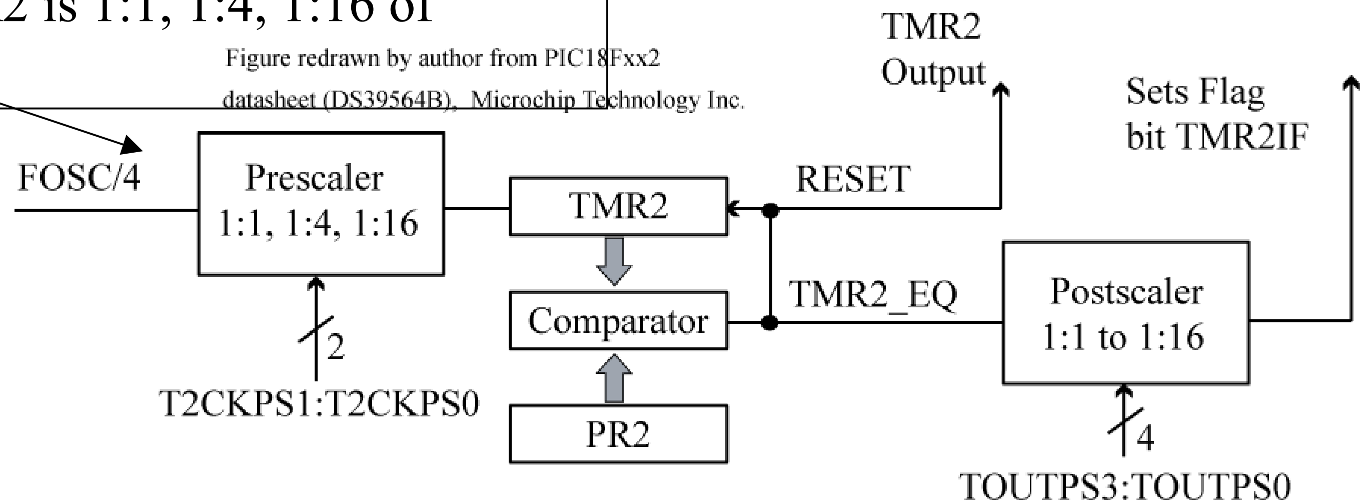
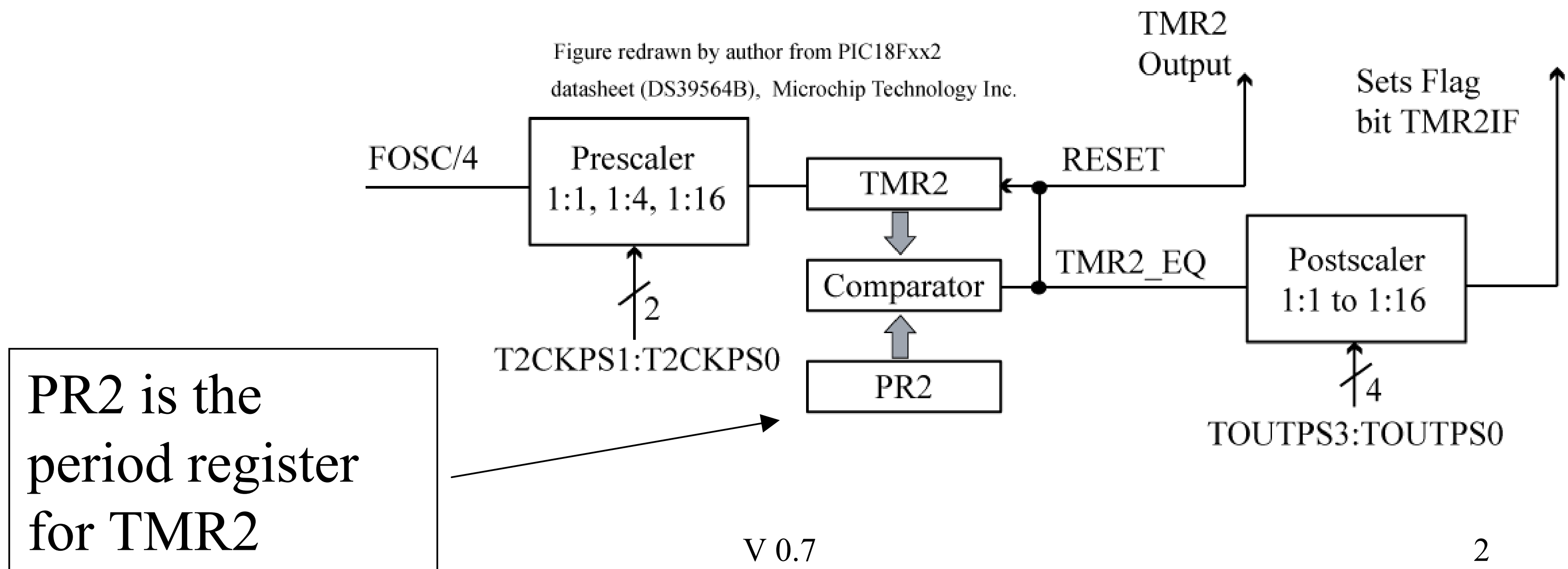


Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Period Register

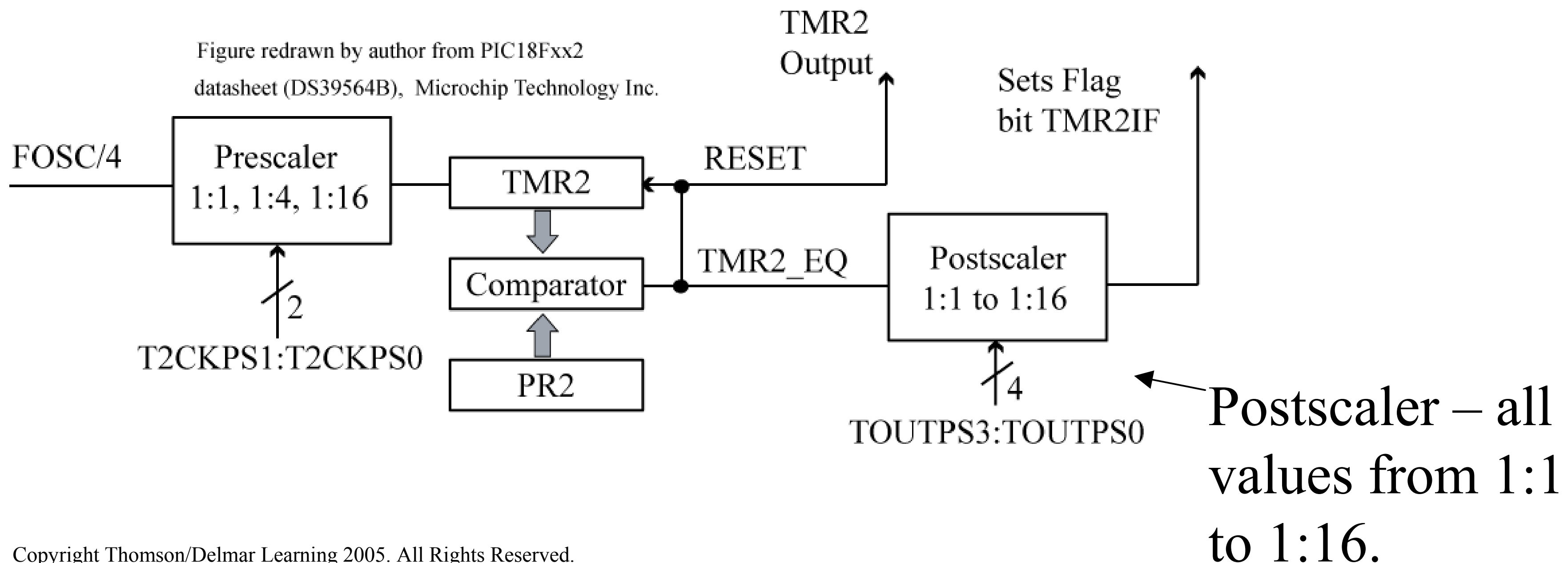
- A timer can be programmed to roll over at any point using the **period** register.
  - An 8-bit timer would typically roll over to 0x00 once it reached 0xFF.
  - However, if the period register is set to 0x7F, then timer will roll over to 0x00 once it increments past 0x7F.



Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Postscaler

- Would like to generate an interrupt when a timer rolls over
- A **Postscaler** is used as a second counter – only generate an interrupt after timer rolls over N times.



Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

PIC18Fxx2 Timer Summary

- Timer0: software selectable as either 8-bit or 16-bit, has a prescaler, clocked by FOSC/4 or external clock.
- Timer1: 16-bit, has a prescaler, used with capture/compare module, clocked by FOSC/4 or external clock (has dedicated oscillator circuit, so can support a second external crystal).
- Timer2: 8-bit, has prescaler/period register /postscaler, used with pulse-width modulation module
- Timer3: A duplicate of Timer 1, shares Timer1's dedicated oscillator circuit.
- Capture/Compare module
  – 16-bit capture register, 16-bit compare register
  – Used with timer1 or timer3 to provide additional time measurement capabilities

The remainder of these notes concentrate on capability associated with Timer2

So....what good are timers?

- Switch Debouncing
- Waveform Generation
- Sampling an input signal with ADC at a fixed frequency
- Pulse Width Measurement
- Pulse Width Modulation
- Many other uses

Computing the Timer2 Interrupt Interval (Chap. 10.8, 10.9)

The equation:

Timer2 interrupt interval = PRE * (PR2+1) * POST * (1/(Fosc/4))

has 3 unknowns: PRE, POST, and PR2.

Use a spreadsheet to calculate. Assume we want a interrupt frequency of 4 KHz,

Interrupt Period = 1/ 4 KHz = 1/ 4000 = 0.00025 = 250 us.

Pick PRE and POST values, and solve the equation for PR2:

PR2 = [(Interrupt Period)/(4 * TOSC * PRE *POST)] - 1

(a) Timer2 solutions for 4 KHz interrupt frequency

FOSC	Pre	Post	Desired Frequency	PR	Actual Frequency	% Error
29491200	1	7	4000	262	4004.8	0.12%
29491200	4	2	4000	229	4007.0	0.17%
29491200	16	1	4000	114	4007.0	0.17%
29491200	1	1	4000	1842	4000.4	0.01%

Invalid, PR2 > 255

Solutions for Timer2 Interrupt Frequency of 4 KHz (Period = 250 us)

(b) Timer2 solutions for 4 KHz interrupt frequency, increasing postscaler value

FOSC	Pre	Post	Desired Frequency	PR	Actual Frequency	% Error
29491200	1	7	4000	262	4004.8	0.12%
29491200	1	8	4000	229	4007.0	0.17%
29491200	1	9	4000	204	3996.1	-0.10%
29491200	1	10	4000	183	4007.0	0.17%
29491200	1	11	4000	167	3989.6	-0.26%
29491200	1	12	4000	153	3989.6	-0.26%
29491200	1	13	4000	141	3993.9	-0.15%
29491200	1	14	4000	131	3989.6	-0.26%
29491200	1	15	4000	122	3996.1	-0.10%

If PR2 > 255, then invalid solution as PR2 is an 8-bit register

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Timer2 Configuration

T2CON: Timer2 Control Register

7	6	5	4	3	2	1	0
-- u --	TOUTPS3	TOUTPS2	TOUTPS3	TOUTPS3	TMR2ON	T2CKPS1	T2CKPS0

-- u -- : unimplemented

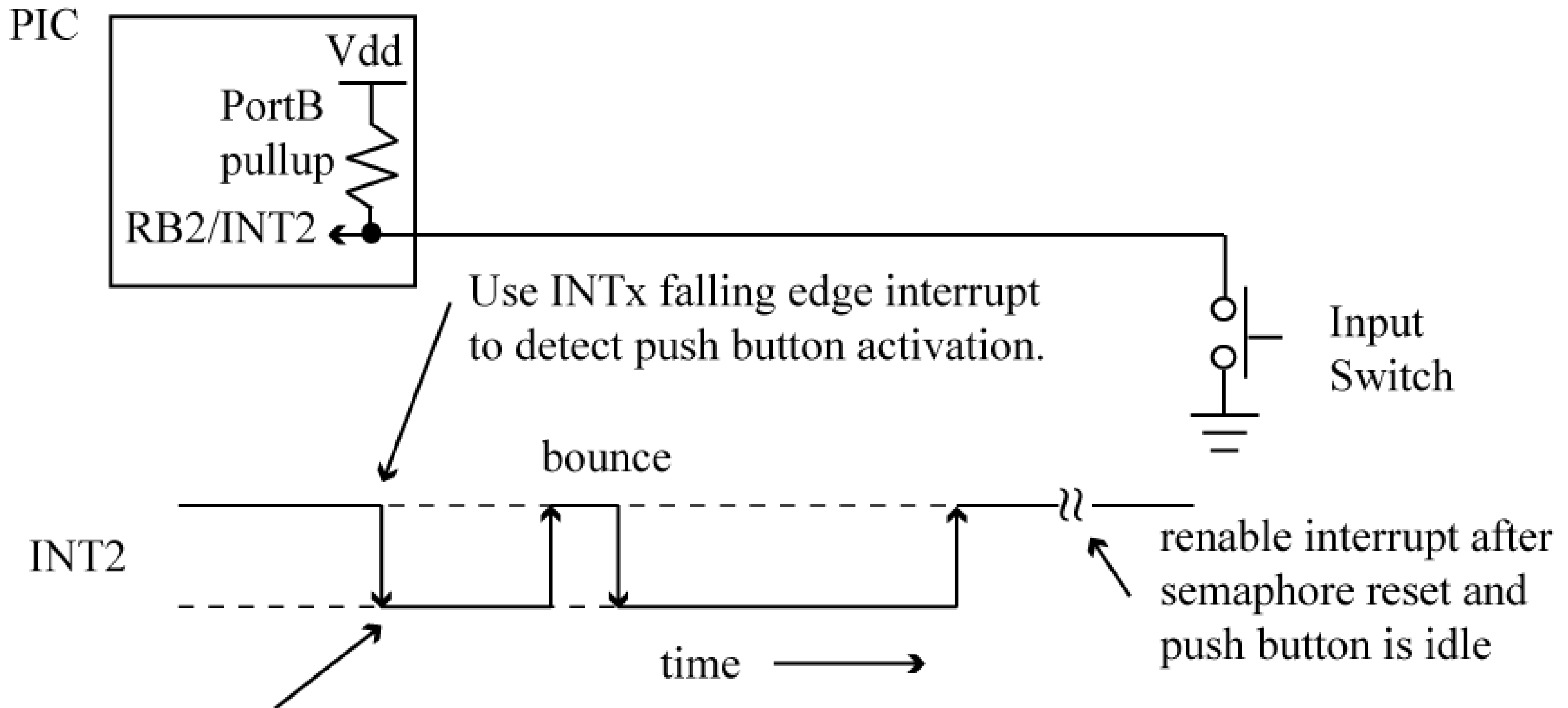
TOUTPS3:TOUTPS2 Postscale Select
- 0000 = 1:1 Postscale
- 0001 = 1:2 Postscale
- 0010 = 1:3 Postscale
- ....
- 1110 = 1:15 Postscale
- 1111 = 1:16 Postscale

TMR2ON: Timer2 On Bit (1 is on, 0 is off)

T2CKPS1: T2CKPS0: Timer2 Clock Prescale
- 00 = Prescaler is 1
- 01 = Prescaler is 4
- 1x = Prescaler is 16

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

Switch Debounce Using Timer2 (Chap. 10.9)



Use Timer2 periodic interrupt to allow switch to settle

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

ISR for Switch Debounce

```
#define DEBOUNCE   5
volatile unsigned char button, button_debounce;

void interrupt pic_isr(void){
 if (INT2IF && INT2IE) {
  // pushbutton detected
  INT2IE = 0;  button = 1;  button_debounce = 0;
 }
 if (TMR2IF) { // debouncing timer
  TMR2IF = 0;
  if (!INT2IE) {
   if (RB2) {
    if (button_debounce != DEBOUNCE)
      button_debounce++;
    }
   else button_debounce=0;
   if (button_debounce == DEBOUNCE && !button){
    //button idle high ,re-enable interrupt
    INT2IF=0; INT2IE=1;
   }
  }
 }
}
```

5 * 6 ms = 24 to 30 ms debounce time

Falling edge interrupt, set the semaphore and disable interrupt

If the interrupt disabled, then debounce the switch by waiting for it to become idle high. After the switch is debounced and the semaphore acknowledged, then reenable the interrupt.

Assume Timer2 configured for 6 ms interrupt period.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

main() for Switch Debounce

```
main(void){
 serial_init(95,1); // 19200 in HSPLL mode, crystal = 7.3728 MHz
 // configure INT2 for falling edge interrupt input
 TRISB2 = 1;  INT2IF = 0; INTEDG2 = 0;  INT2IE = 1;
 RBPU = 0; // enable weak pullup on port B
 // configure timer 2
 // post scale of 11, prescale 16, PR=250, FOSC=29.4912 MHz
 // gives interrupt interval of ~ 6 ms
 TOUTPS3 = 1; TOUTPS2 = 0; TOUTPS1 = 1; TOUTPS0 = 0;
 T2CKPS1 = 1;  PR2 = 250;
 // enable TMR2 interrupt
 IPEN = 0;  TMR2IF = 0; TMR2IE = 1;   PEIE = 1;  GIE = 1;
 TMR2ON = 1 ;
 pcrlf();  printf("Pushbutton with Timer2 Debounce");  pcrlf();
 while(1) {
  if (button) {
   button=0; // acknowledge this semaphore
   printf("Push Button activated!"); pcrlf();
  }
 }// end while
}//end main
```

Configure INT2 for falling edge interrupt

Configure Timer2 for ~6 ms interrupt period

If pushbutton activated, then print message, reset the semaphore

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

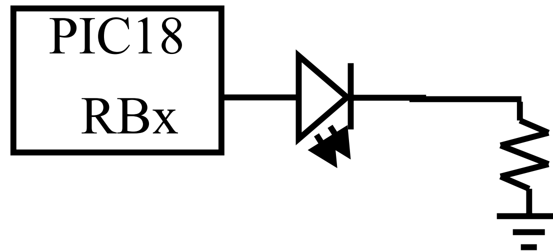
Capture/Compare/PWM Module (Chap 13.7)

- Each CCP Module contains
  - 16-bit Capture Register, 16-bit Compare Register
  - PWM Master/Slave Duty Cycle Register
- Pulse Width Modulation (PWM) mode is used to produce a square wave without processor intervention
  - Uses timer2 resource, and Compare register
  - Square wave on output pin 100% hardware generated, no software intervention
  - Can vary the duty cycle via the Timer2 PR2 register

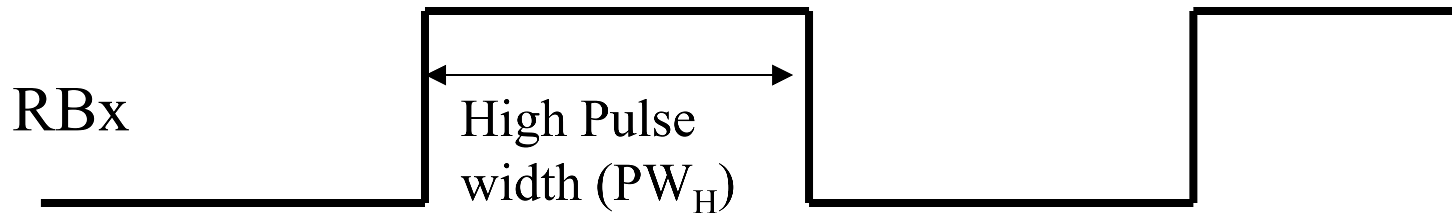
The remainder of these nodes discuss PWM mode, waveform generation using period interrupts. Capture/Compare is discussed later.

Pulse Width Modulation

Pulse Width Modulation (PWM) is a common technique for controlling average current to a device such as a motor.

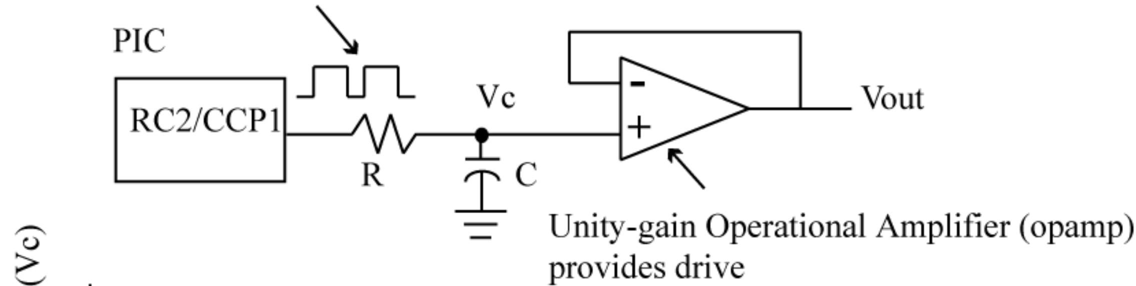


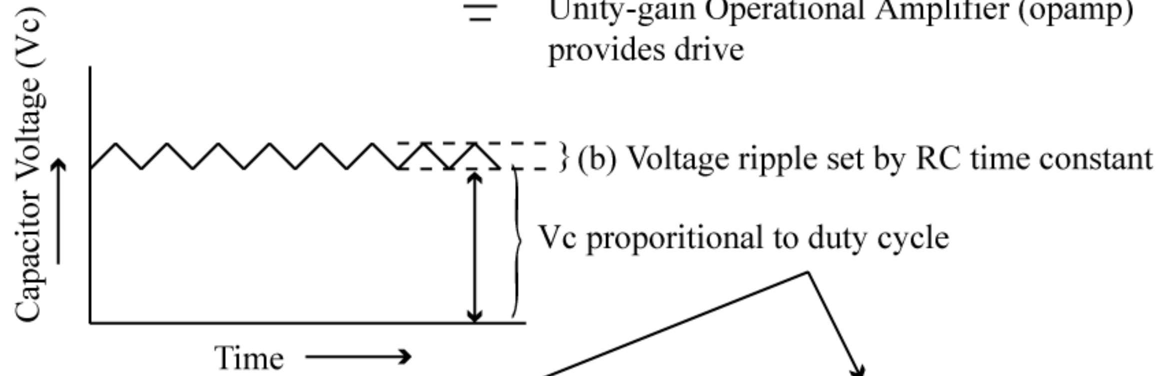
Uses a square wave with **fixed frequency**, **varying duty cycle**.

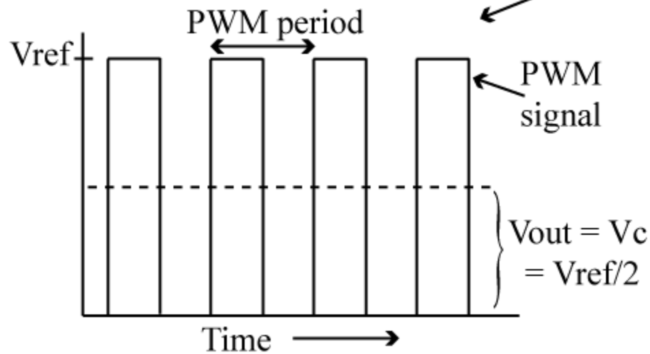


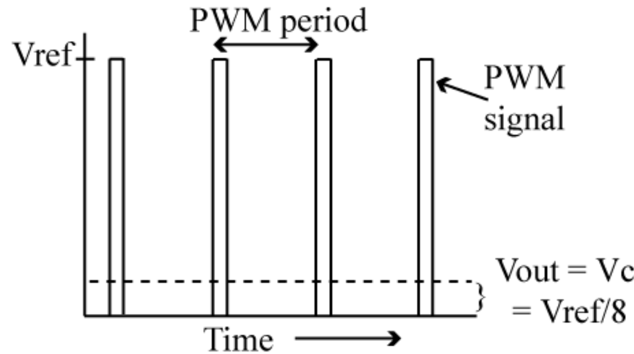
For a fixed frequency, the brightness of the LED will vary directly with duty cycle. The higher the duty cycle (the longer the high pulse width), the brighter the LED because it stays on longer (more current delivered to the LED)

PWM DAC





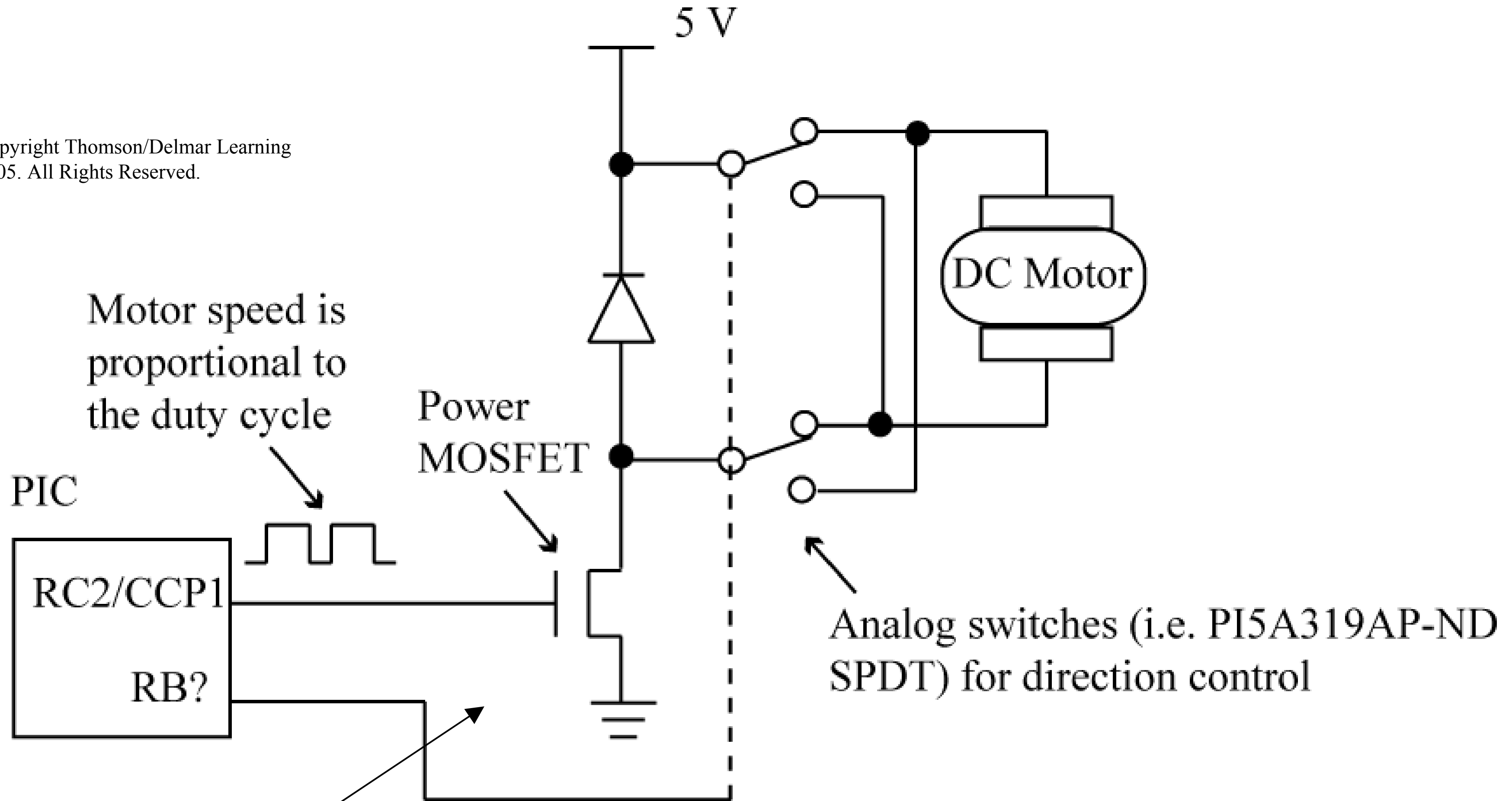




Can build a low-component count DAC by using PWM to control the voltage across a capacitor. The capacitor voltage varies linearly with the duty cycle.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

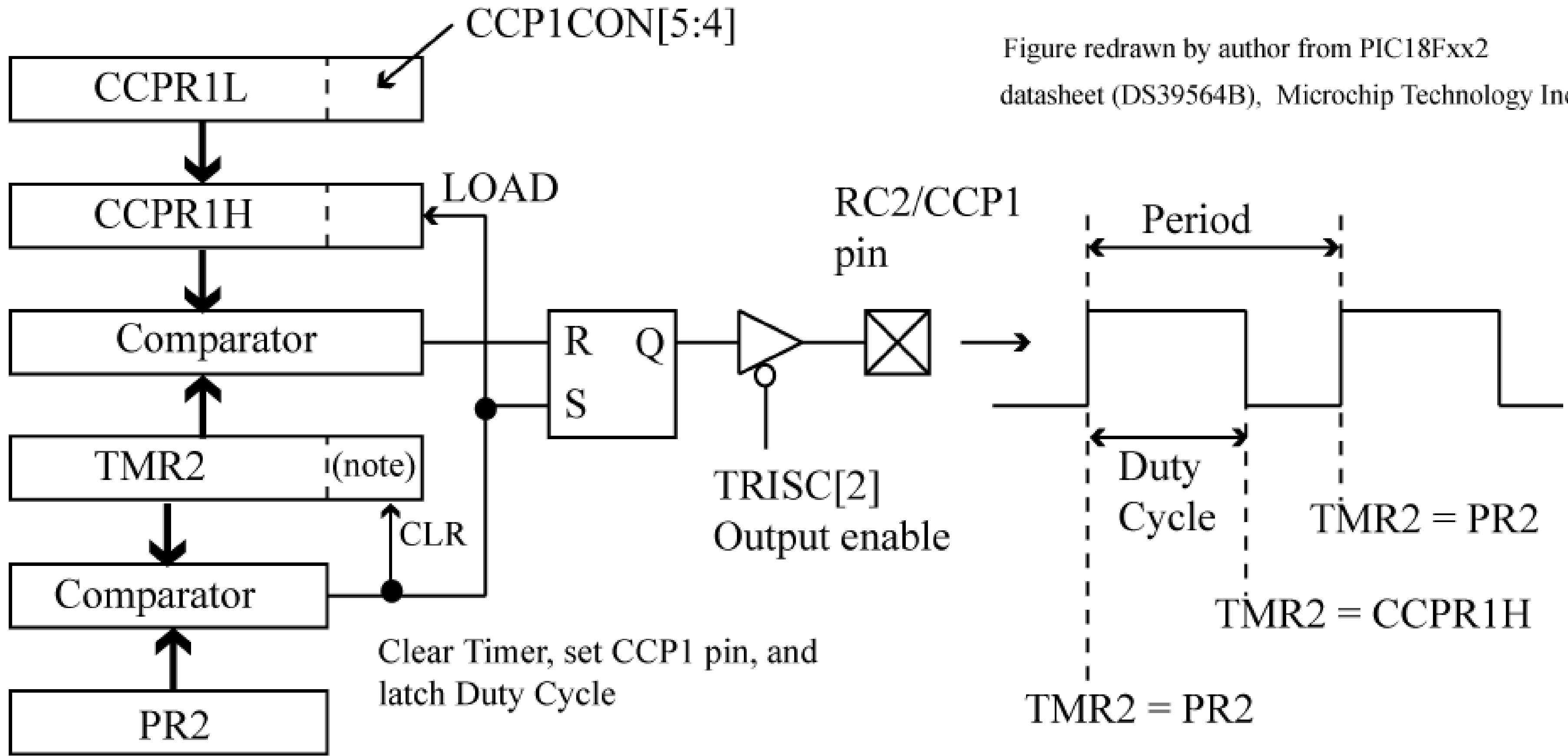
PWM Motor Control



Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

The voltage on the gate of the MOSFET varies linearly with the duty cycle (the gate of the MOSFET looks like a capacitor). This in turn varies the current through the MOSFT, controlling the motor speed. The MOSFET is NOT rapidly turning off/on.

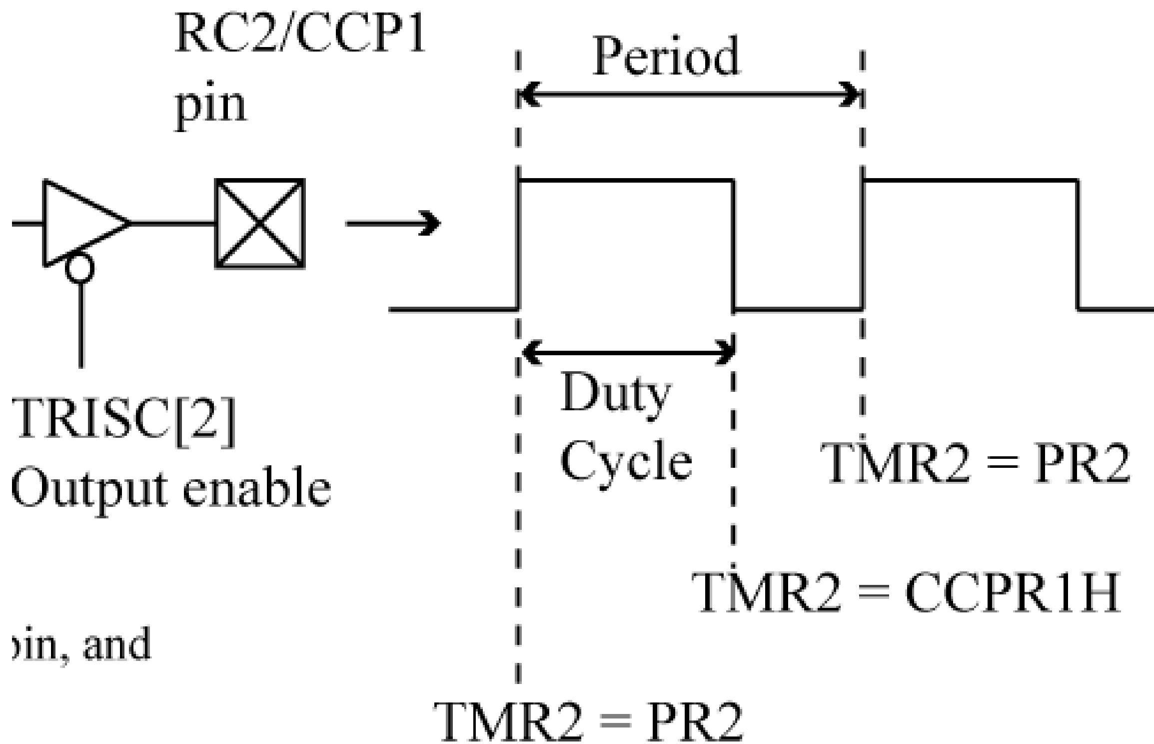
PIC18Fxx2 PWM



Note: 8-bit timer is concatenated with 2-bit internal Q clock or 2 bits of the prescaler to create 10-bit time-base

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

PIC18Fxx2 PWM Period

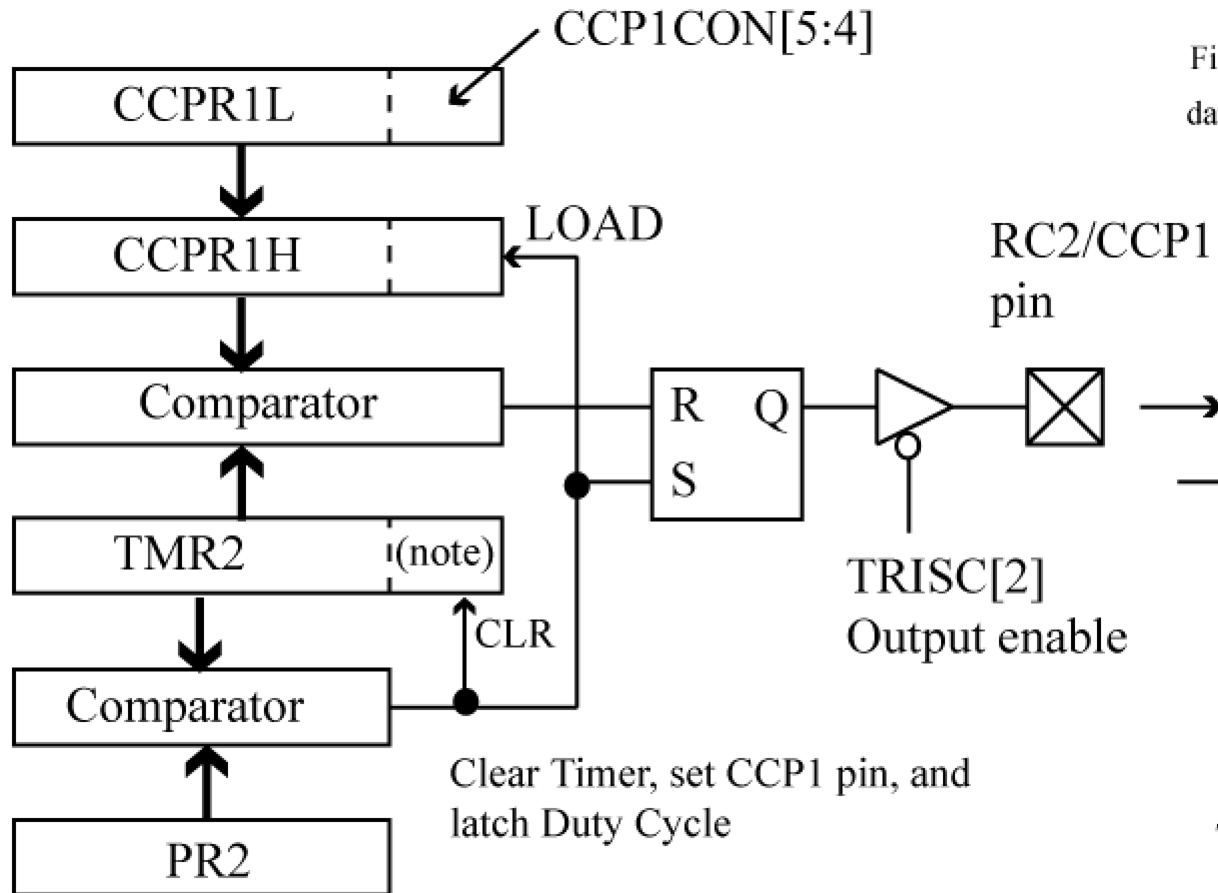


Period = (PR2+1) * 4 * (1/Fosc) * TMR2_Prescale

Note that when TMR2 is used for PWM, the postscaler is NOT used.

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

PIC18Fxx2 PWM Duty Cycle



Duty cycle has 10-bit resolution, upper 8-bits in CCPR1L, lower two bits are CCP1CON<5:4>

CCPR1H used to double buffer the PWM operation.

When TMR2=PR2, output SET, TMR2 reset to 0.

When TMR2 = CCPR1H, then output RESET

Copyright Thomson/Delmar Learning 2005. All Rights Reserved.

PIC18Fxx2 PWM Duty Cycle

Duty Cycle = CCPR1L:CCPCON<5:4> * (1/Fosc) * TMR2_prescale

↑ 10 bits.

Recap: Period defined by PR2, duty cycle by CCPR1L + 2 bits

The duty cycle time should be less than the period, but this is NOT enforced in hardware.

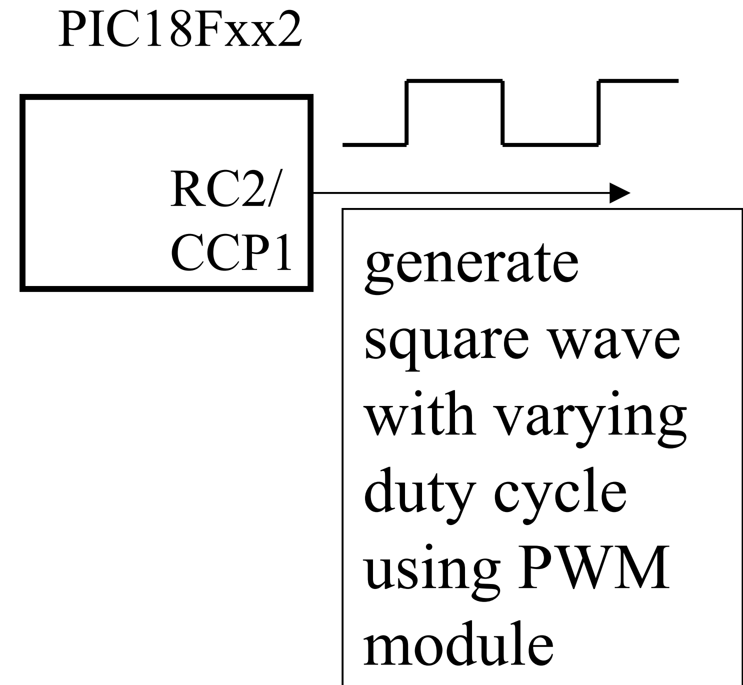
If duty cycle > Period, then the output will always be a high (will never be cleared).

In our calculations, will ignore lower 2-bits of duty cycle and only use 8-bits that are in CCPR1L.

PWM Test: sqwave.c

Generate a square wave using TMR2 and the PWM capability.

```
// configure timer 2
// post scale of 1

TOUTPS3 = 0; TOUTPS2 = 0;
TOUTPS1 = 0;  TOUTPS0 = 0;

// pre scale of 4

T2CKPS1 = 0;  T2CKPS0 = 1;

// start timer 2

TMR2ON = 1 ;

PR2 = 255;  // set timer2 PR register
CCPR1L = (255 >> 1); // 255/2 = 50% duty cycle
bitclr(CCP1CON, 5); // lower two bits are 00
bitclr(CCP1CON, 4);
```



sqwave.c (cont)

```
// set CCP1 output
 TRISC2 = 0;
```

Pin RC2 must be configured as output.

```
 // PWM Mode
 bitset(CCP1CON, 3);
 bitset(CCP1CON, 2);
```

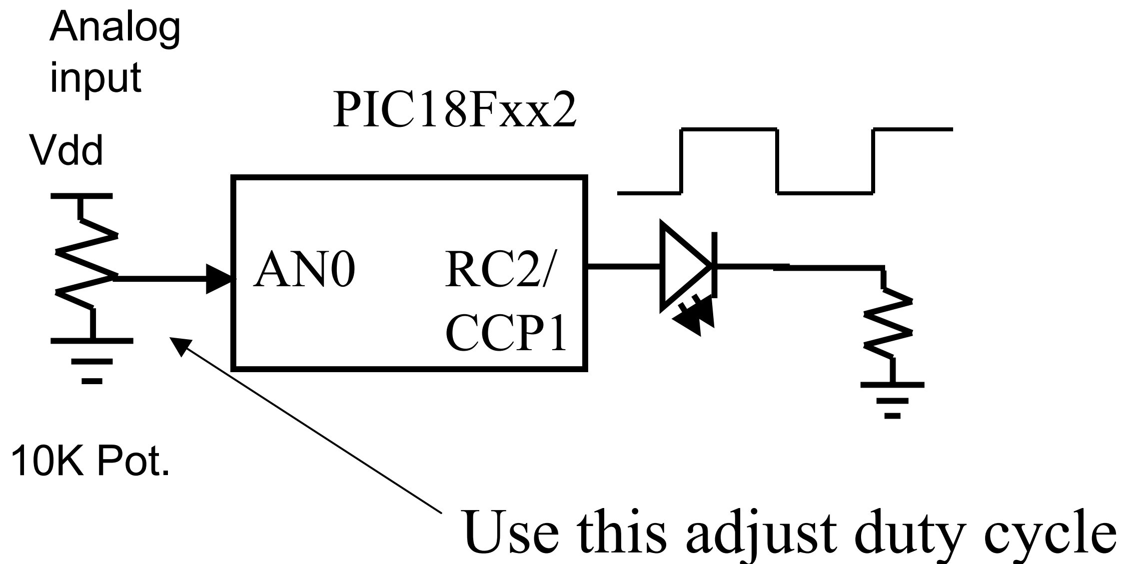
Configures Capture/Compare/PWM module for PWM operation

```
 while(1) {
   // prompt user PR2, Prescale
   // values, code not shown.
   }
}
```

At this point, the square wave is active, no other intervention necessary.

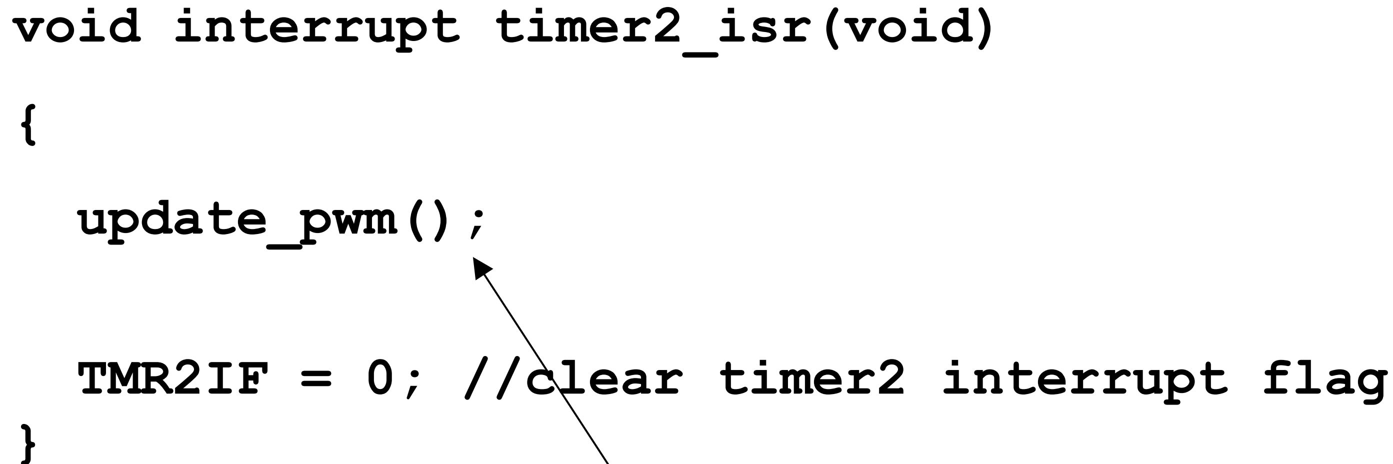
PWM test: ledpwm.c

Use potentiometer and PIC18Fxx2 ADC to adjust duty cycle for PWM to LED



Will initialize PWM module in the same way as before, except TMR2 interrupt will be enabled. In ISR, read ADC value and update duty cycle.

ledpwm.c (cont)



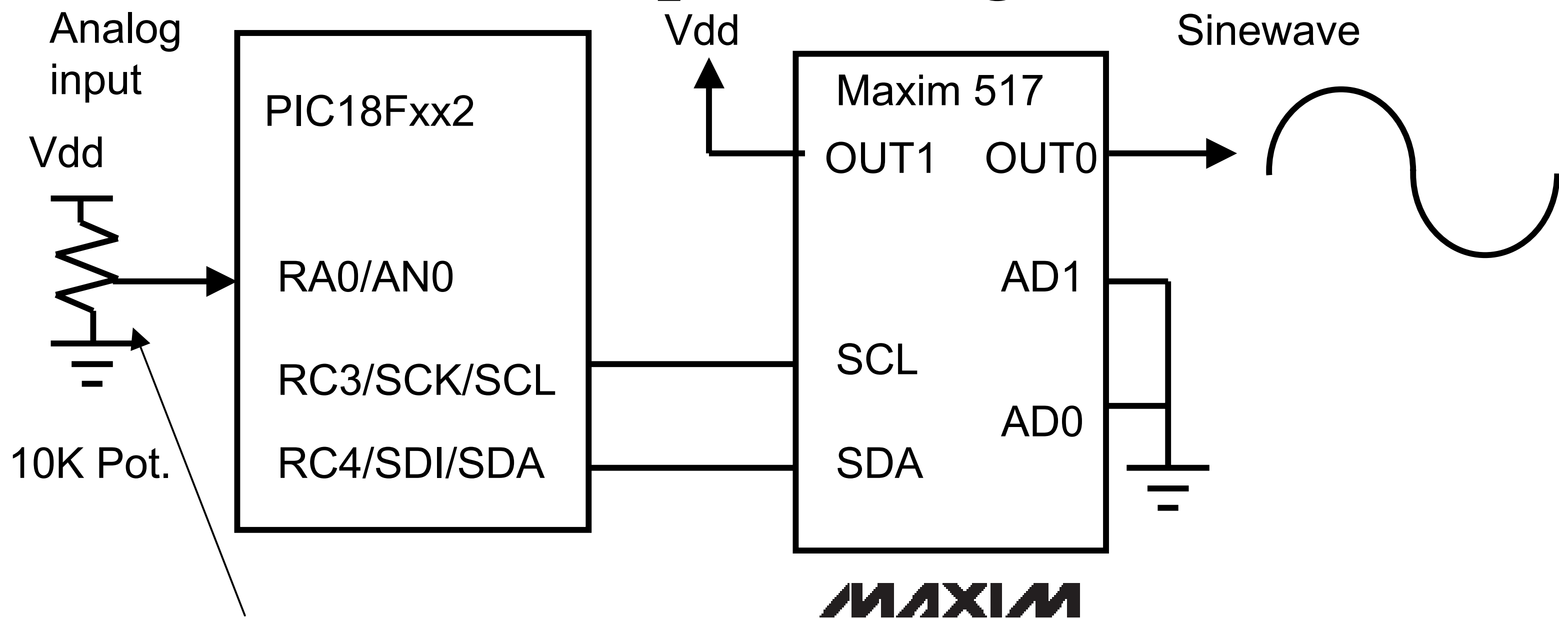
```
void interrupt timer2_isr(void)
{
  update_pwm();

  TMR2IF = 0; //clear timer2 interrupt flag
}
```

This subroutine does work of reading A/D and changing the duty cycle.

ledpwm.c (cont)

```
update_pwm(){
  unsigned char rval1;

  rval1 = ADRESH; // A/D value left justified
  CCPR1L = rval1; // upper 8 bits of duty cycle
  rval1 = ADRESL;
  // update lower two bits of duty cycle
  if (bittst(rval1,7))   bitset(CCP1CON, 5);
      else bitclr(CCP1CON, 5);

  if (bittst(rval1,6))  bitset(CCP1CON, 4);
      else bitclr(CCP1CON, 4);

  // start new conversion

  GODONE = 1; // start conversion

}
```

Waveform Generation using Periodic Interrupts: sinegen.c



Potentiometer used to vary the PR2 timer value between a min value of 25 and max value of 100. This varies TMR2 interrupt time. Sinewave is produce by table look-up (either 16-entry table or 64-entry table). Period of Sinewave is number_table_entries * tmr2_interrupt_interval.

16 Entry Table

```
const unsigned char sine16tab[] =
{0x80,0xb0,0xda,0xf6,0xff,0xf6,0xda,0xb0,0x80,0x4f,
0x25,0x9,0x0,0x9,0x25,0x4f};
```

Period

TMR2 interrupt interval. New entry read from table each interrupt.

Period = 16 * TMR2 interrupt interval.

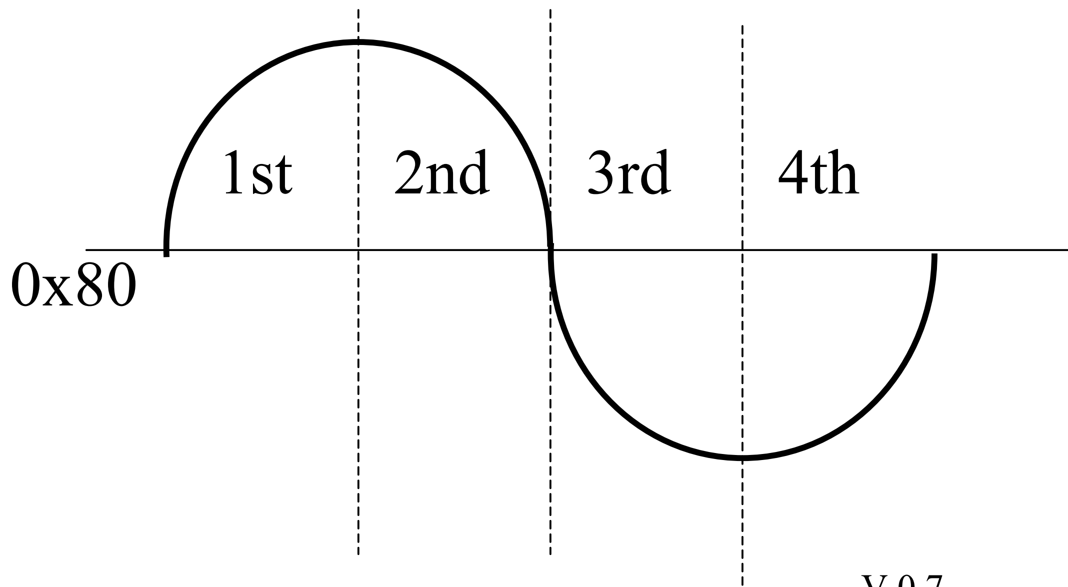
300
250
200
150
100
50
0

128 176 218 246 255 246 218 176 128 79 37 9 0 9 37 79

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Saving Data Space

- 64-entry table is same as 16-entry, just finer grain
  - Both end up being stored in File Register memory (Bank1)
- Could save File Register memory by noting that really only have to save ¼ of the waveform



2nd quarter is 1st quarter read in reverse order.

3rd quarter = 1st – 0x80

4th quarter = 1st – 0x80 read in reverse order

sinegen.c (main loop)

```
while(1) {
 if(dac_flag) {
   if (tabmax == 16)
     update_dac(sine16tab[sineptr]);
   else  update_dac(sine64tab[sineptr]);
   dac_flag = 0;
   if (pr_flag) {
     update_period();
     pr_flag = 0;
   }
 }
}
```

dac_flag set by ISR

tabmax chooses between 16-entry, 64-entry tables.

Read value out of table, update DAC.

pr_flag set by ISR when at end of period (16 or 64)

Read ADC to get new TMR2 interval, update PR2

sinegen.c (ISR)

```
void interrupt timer2_isr(void)
{
  sineptr++;
  if (sineptr == tabmax){
    sineptr = 0;
    pr_flag = 1;
  }
  dac_flag = 1;
  TMR2IF = 0; clear timer interrupt flag

}
```

Move to next entry in table

Reset if at end of table

Signal to main loop that end of period reached, need to update period

Signal to main loop that interrupt occurred.

Clear TMR2 interrupt flag so interrupt can happen again.

sinegen.c (update_period)

Read A/D, get upper 7 bits, clip to between MIN/MAX to limit range of interrupt period. ADC configured for left justification.

```
update_period(){
// read ADC result,
// use as period, start new conversion
  rval = ADRESH >> 1; // only keep upper 7 bits
  if (rval < MIN_PERIOD) rval = MIN_PERIOD;
  if (rval > MAX_PERIOD) rval = MAX_PERIOD;
  PR2 = rval;  // set timer2 period register
  // start new conversion
  GODONE = 1;
}
```

Set new PR2 value, start new conversion.

Experiment 11 Task: Waveform Generation

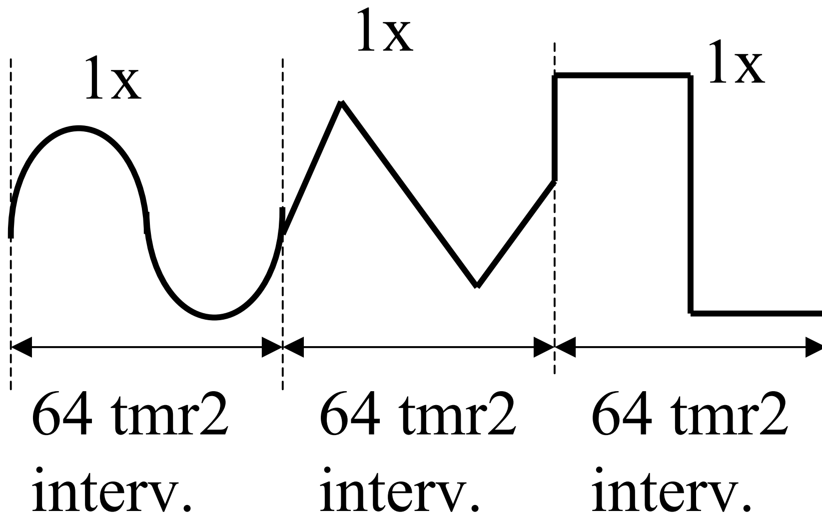
Generate a waveform that consists of 1 or more sine wave cycles, 1 or more triangle wave cycles, 1 or more square wave cycles.

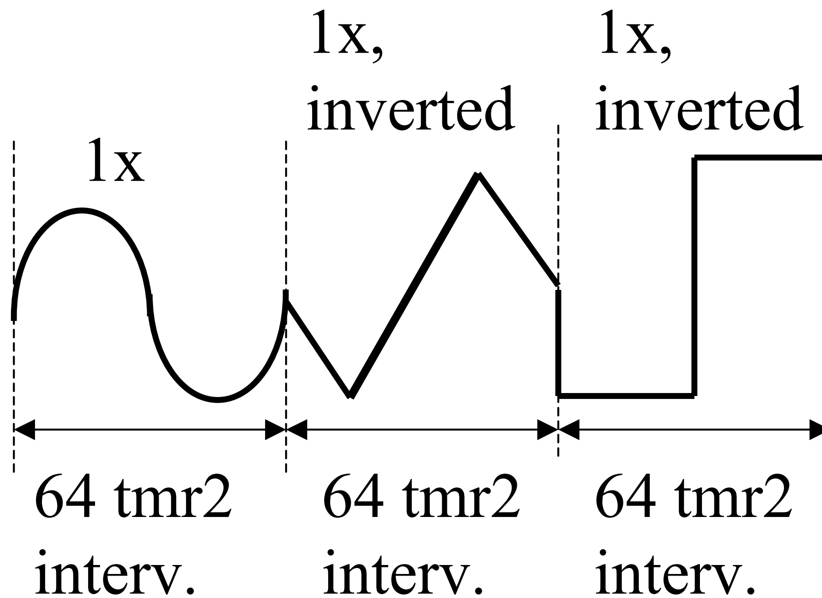
A 1x *period* is defined as 64 * TMR2 interrupt interval – sine wave will ALWAYS have a 1x period (use 64 table lookup).

Square wave and Triangle waves can have a period of 0.5X (32 intervals), 1.0X (64 intervals) or 2.0X (128 intervals).

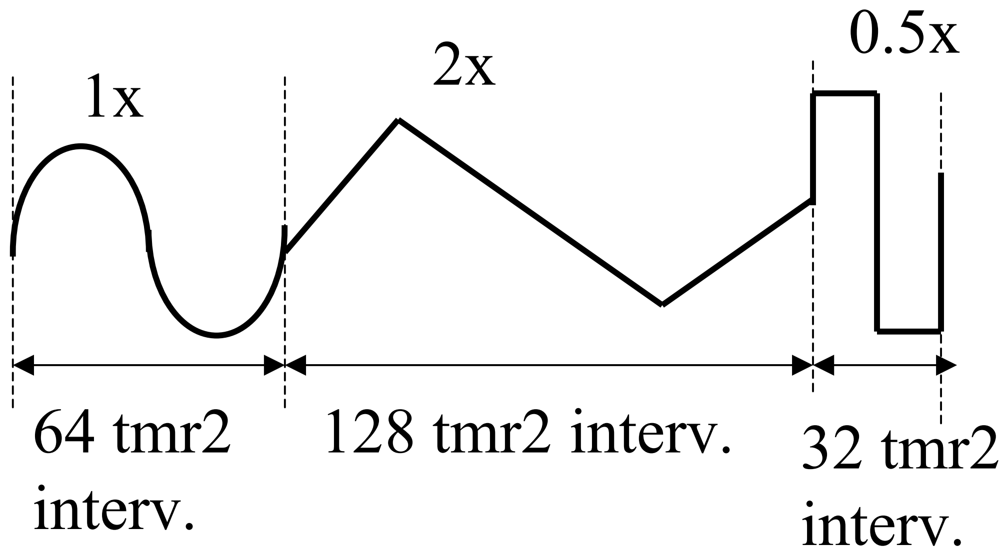
Square wave/Triangle wave can also be inverted.

Waveform Examples





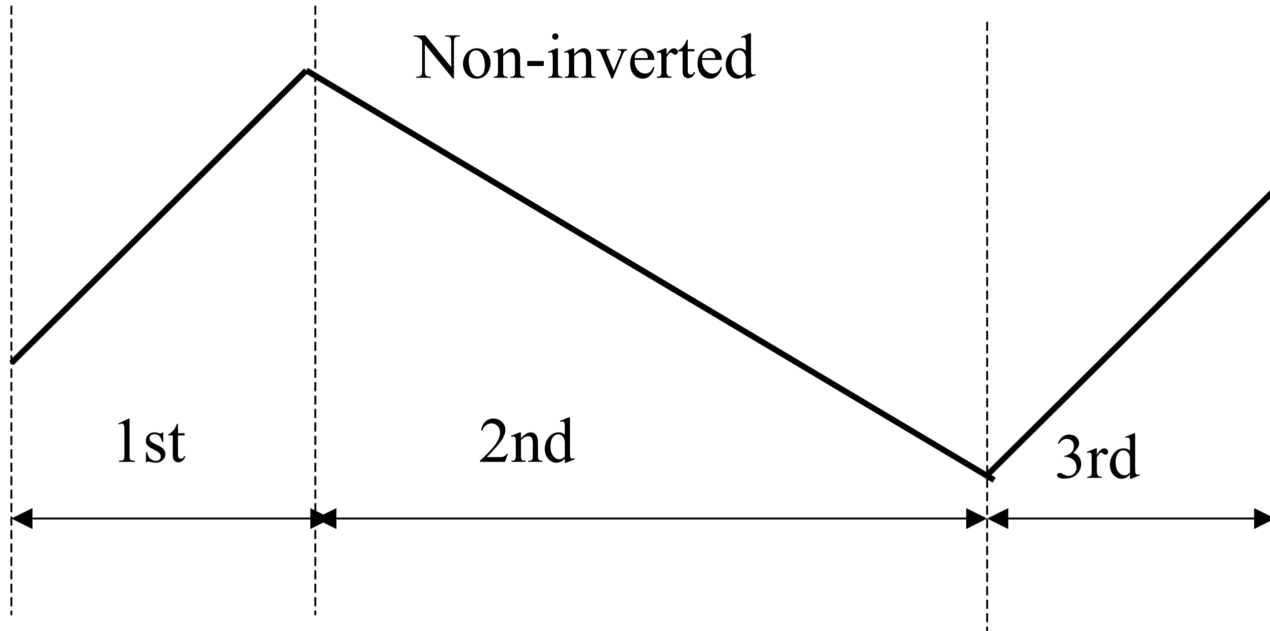
Waveform Examples



Your assigned waveform is fixed in terms of type of component waveforms, and inversion.

To generate square wave/triangle wave – can use table lookup if desired as long as tables are stored in Program Memory using the ‘const’ modifier. As an alternative, square wave and triangle wave values are easy to ‘compute’ based on current value and sample number.

Triangle Wave Computation



In 1st, 3rd sections: **`new_value = old_value + delta`**

In 2nd section: **`new_value = old_value - delta`**

Delta depends on number of points (period) of waveform.

Be careful of wrap-around at low/high points (0x00, 0xFF).