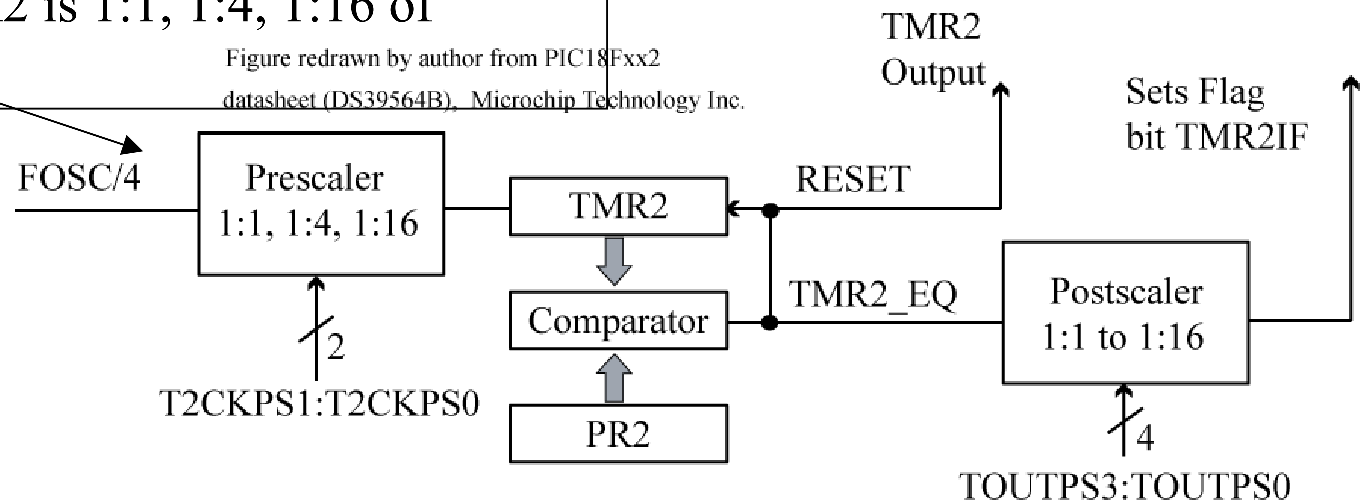
Timers

- 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

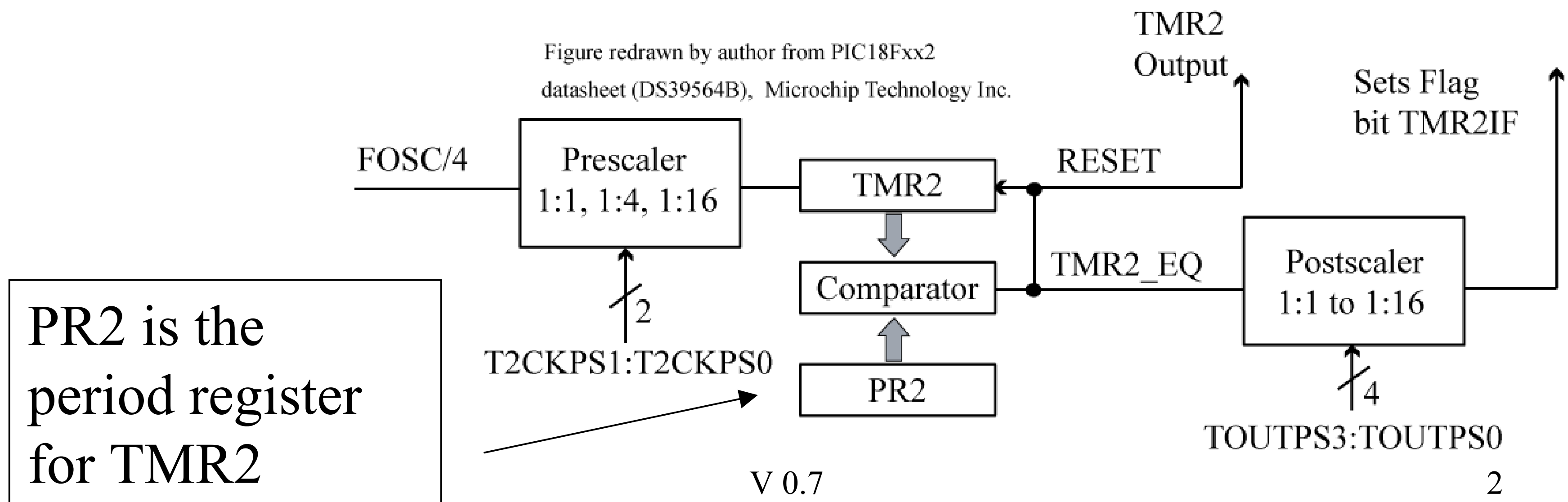
Timer2 on the PIC18 is an 8-bit counter.

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



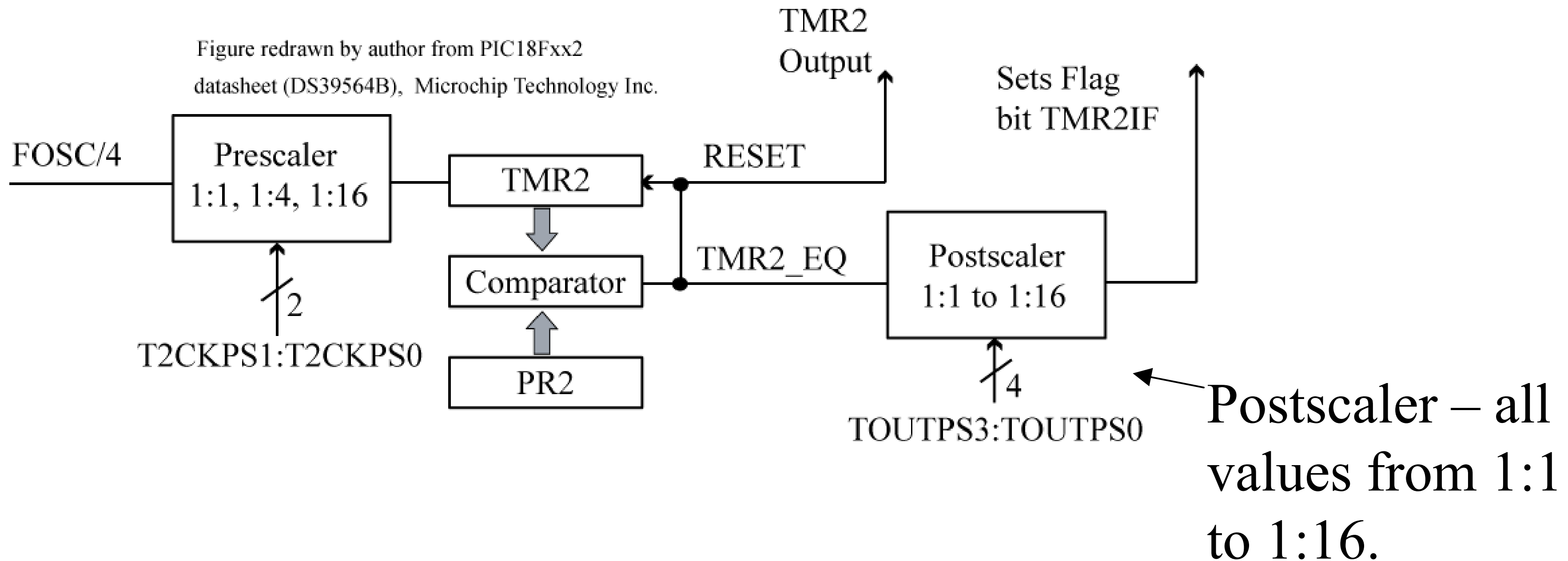
Period Register

- 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.



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.



PIC18 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

Timer2 Configuration

T2CON: Timer2 Control Register

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

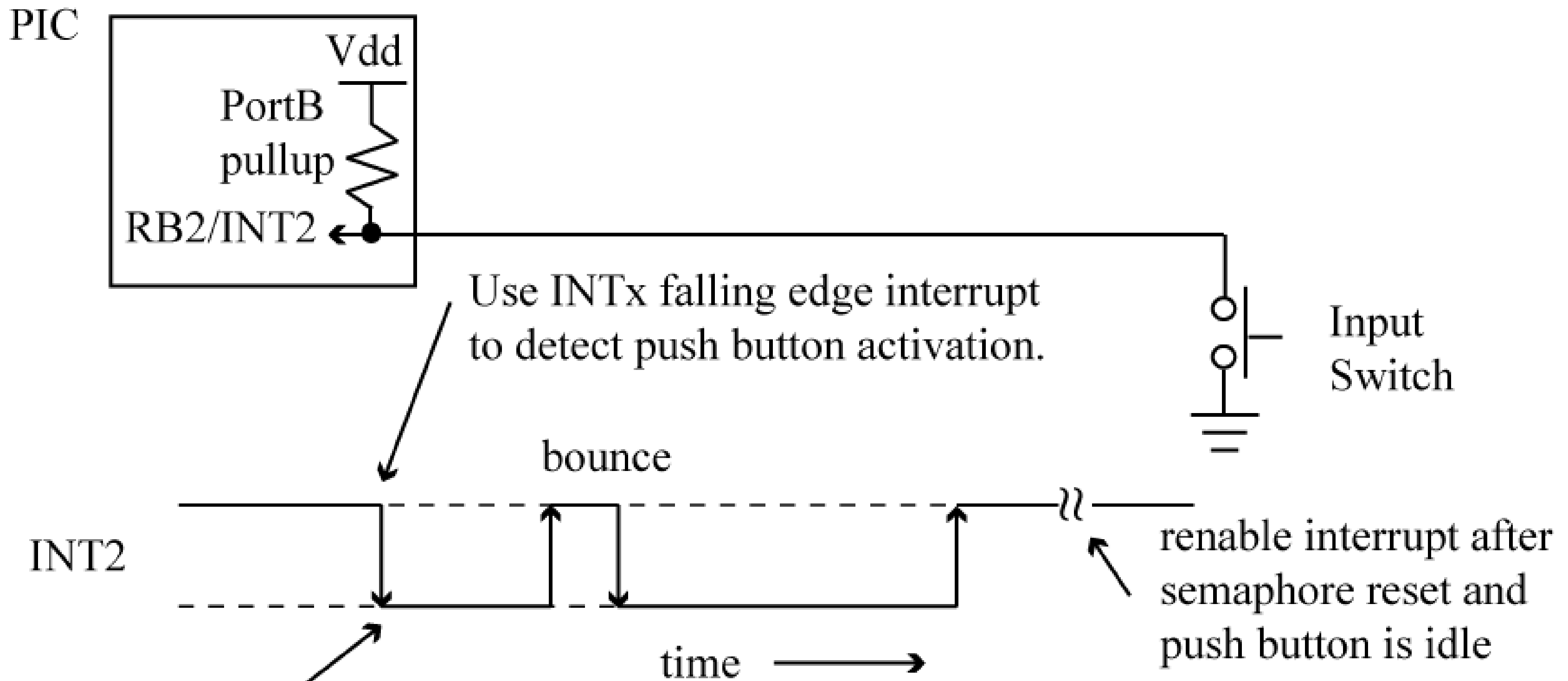
-- u -- : unimplemented

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

Switch Debounce Using Timer2 (Chap. 10.9)



Use Timer2 periodic interrupt to allow switch to settle

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.

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

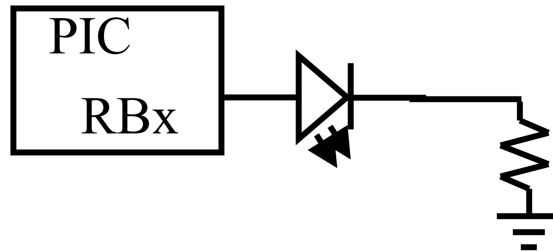
Capture/Compare/PWM Module (Chap 13.7)

- 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

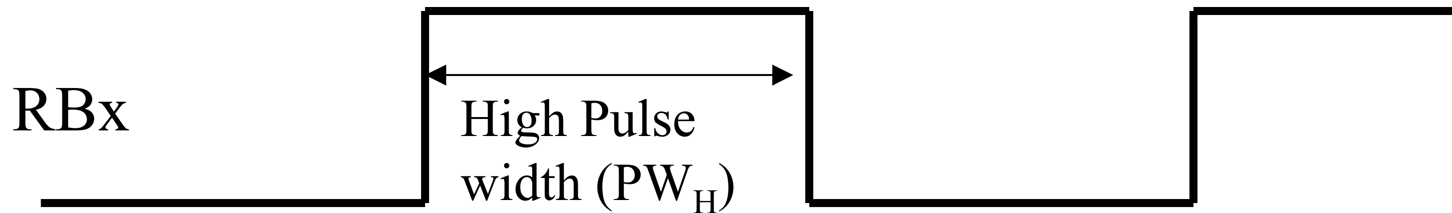
The remainder of these nodes discuss PWM mode, waveform generation using period interrupts. Capture/Compare is discussed later.

Pulse Width Modulation

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

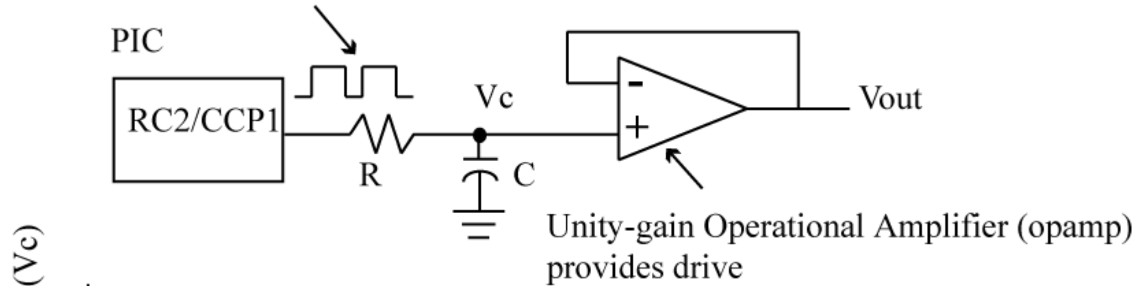


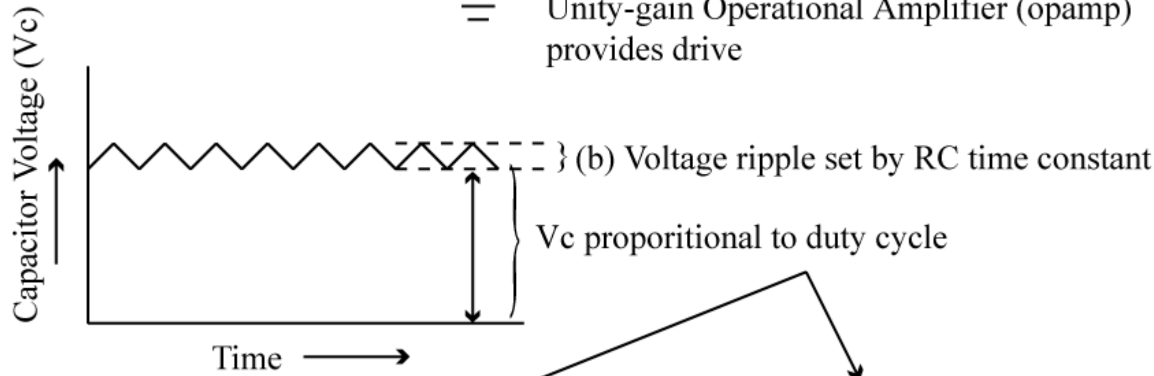
Uses a square wave with **fixed frequency**, **varying duty cycle**.

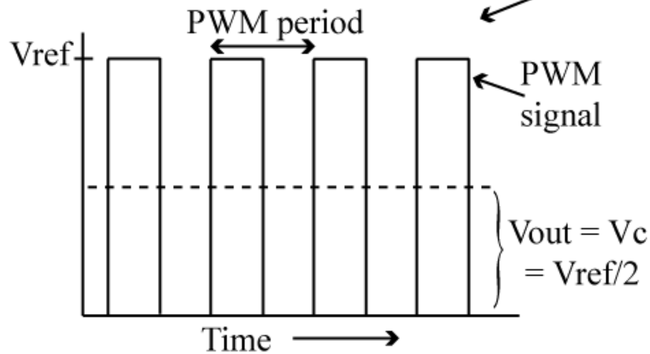


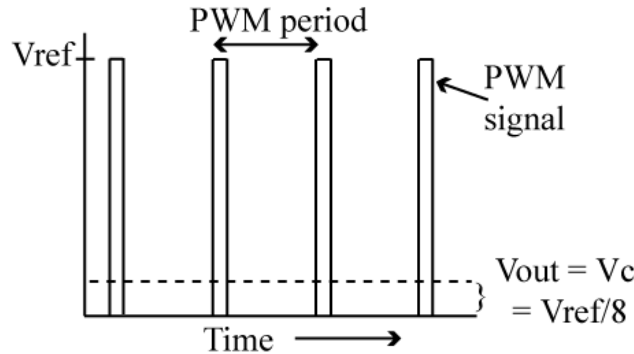
For a fixed frequency, the brightness of the LED will vary directly with duty cycle. The higher the duty cycle (the longer the high pulse width), the brighter the LED because it stays on longer (more current delivered to the LED)

PWM DAC



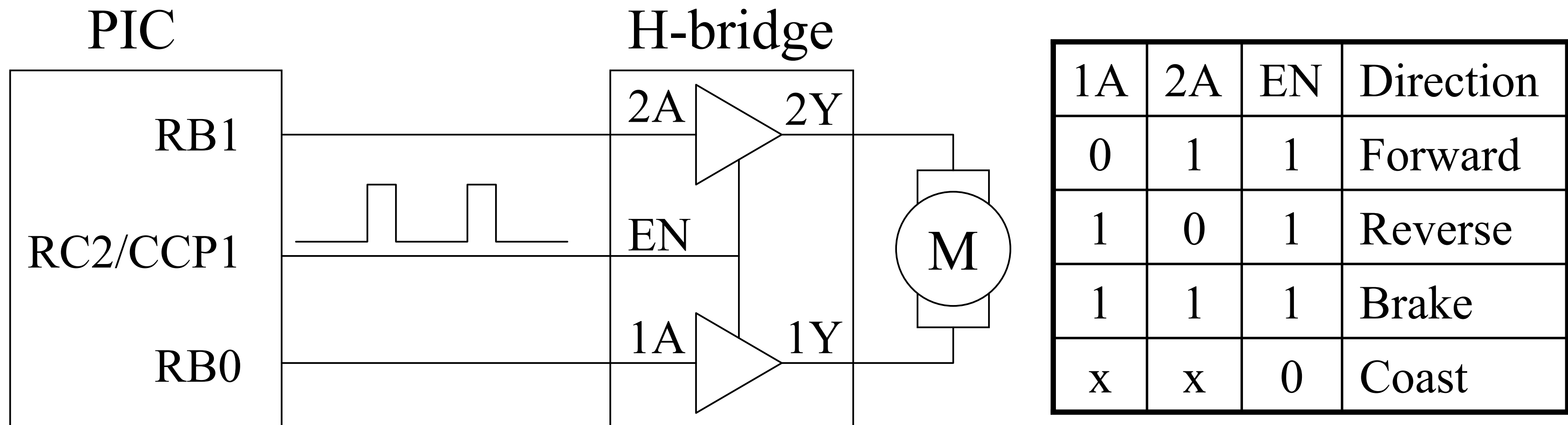






Can build a low-component count DAC by using PWM to control the voltage across a capacitor. The capacitor voltage varies linearly with the duty cycle.

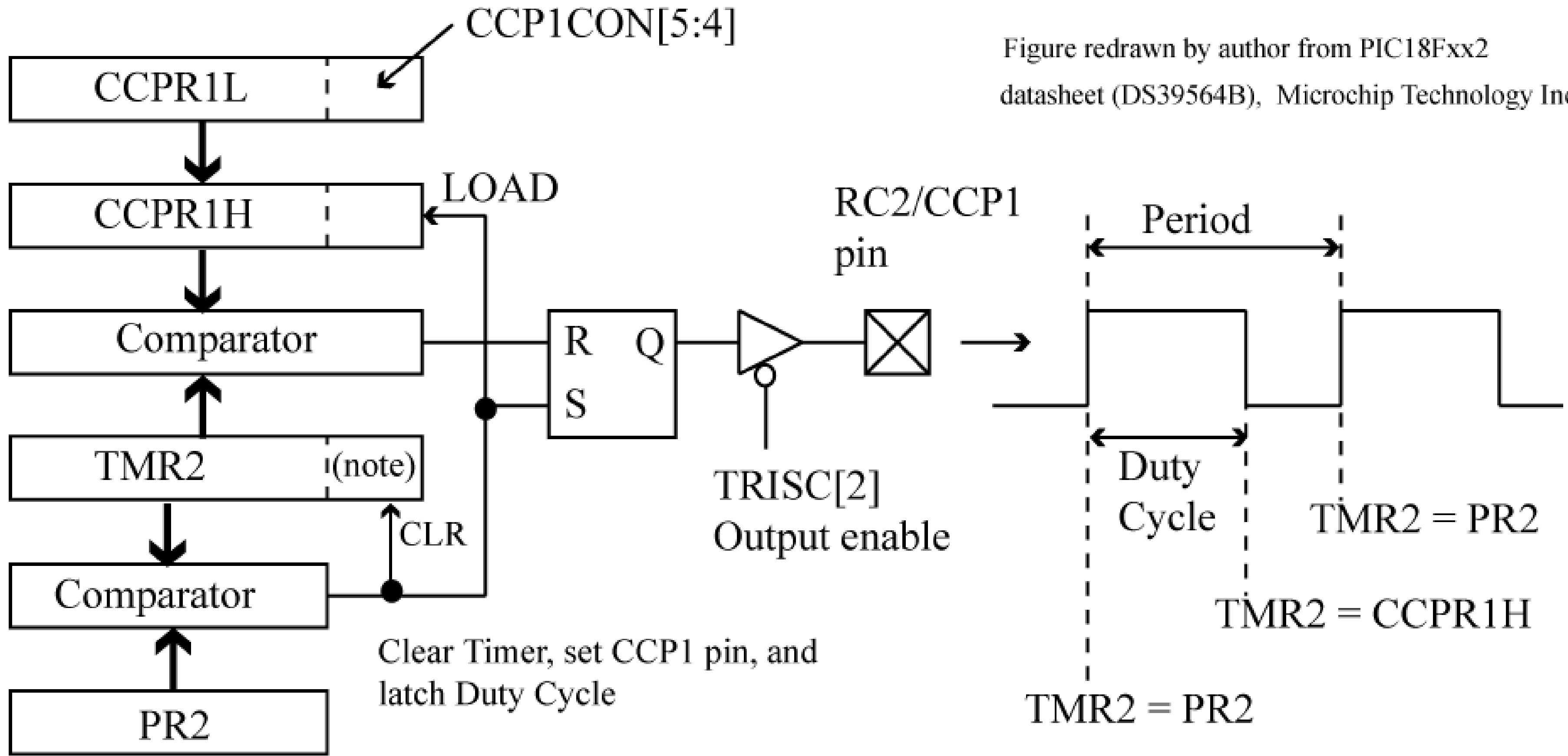
PWM Motor Control



1A	2A	EN	Direction
0	1	1	Forward
1	0	1	Reverse
1	1	1	Brake
x	x	0	Coast

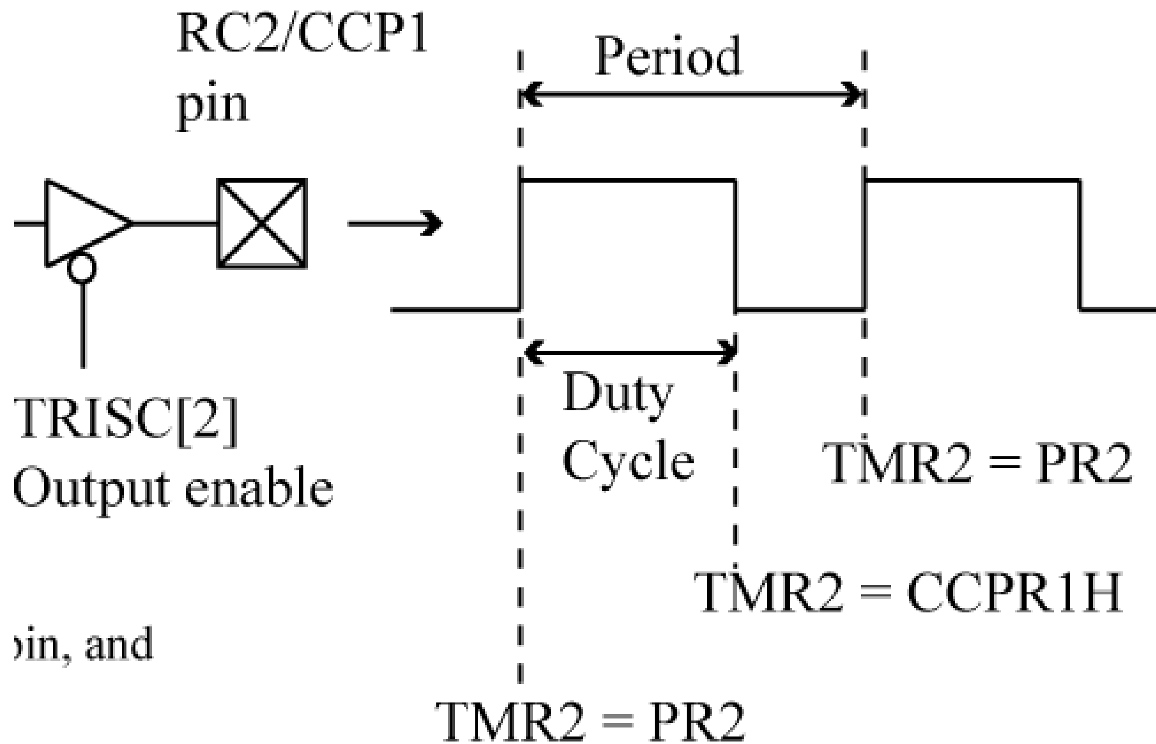
RB1 and RB0 control the motor direction. A logic ‘1’ at the H-bridge’s input becomes a high-current voltage applied to the motor while a logic ‘0’ becomes a high-current ground to the motor. Pulsing the enable pin with a PWM signal alternates applying current to the motor with periods in which no current is applied, causing the speed of the motor to vary with the PWM duty cycle.

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

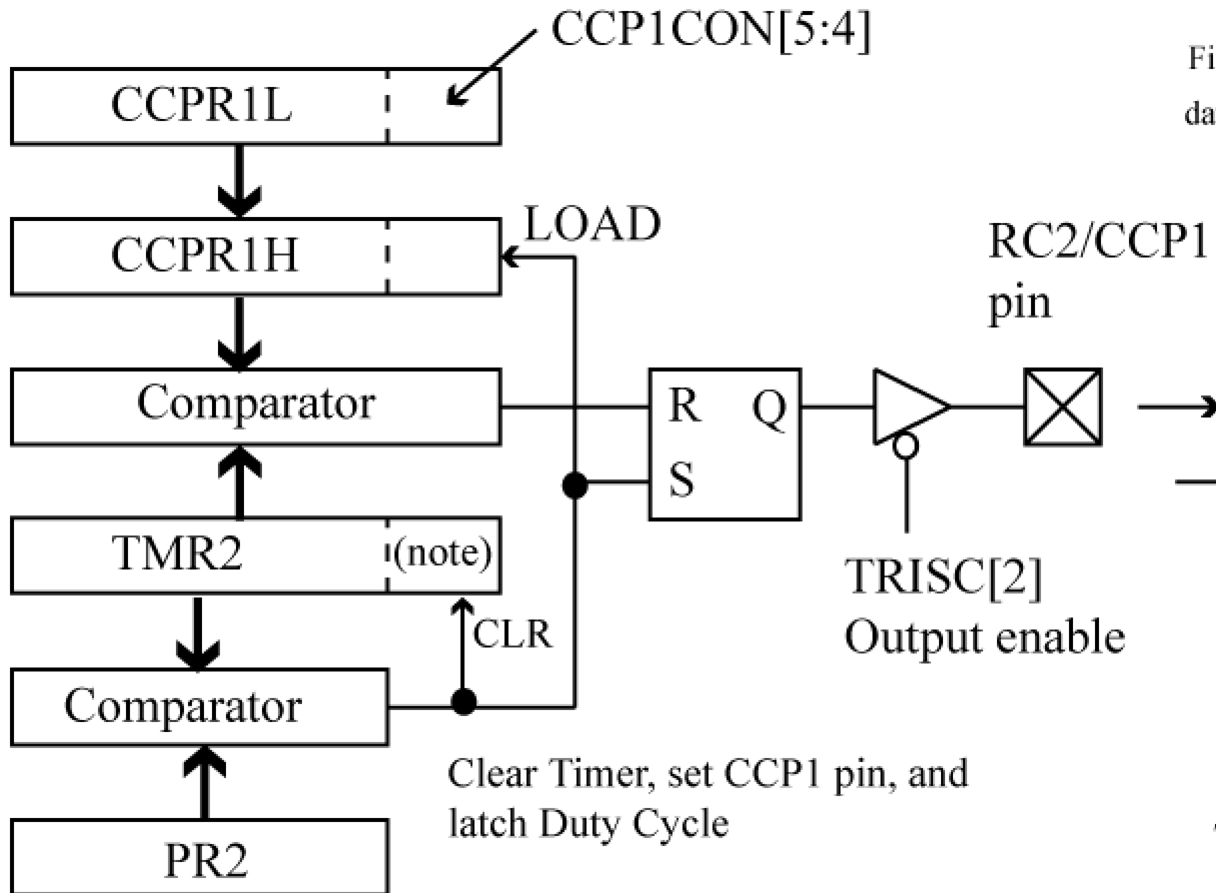
PIC18Fxx2 PWM Period



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

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

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

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.

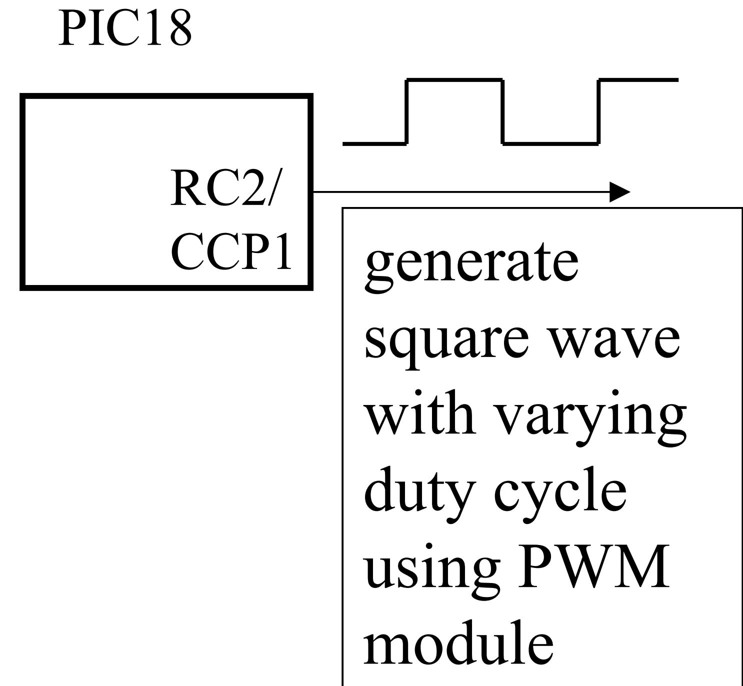
If duty cycle > Period, then the output will always be a high (will never be cleared).

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);
```

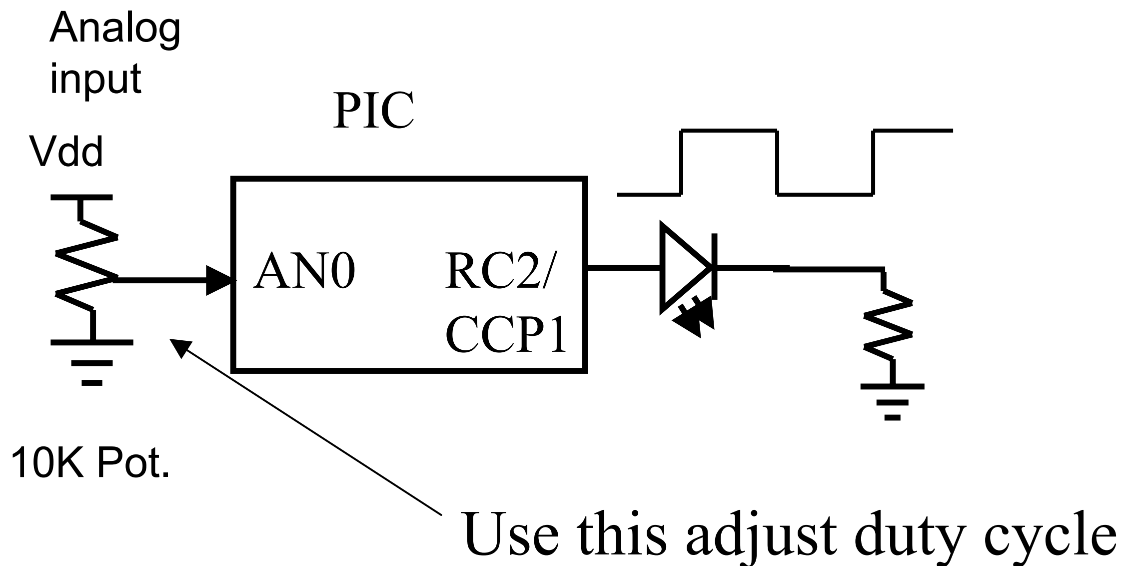
Configures Capture/Compare/PWM module for PWM operation

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

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

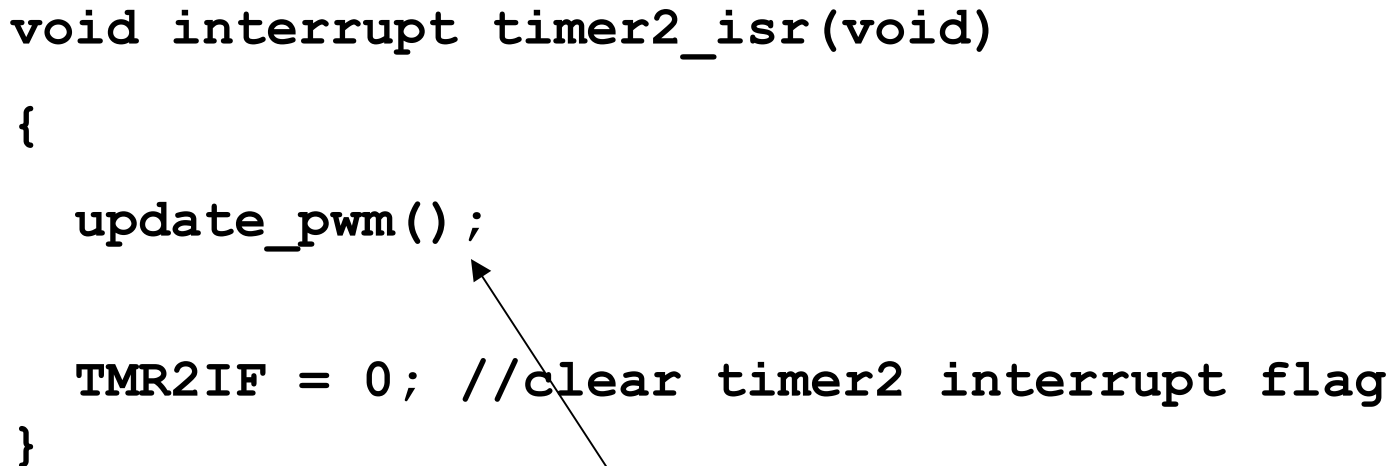
PWM test: mtrpwm.c

Use potentiometer and PIC ADC to adjust duty cycle for PWM to motor



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. User input specifies motor direction.

mtrpwm.c (cont)



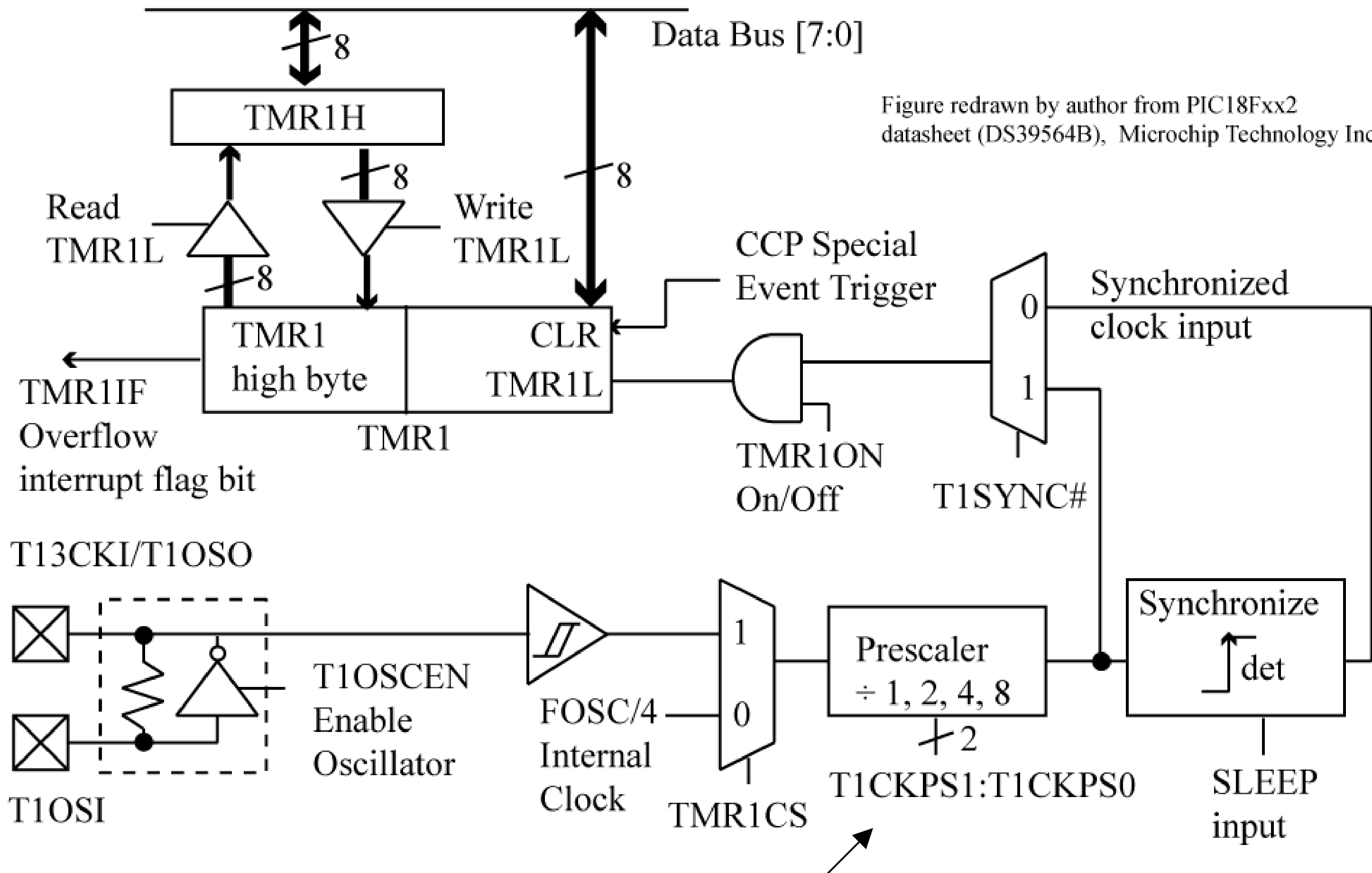
```
void interrupt timer2_isr(void)
{
  update_pwm();

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

This subroutine does work of reading A/D and changing the duty cycle. You must provide code which performs this task.

Other Timers on the PIC18F242

- Timer0 – can function as either a 16-bit or 8-bit timer
- Timer1/Timer3 – 16-bit timers with near identical capabilities
  - Timer1/Timer3 in capture/compare module
  - Timer1 is used in the 2nd half of the Servo.motor lab

Timer1 Block Diagram



Precaling options are 1,2,4,8. Can also be clocked by an source independent of main oscillator.

Reading/Writing Timer1

```
unsigned int tmr1_tics;
 char *ptr;

(1) ptr = (char *) &tmr1_tics;    // ptr points to LSB of tmr1_tics
(2) tmr1_tics = TMR1;                      // this works for read
(3) *ptr = TMR1L; *(ptr+1) = TMR1H;  // also works for read
(4) *(ptr+1) = TMR1H, *ptr = TMR1L;  // wrong order for read
(5) TMR1 = tmr1_tics;                      // wrong order for write
(6) TMR1H = *(ptr+1); TMR1L= *ptr;   // correct order for write
(7) TMR1H = (tmr1_tics)>>8; TMR1L= (tmr1_tics & 0xFF);  //ok as well
```

When reading, read TMR1L first!

When writing, write TMR1L last!

The statement *tmr1_tics = TMR1* works for read because TMR1L, TMR1H are stored in little endian order in the file registers, and TMR1L is read first, then TMR1H by the compiler generated code.

Measuring Time with Timer1/Timer3

To measure elapsed time, must convert time to timer tics and vice versa.

Timer tics to time

Time = Timer_tics * Timer_period

= Timer_tics * Prescale * 4/FOSC

Time to Timer Tics

Timer_tics = Time * FOSC/(4*Prescale)

Sample Computations

Let FOSC = 40 MHz, Prescale = 8

How much time is 2000 Timer1 tics?

Time = Timer_tics * Prescale * 4/FOSC

= 2000 * 8 * (4/40 MHz)

= .0016 seconds = 1.6 ms = 1600 uS

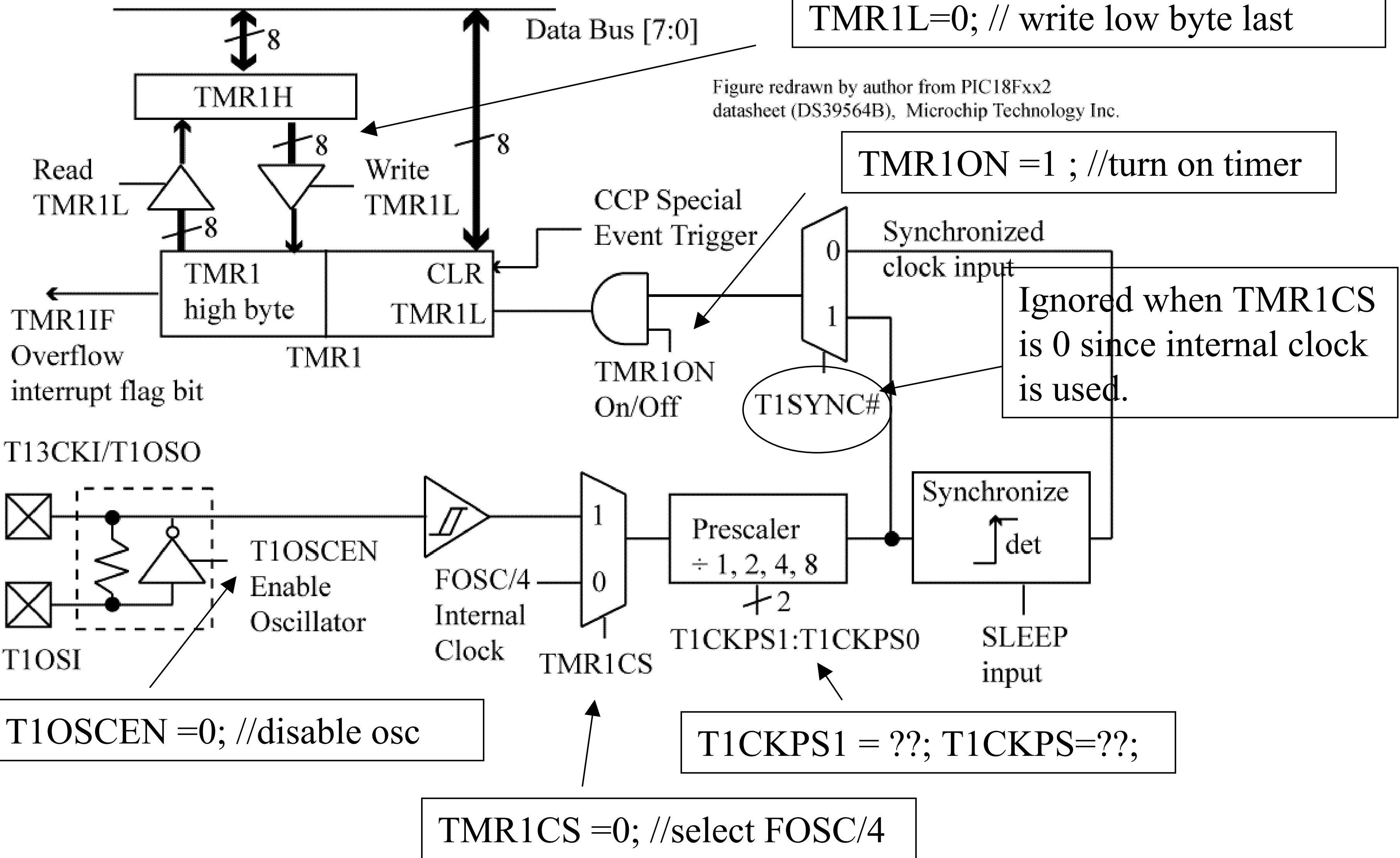
How many Timer1 tics is 1 ms? (0.001)

Timer_tics = Time * FOSC/(4*Prescale)

= 0.001 * 40 MHz / (4*8)

= 1250

Timer1 Configuration



Compare Mode

Generate an interrupt when CCPR1 register contents matches TMR1 contents

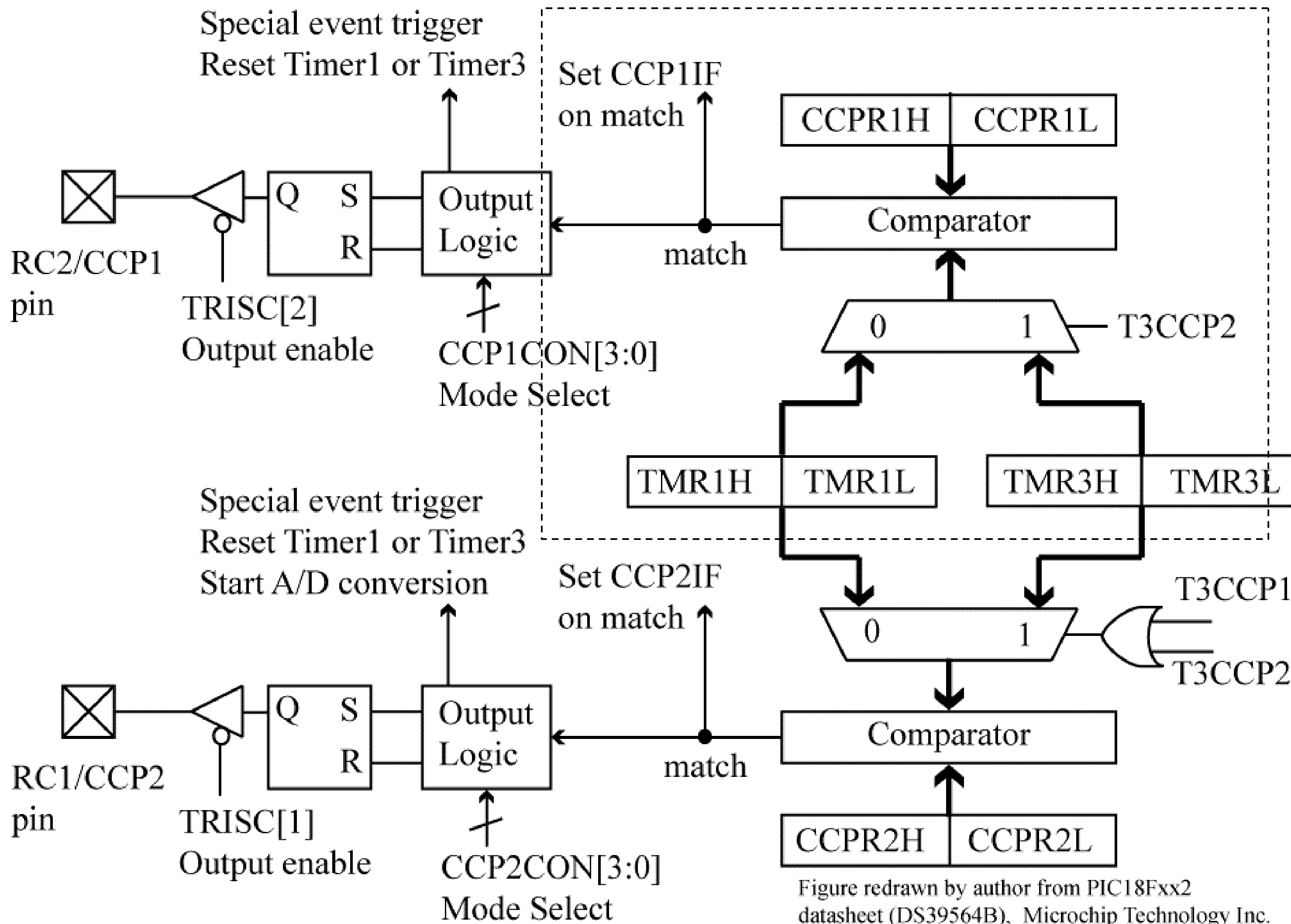


Figure redrawn by author from PIC18Fxx2 datasheet (DS39564B), Microchip Technology Inc.

Example: Wait for 1 ms to Elapse

Let FOSC = 40 MHz, Prescale = 8. Previously, saw that for these conditions, 1250 Timer1 tics is equal to 1 ms.

The code below is not interrupt driven!!!

```
// Assume Capture/Compare module configured for
//compare mode, and Timer1 is also configured
TMR1ON = 0;  // turn off timer
CCP1IF =0;     //clear CCP1 flag
CCPR1 = 1250;   // write 1250 to CCP1 register
TMR1H=0;     // clear timer1
TMR1L=0;     // write low byte last
TMR1ON =1;  // turn on timer
 while(!CCP1IF);  // wait for match of Timer1 and CCP1
 //when CCP1F is set and loop exited, 1 ms has elapsed
```

Compare Mode Configuration

REGISTER 14-1: CCP1CON REGISTER/CCP2CON REGISTER

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
bit 7							bit 0

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

bit 5-4 **DCxB1:DCxB0**: PWM Duty Cycle bit1 and bit0

Capture mode:
Unused

Compare mode:
Unused

PWM mode:
These bits are the two LSbs (bit1 and bit0) of the 10-bit PWM duty cycle. The upper eight bits (DCx9:DCx2) of the duty cycle are found in CCPRxL.

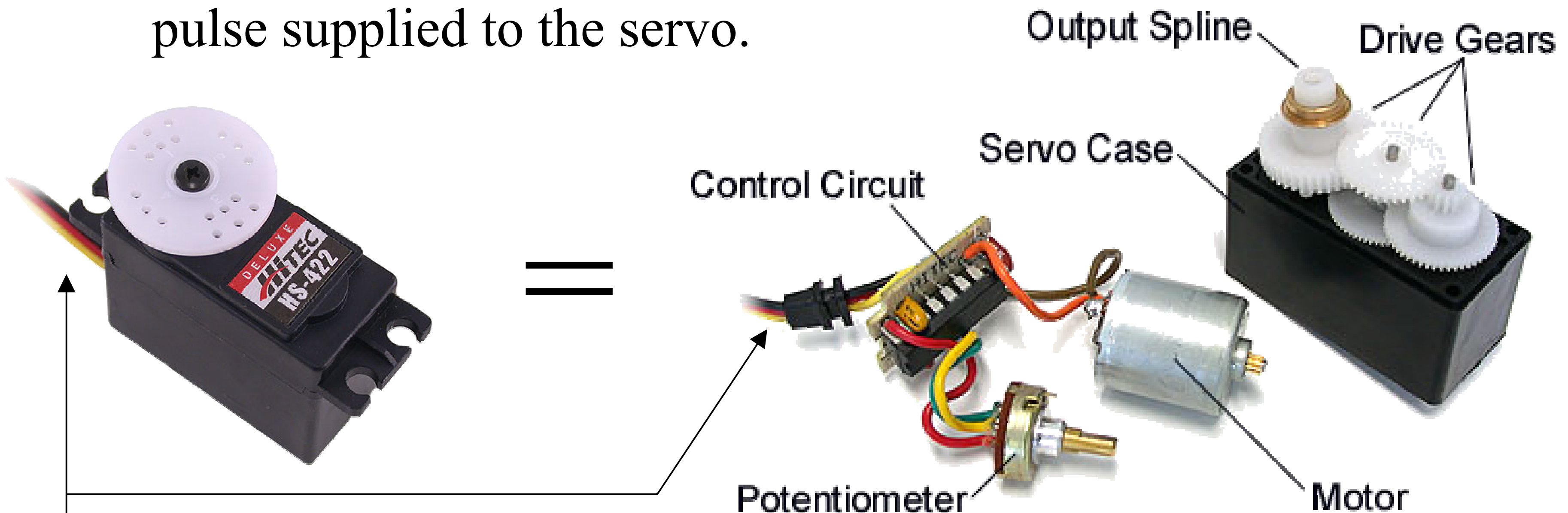
bit 3-0 **CCPxM3:CCPxM0**: CCPx Mode Select bits

0000 = Capture/Compare/PWM disabled (resets CCPx module)
0001 = Reserved
0010 = Compare mode, toggle output on match (CCPxIF bit is set)
0011 = Reserved
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,
Initialize CCP pin Low, on compare match force CCP pin High (CCPIF bit is set)
1001 = Compare mode,
Initialize CCP pin High, on compare match force CCP pin Low (CCPIF bit is set)
1010 = Compare mode,
Generate software interrupt on compare match (CCPIF bit is set, CCP pin is unaffected)
1011 = Compare mode,
Trigger special event (CCPIF bit is set)
11xx = PWM mode

Just sets the CCPxIF flag bit on compare match

Servos

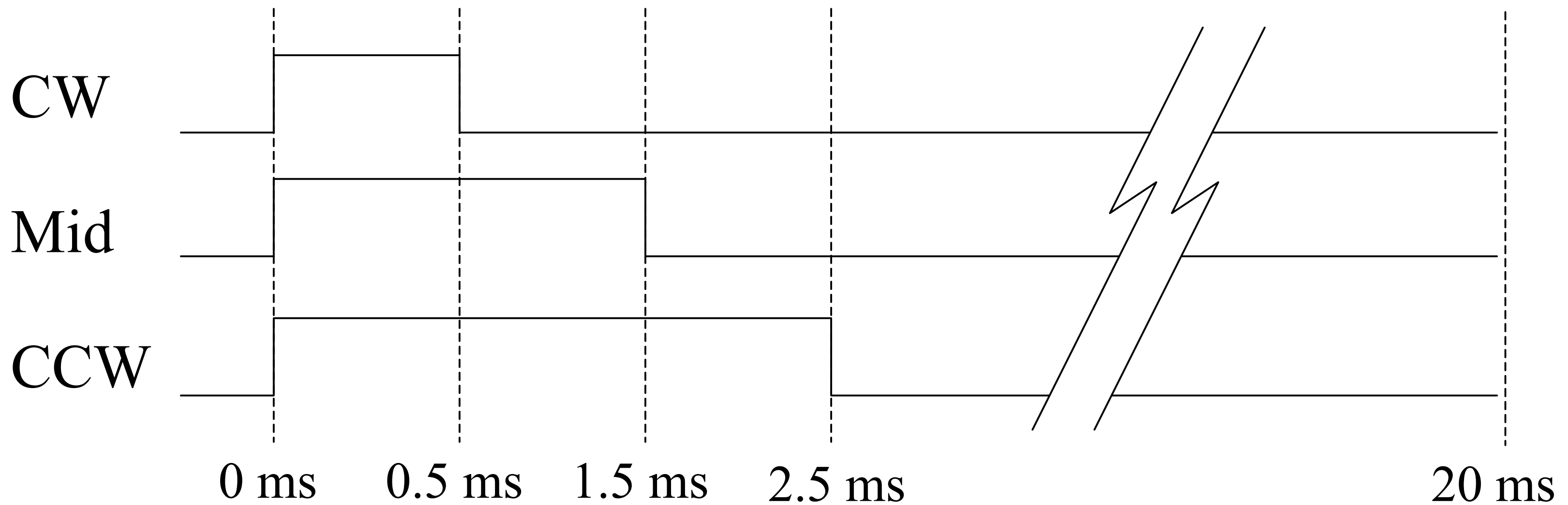
Servos, widely used to steer model cars, airplanes, and boats, consist of a motor with gearing to reduce the output speed and increase output torque and a control circuit which spins the motor until the motor's position measured by the potentiometer matches the desired position specified by a pulse supplied to the servo.



Cable: red = power (4.8V – 6.0V),
black = ground,
yellow = desired position

Controlling servos

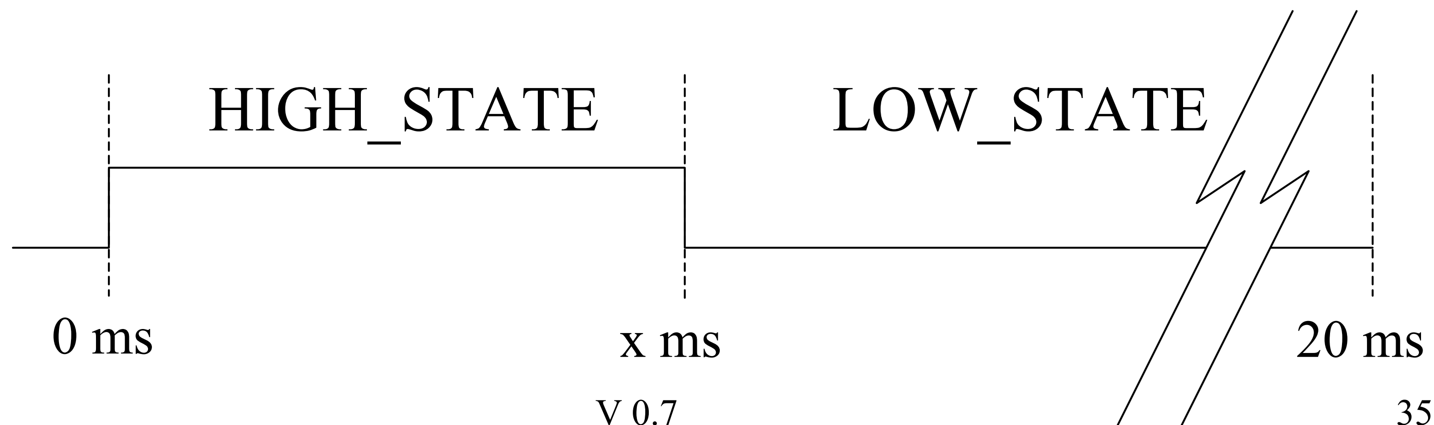
The high time of a pulse gives the desired position of the servos. The pulse width must be 20 ms. A high time of 1.5 ms moves the servo to its center position; smaller or larger values moves it clockwise or counterclockwise from the center position. This is termed pulse code modulation (PCM).



Not all servos can cover this entire range!

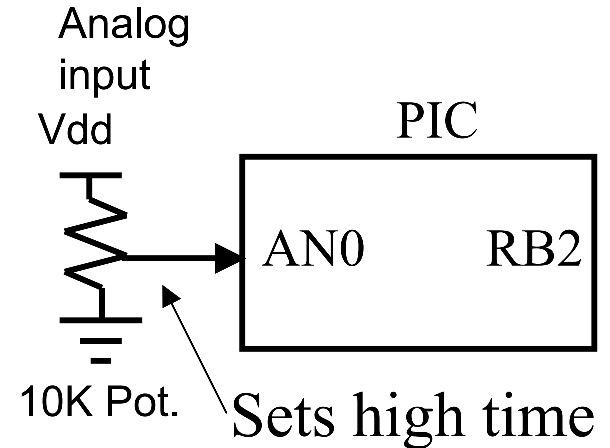
Controlling servos (cont.)

Because of the low frequency required for servos, the PWM hardware in the PIC cannot be used. Instead, a PCM signal must be generated using the compare function of the PIC. An ISR, called when a compare interrupt occurs, should perform PCM. The ISR should be written as a state machine with two states: HIGH_STATE and LOW_STATE.



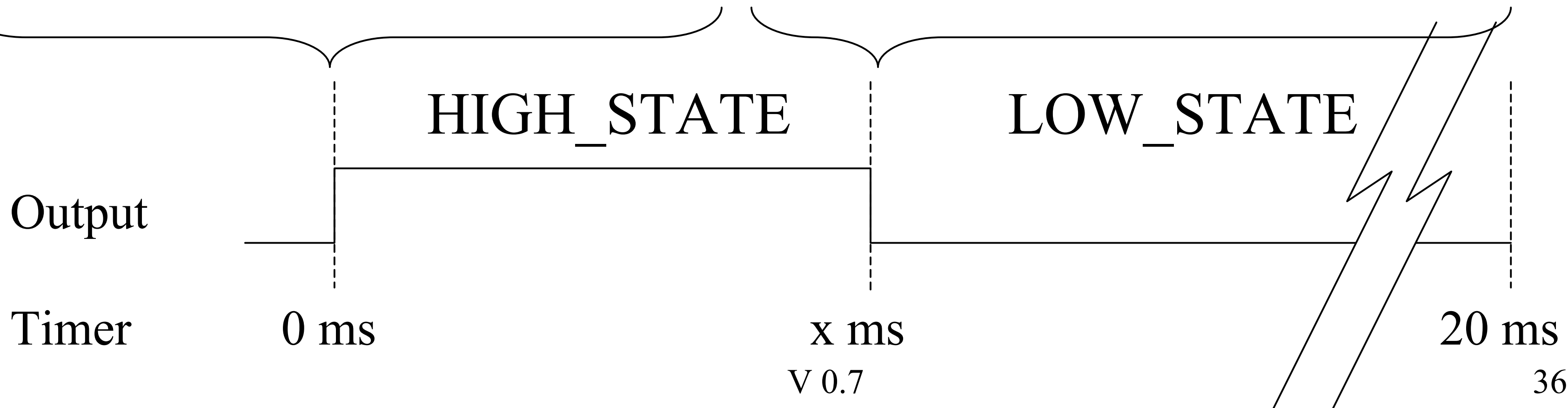
Controlling servos (cont.)

At each interrupt, the ISR should set up for the next interrupt. The high time of x ms, which specifies the servo position, should be read from a potentiometer connected to the PIC.



1. Set next state to LOW_STATE
2. Set output = 1
3. Reset timer to 0
4. Set next interrupt to x ms

1. Set next state to HIGH_STATE
2. Set output = 0
3. Set next interrupt to 20 ms



pcmgen.c

On HIGH_STATE interrupt:

1. Set next state to LOW_STATE
2. Set output = 1
3. Reset timer to 0
4. Set next interrupt to x ms

```
void interrupt picIsr()
{
  if (CCP1IF && CCP1IE)
  {
    // Is this the first interrupt?
    if (state == PWM_HIGH_STATE )
    {
❶     state = PWM_LOW_STATE; // on next interrupt, set output low

❷     // Clear timer1: write high byte then low byte
      TMR1H = 0;  TMR1L = 0;

      // Set RB2 output to 1 at beginning of PCM pulse
❸     RB2 = 1;

      // Set CCPR1 to generate at interrupt based on A/D result
      // Left justification assumed
      // interrupt is generated when Timer1 matches CCPR1 value
      // fill this in
❹     CCPR1 = ???  // computation based on
                   // ADRESH, PCM_0004_MS, and PCM_MULT_FACTOR
```

pcmgen.c (cont.)

```
❹    // Start a new A/D conversion for next time
        ??? //Add code here to begin a new A/D conversion

   } else {
❶    state =  PWM_HIGH_STATE; // on next interrupt set output high

     // Set RB2 output to 1 to end PCM pulse
❷    RB2 = 0;

     // Set CCPR1 to generate an interrupt at 20 ms
     // interrupt is generated when Timer1 matches CCPR1 value
❸    CCPR1 = ???????;
   }

   // Clear interrupt
   CCP1IF = 0;
  }
}
```

On LOW_STATE interrupt:

1. Set next state to HIGH_STATE
2. Set output = 0
3. Set next interrupt to 20 ms

main() configures interrupts / ports then enters an infinite loop