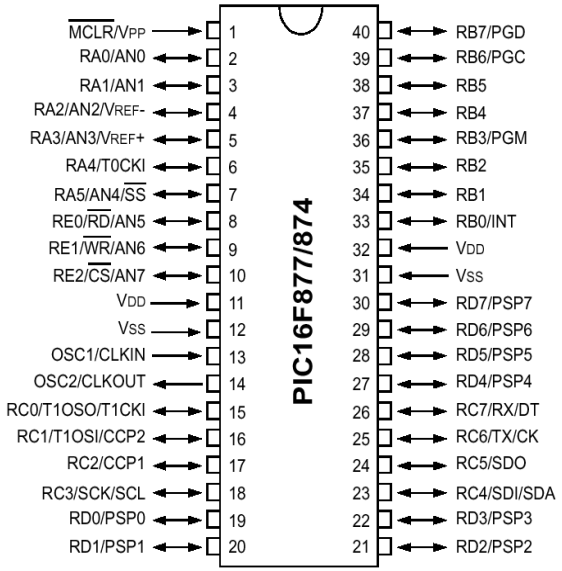
PIC Functionality

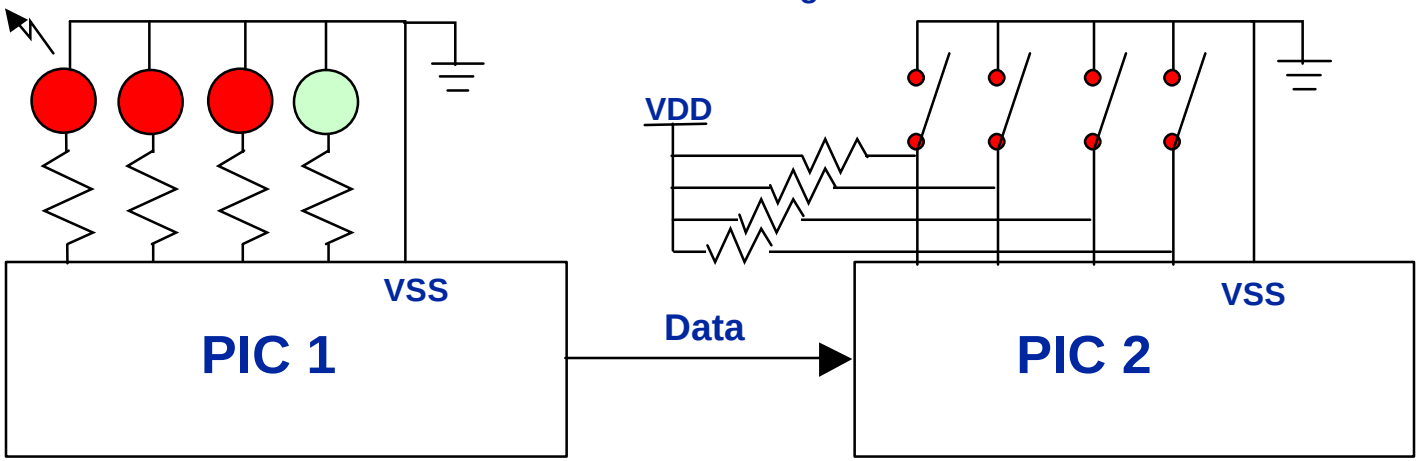
- General I/O
- Dedicated Interrupt
- Change State Interrupt
- Input Capture
- Output Compare
- PWM
- ADC
- RS232

General I/O

- **Logic Output**
  - light LEDs
  - Trigger solenoids
  - Transfer data
- **Logic Input**
  - Monitor switches
  - Monitor sensor logic
  - Monitor via polling
  - Transfer data

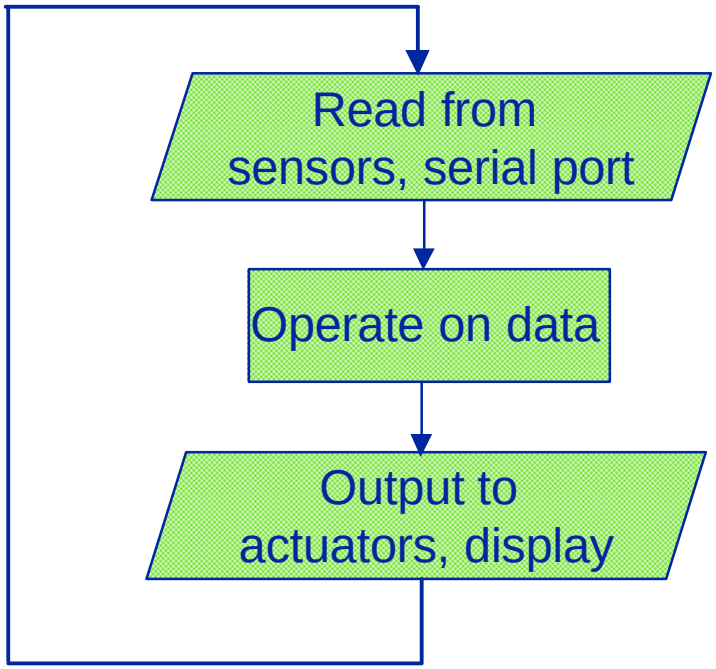


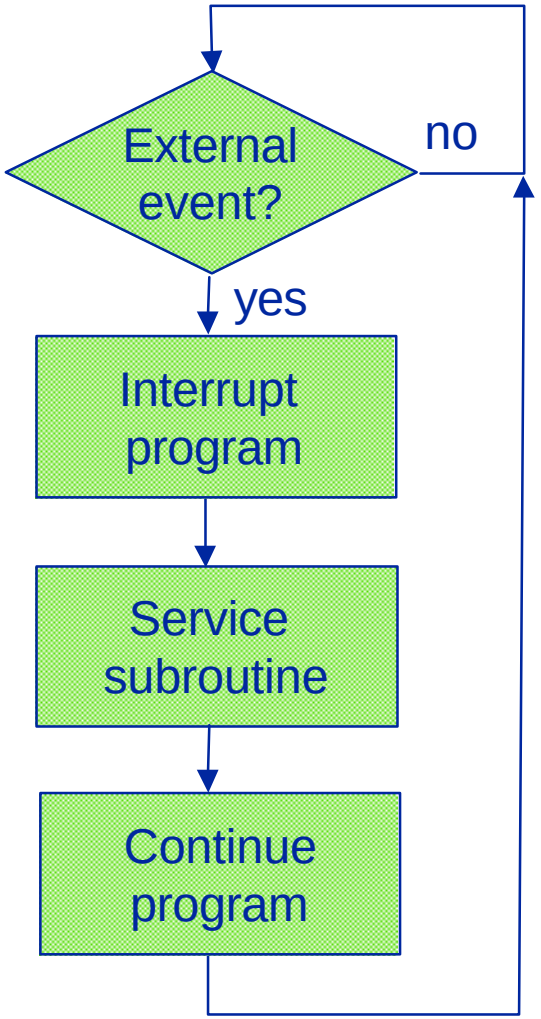
TRIS Register
Port ID
Read / Write via register



Interrupts

- **Program may be interrupted by external events which have a sense of urgency**
  - **reset**
  - **power failure**
  - **servicing external events**
  - **timing for external inputs**



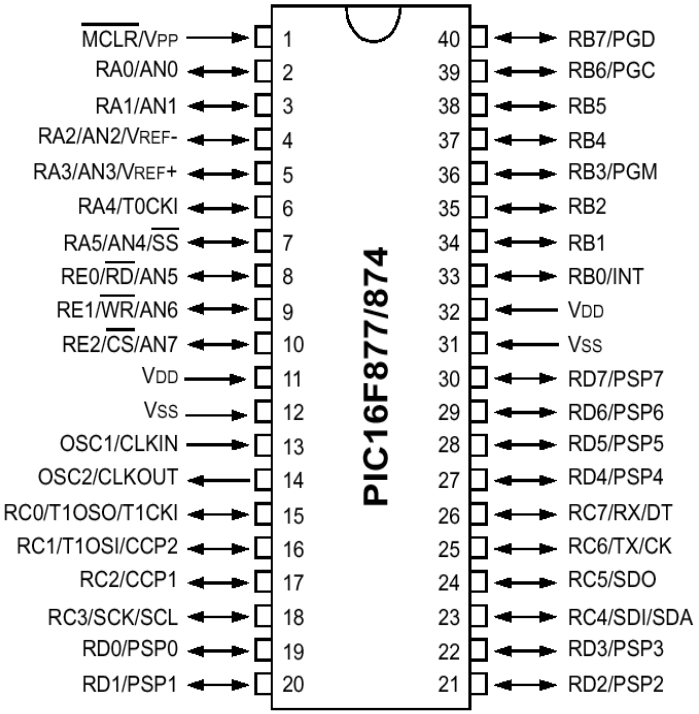


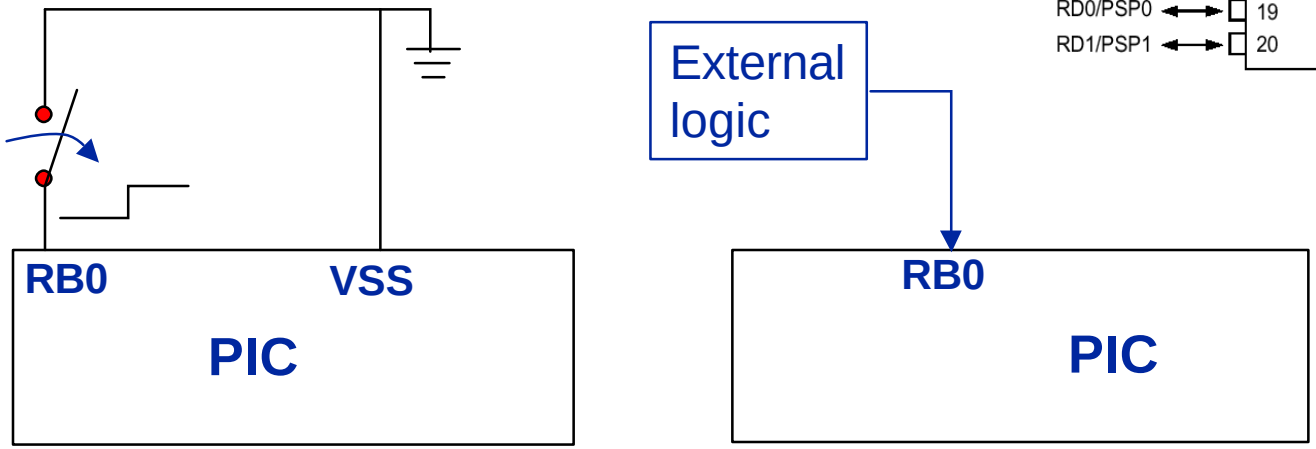
Interrupts

- **14 interrupts**
  - rb0 external interrupt
  - rb4:7 state change interrupt
  - analog to digital capture interrupt
  - parallel port read/write interrupt
  - usart receive interrupt
  - usart transmit interrupt
  - synro serial port interrupt (I^2C…)
  - timer0 overflow interrupt
  - timer1 overflow interrupt
  - capture compare / pwm 1 interrupt
  - capture compare / pwm 2 interrupt

Event Interrupt - RB0

- **RB0**
  - **Dedicated interrupt**
  - **Ideal for immediate action on change state**
  - **Can be set to detect a rising or falling edge**
  - **Port B has optional internal pull-up resistors for logic high**
    - **port_b_pullups(TRUE)**





Interrupt Subroutines

- Use the compiler directive `#INT_xxx`
- The compiler will generate code to jump to the ISR when the interrupt xxx is detected
- It will generate code to save and restore the machine state
- The compiler will also clear the interrupt
  - Call `ENABLE_INTERRUPT` initially!

- Method 2: use assembly language…

Digital I/O with interrupts

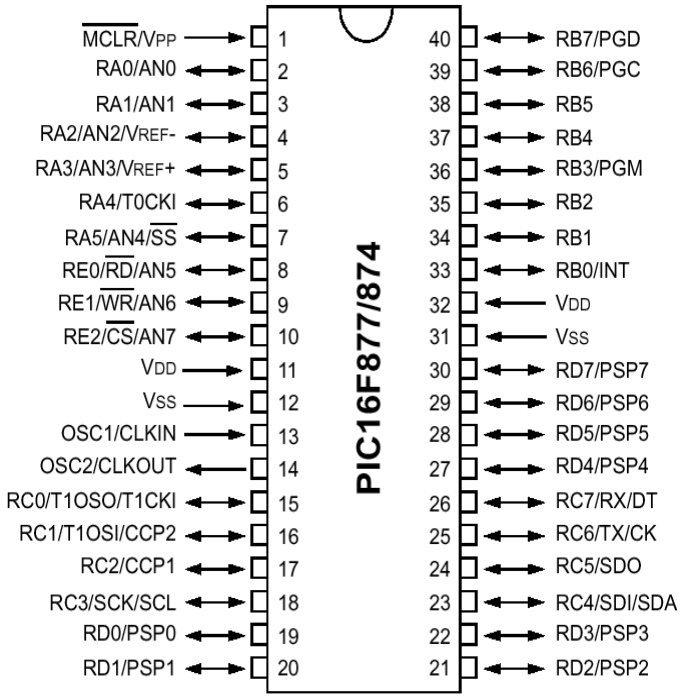
- **Port B can be set to generate interrupts when individual pins RB4:RB7 change**
  - **Pins must be configured as inputs can cause this interrupt to occur**
  - **You must write an interrupt service routine to handle the interrupt**

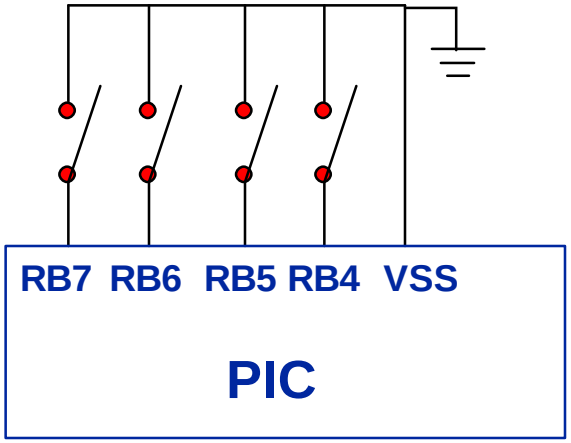
More on this part later...

```
#INT_RB
Portb_Change_ISR() {
 if(seconds!=100)  {
       ++events;
       printf("%u chickens. \n",events);
    }
}
```

Event Interrupt - PORTB Change 3-7

- **RB3-7**
  - Detect change on any of 4 pins
  - Must read register to know which
  - Good for reading encoder/keypads





RB3-7

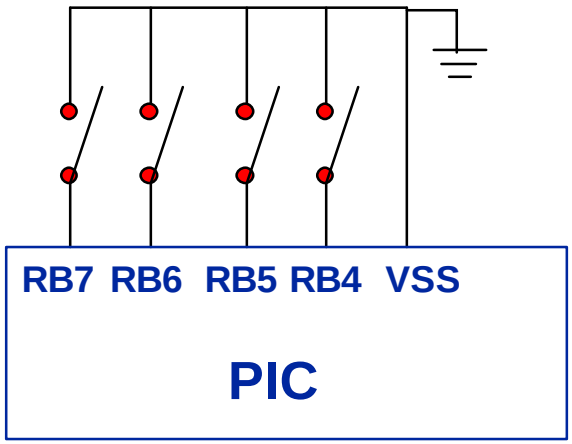
```
enable_interrupts(GLOBAL);
enable_interrupts(INT_RB);

#int_rb
rb_isr ( ) {
  byte changes;
  changes = last_b ^ port_b;
  last_b = port_b;
  if (bit_test(changes,4 )&& !bit_test(last_b,4)){
    //b4 went low
  }
  if (bit_test(changes,5)&& !bit_test (last_b,5)){
    //b5 went low
  }
  . . .
  delay-ms (100);
}
```

Enable this interrupt

Define the interrupt

Need to read port and compare to previous value to know which pins have changed

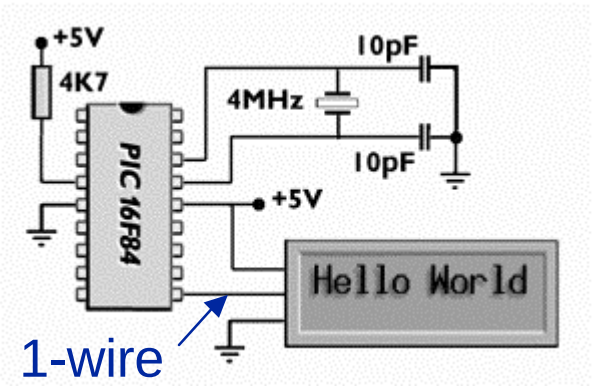
Can test for change pins by masking



LCD Interface

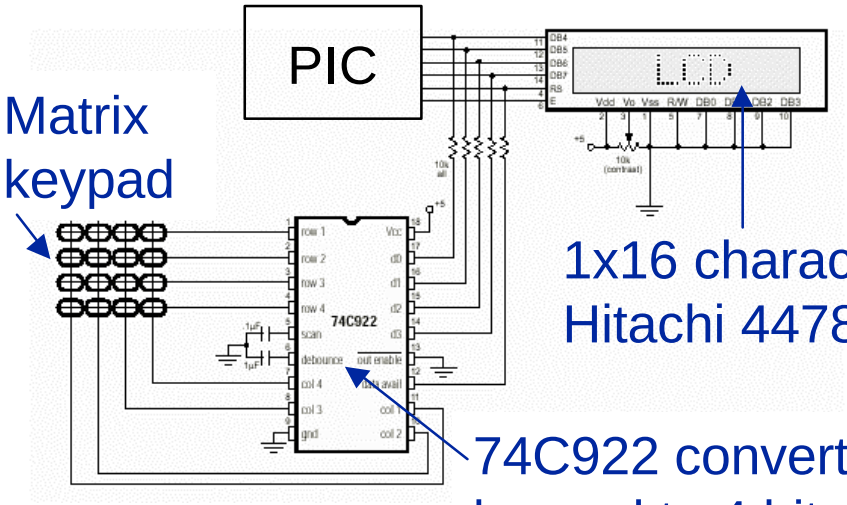
- Adding an LCD allows remote viewing of operation
- LCDs come in two flavors; serial and parallel
- Parallel LCDs can be multiplexed with keypad
  - Matrix keypad: pressing a key shorts row to column

Serial LCD



1-wire serial input

Parallel LCD w/keypad



See example code EX_92LCD.C

Interrupt Handling

- When an interrupt is responded to, the GIE is cleared to disable any further interrupt, the return address is pushed onto the stack and the PC is loaded with 0004h.
- Once in the interrupt service routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits.
- The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid recursive interrupts.

Ok; turn it on. Is it working…?

- Implement routines to debug your circuit
- Use test points to signal events
- Use the RS232 port to display variables, status, state of the circuit, etc.
- Use LEDs, buzzers, LCD displays, etc.
- A diagnostic connector is not a bad idea…

Capture/Compare/PWM (CCP) Modules

- **Each CCP module contains a 16-bit register which can operate as a:**
  - **16-bit Capture register**
  - **16-bit Compare register**
  - **PWM master/slave Duty Cycle register**

- **CCP modules also require timer resources**

CCP MODE	Timer Resource
Capture	**Timer1**
Compare	**Timer1**
PWM	**Timer2**

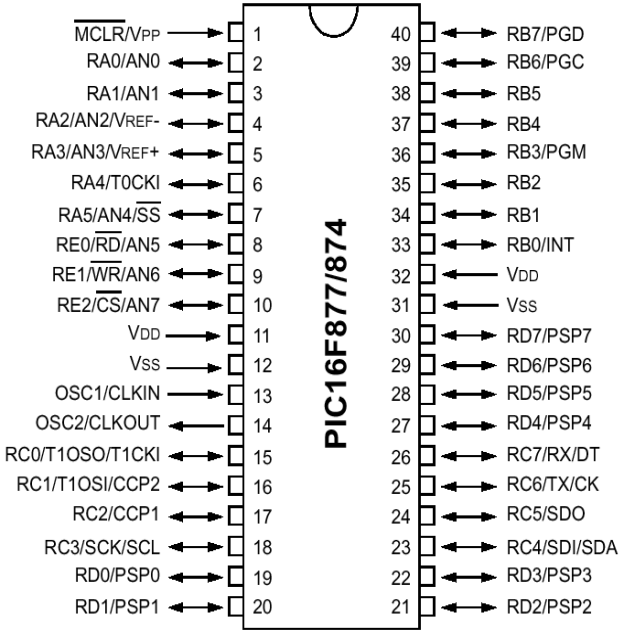
Capture/Compare/PWM Modules (cont.)

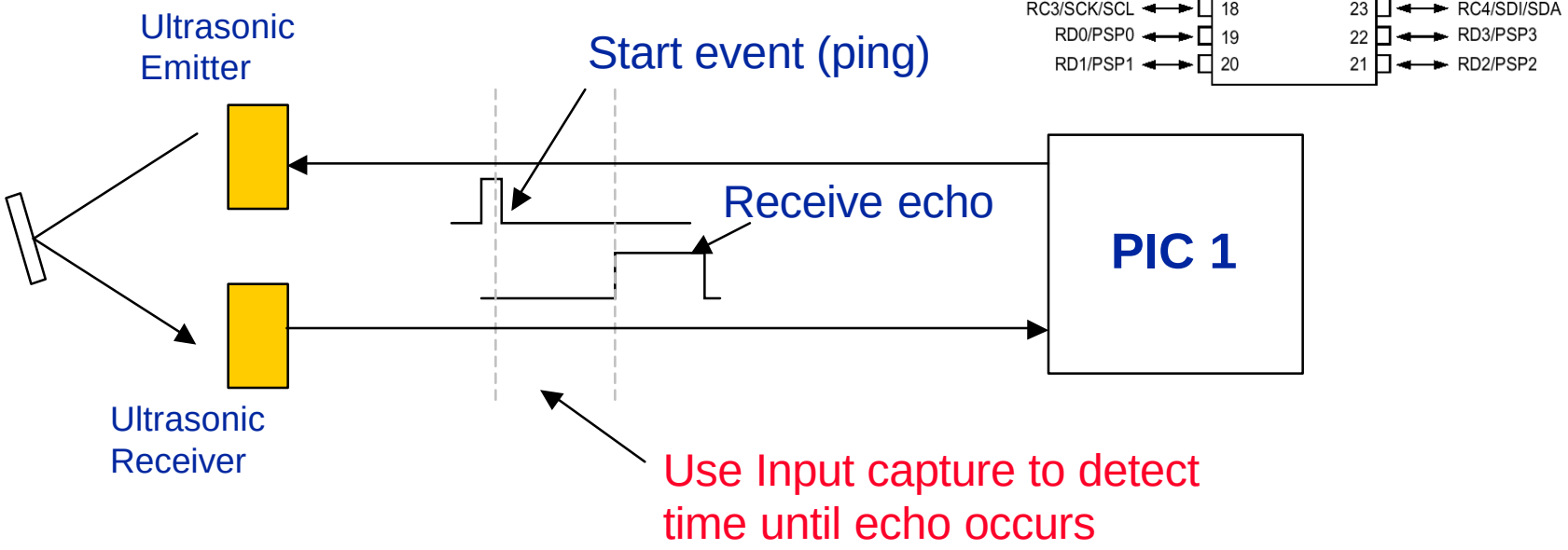
- **Both CCP1 and CCP2 are identical in operation, with the exception being the operation of the special event trigger.**

CCPx Mode	CCPy Mode	Interaction
Capture	Capture	Same TMR1 time-base.
Capture	Compare	The compare should be configured for the special event trigger, which clears TMR1.
Compare	Compare	The compare(s) should be configured for the special event trigger, which clears TMR1.
PWM	PWM	The PWMs will have the same frequency and update rate (TMR2 interrupt).
PWM	Capture	None.
PWM	Compare	None.

Input Capture

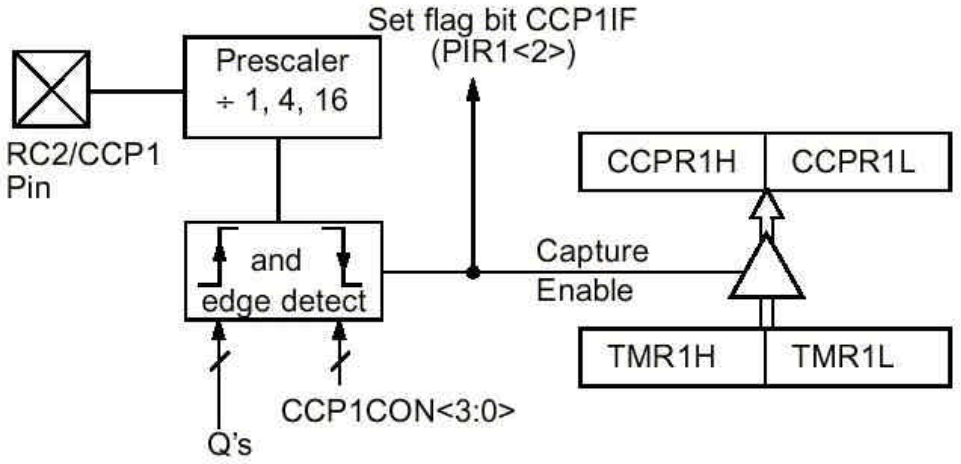
- **Input Capture**
  - **Measure timing of external event**
  - **Trigger when pin changes state**
  - **Trigger stores current timer value**
  - **Utilizes timer 1**





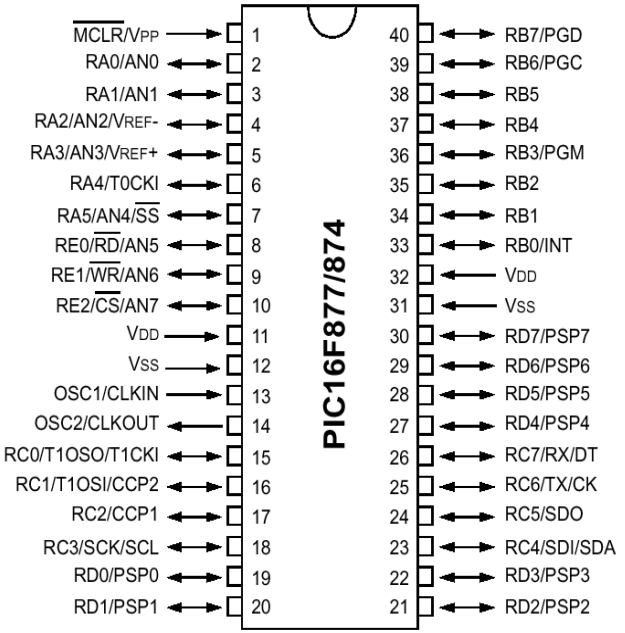
Using Input Capture

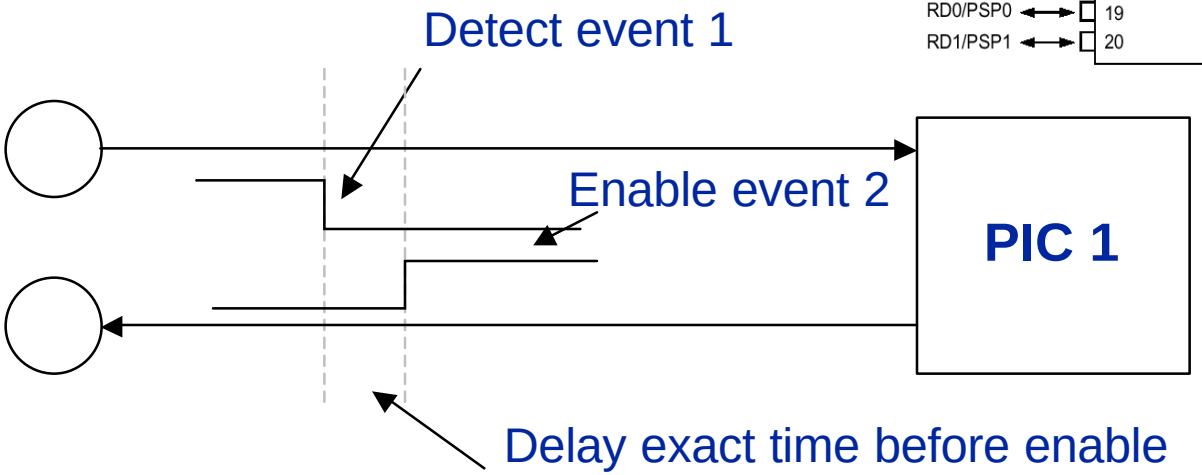
- In Capture mode, CCPR1H:CCPR1L captures the 16-bit value of the TMR1 register when an event occurs
- An event is defined as:
  - Every falling edge = CCP_CAPTURE_FE
  - Every rising edge = CCP_CAPTURE_RE
  - Every 4th rising edge = CCP_CAPTURE_DIV4
  - Every 16th rising edge = CCP_CAPTURE_DIV16
- Use this if you want to precisely time the rising and/or falling edge of a digital input



Output Compare

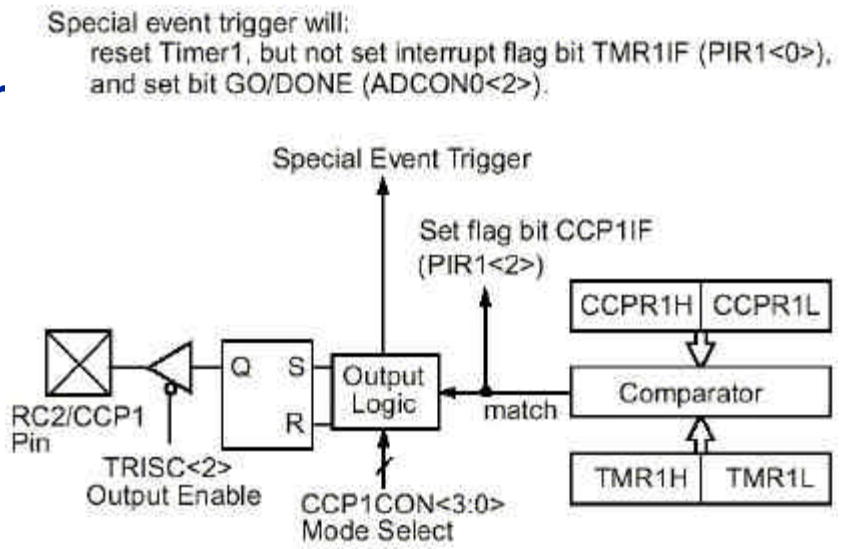
- **Output Compare**
  - Generate exact timing signals or events
  - Compare system clock to trigger clock
  - When registers match, changes state of output pin
  - Can set output pin high, low or toggle
  - Utilizes timer 1





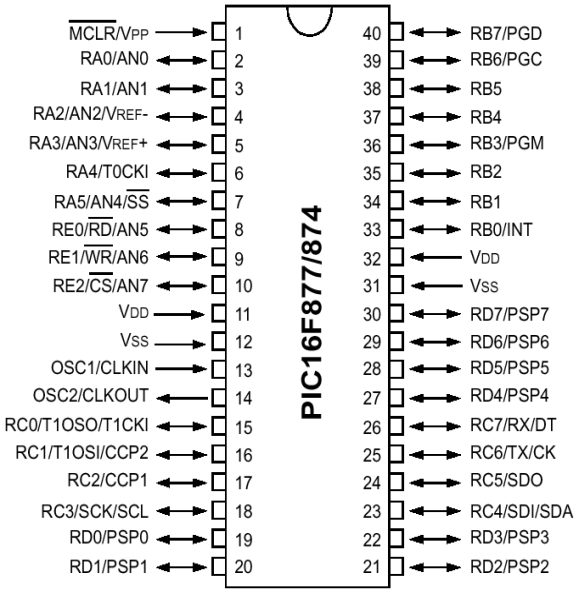
Using Output Compare

- In Compare mode, the 16-bit CCPR1 register value is constantly compared against the TMR1 register pair value. When a match occurs, the RC2/CCP1 pin is
  - Driven high
  - Driven low
  - Remains unchanged
- Use this if you want to output precisely timed digital waveforms (timing, etc.)
- CCP_COMPARE_SET_ON_MATCH
- CCP_COMPARE_CLR_ON_MATCH
- CCP_COMPARE_INT
- CCP_COMPARE_RESET_TIMER

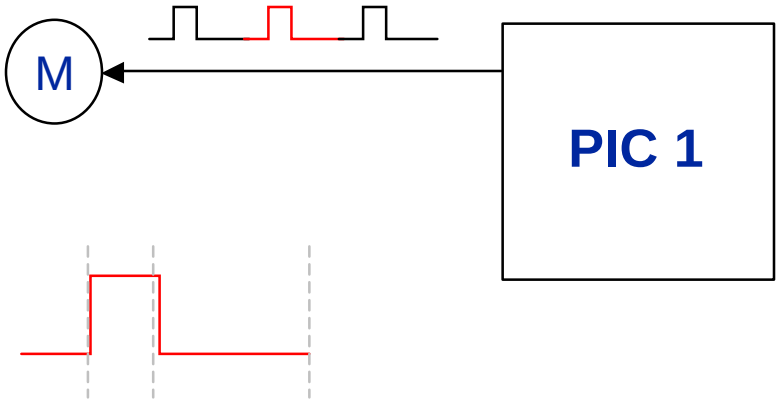


PWM

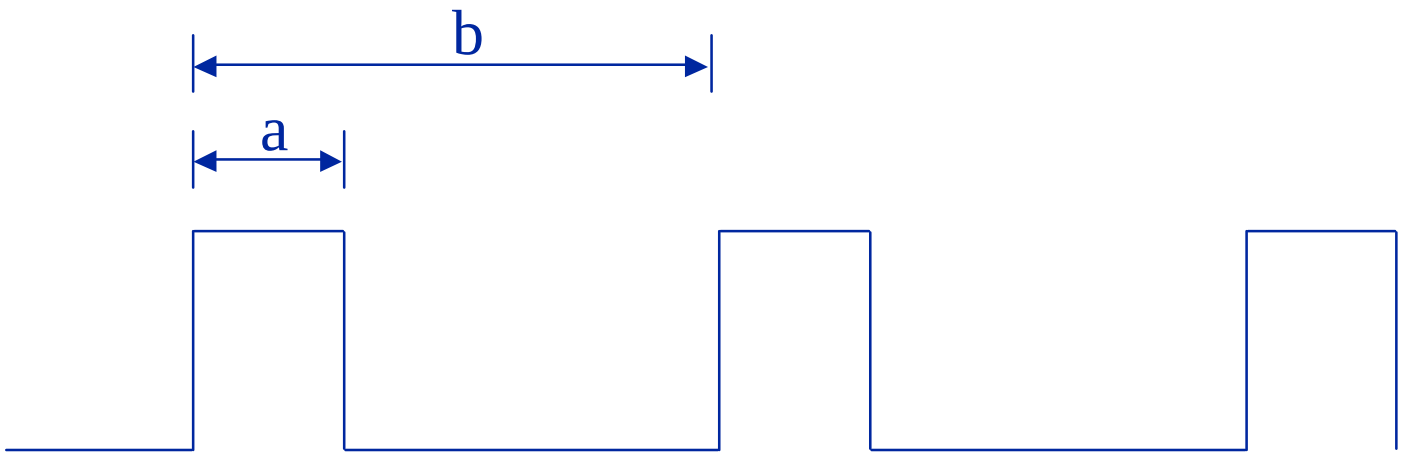
- **PWM**
  - **Generate timed out repeatedly (output capture high / output capture low)**
  - **Digitally produce varying output voltage**
  - **Often used to drive motors**
  - **Can be used for Servo motors also**
  - **PIC can generate 2 independent signals**



← Continuous signal



Using PWM Generator



$$\text{Duty cycle} = \frac{a}{b} \times 100\% \qquad \text{Freq} = \frac{1}{b} \qquad \text{(usually constant)}$$

- **PWM is commonly used as a control signal for DC motors.**
- **PIC has this feature built-in: 2 Capture/Compare/PWM modules.**
- **Compiler supports PWM functions:**
  - `setup_ccp1(CCP_PWM)`
  - `set_pwm1_duty(value)`

PWM Generator (cont.)

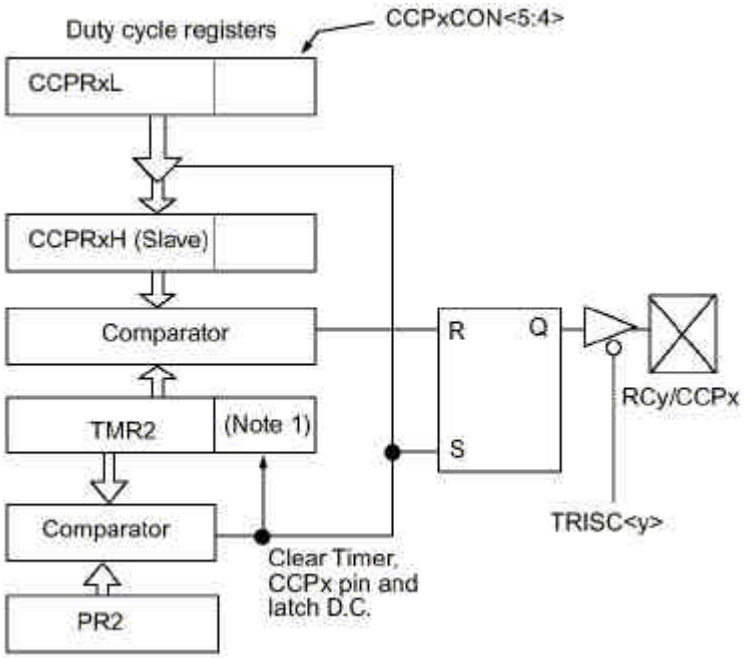
- **Timer2 resources must be used in conjunction with the PWM module**

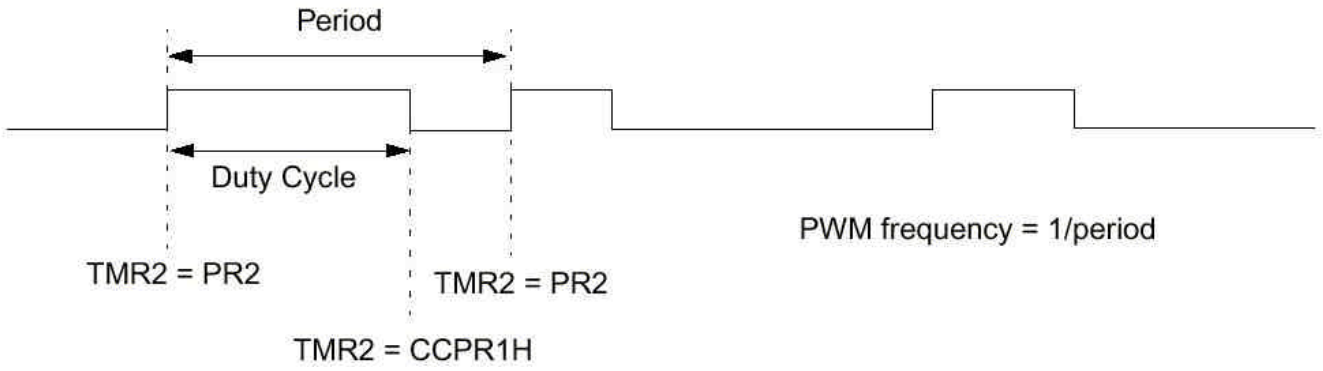
CCP MODE	Timer Resource
Capture	**Timer1**
Compare	**Timer1**
PWM	**Timer2**

- **The PWM signal can attain a resolution of up to 10-bits, from the 8-bit Timer2 module**
  - **This gives 1024 steps of variance from an 8-bit overflow counter.**

PWM Generator (cont.)

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





PWM

```
main() {
  output_high(PIN_B7);
  output_low(PIN_B6);
  output_high(PIN_B5);
  output_low(PIN_B4);

  setup_ccp1(CCP_PWM);          // Configure CCP1 as a PWM
  setup_ccp2(CCP_PWM);          // Configure CCP2 as PWM

  //The timing values here will need to be set
  //according to the clock you are using, and the
  //PWM rate you require. These values give a PWM
  //pulse that is at about 4KHz with a 4MHz clock,
  //((clock/4/4)/64)=3906Hz, and an 'update' interval
  //at (3906Hz)/15 = 244Hz.

  setup_timer_2(T2_DIV_BY_4,64,15);

  set_pwm1_duty(left_motor_speed);
  set_pwm2_duty(right_motor_speed);
}
```

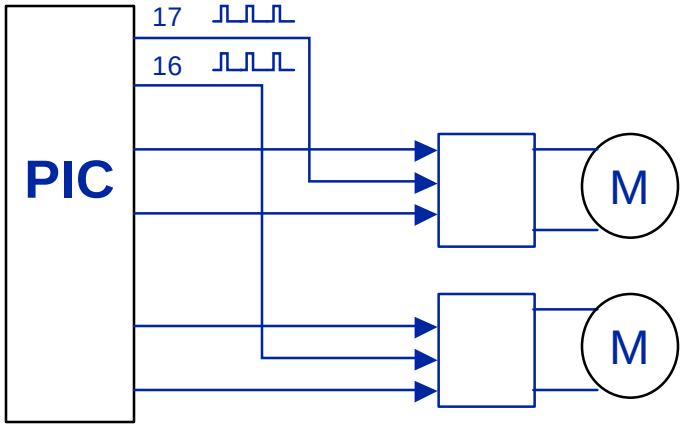
Set motor logic pins for direction

Set CCP1 and CCP2 as PWM

Set up timer 2 for PWM period
(prescale, period for overflow reset, postscale for interrupt)

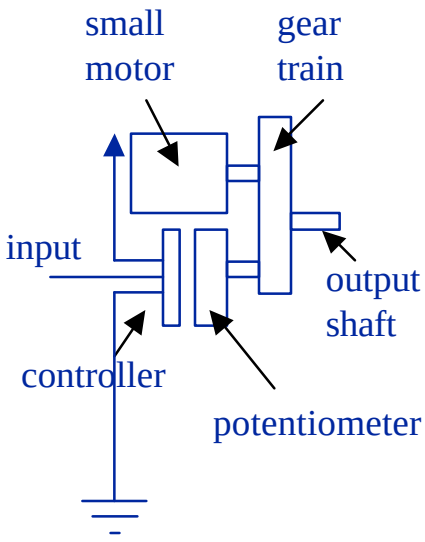
Now set the PWM for each motor (val 0-255).

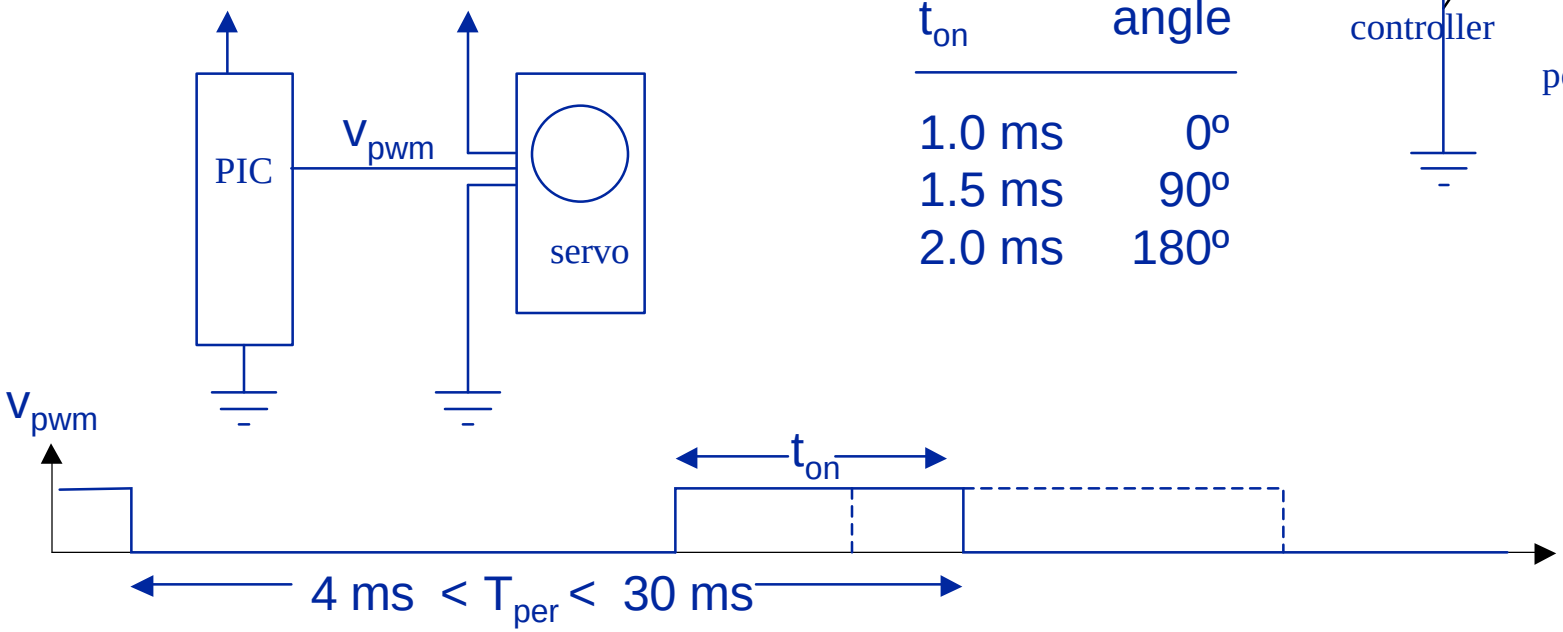
In most cases, the duty cycle update is inside its own function so it can be called by a state machine or periodic interrupt



Driving a R/C Servo

- **Angular position of servo set by pulse width**
- **Convenient to drive with PIC PWM output or output compare**



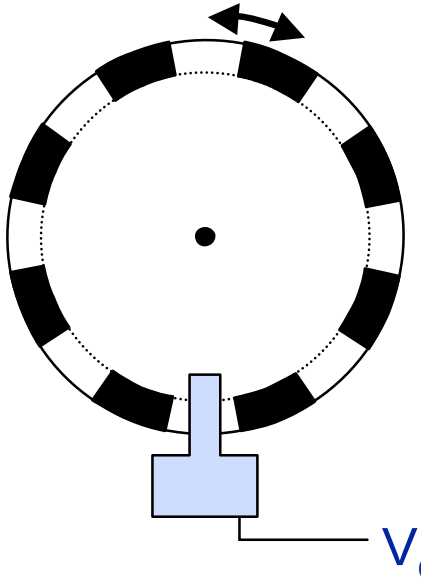


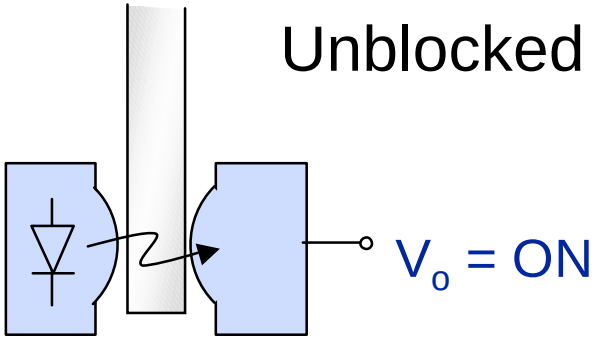
t_{on}	angle
1.0 ms	0º
1.5 ms	90º
2.0 ms	180º

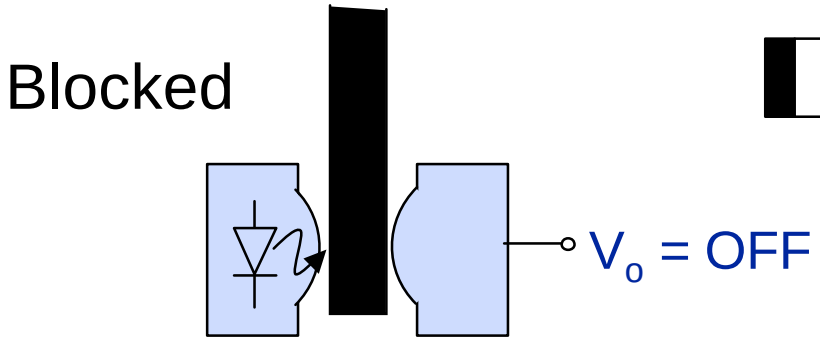
http://www.piclist.com/techref/microchip/servotst.c.htm
http://www.piclist.com/techref/microchip/language/c/io/servo-jpb.htm

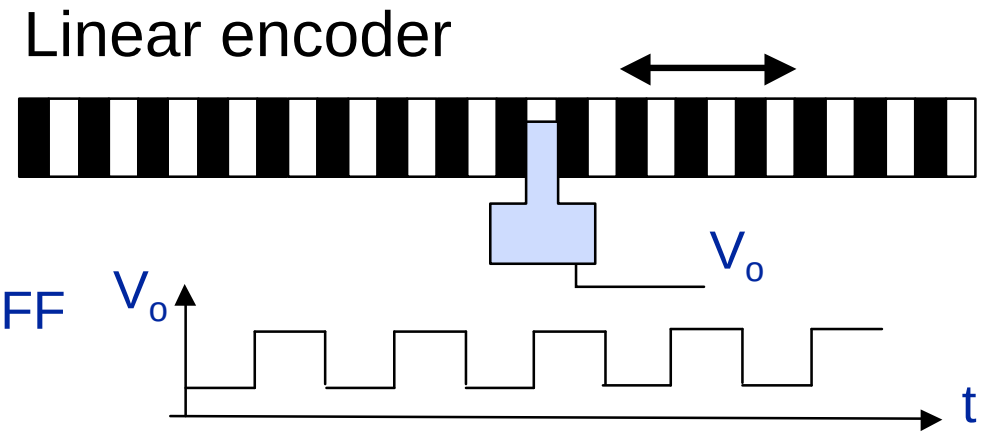
Encoders

- **Infrared emitter and detector placed across gap**
- **Moving pattern between gap modulates light**
- **Motion is detected by electronically counting pulses of light**





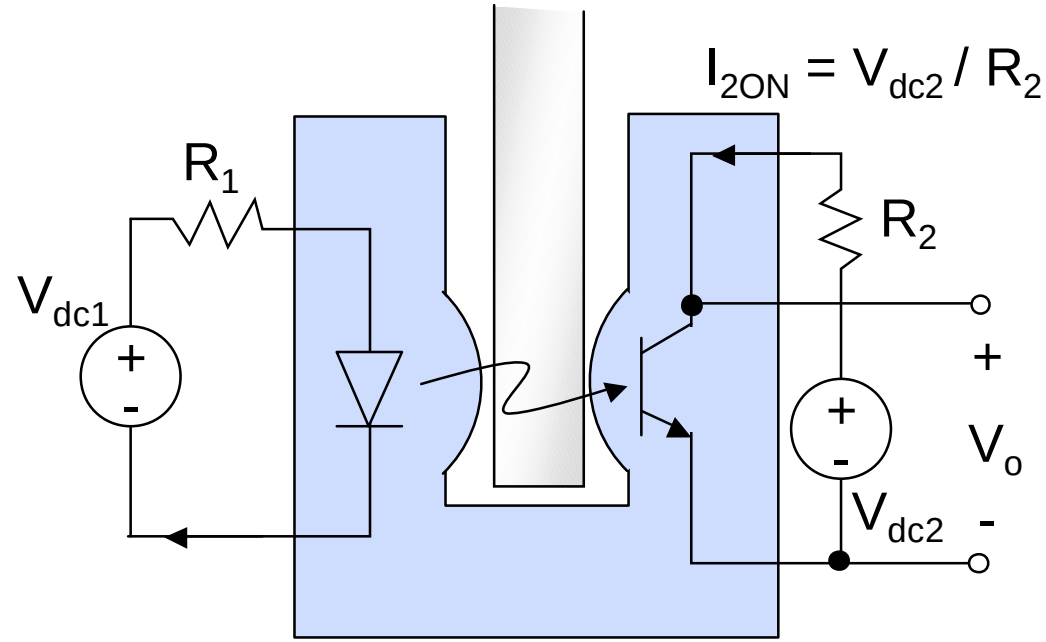




IR Emitter-Detector Components

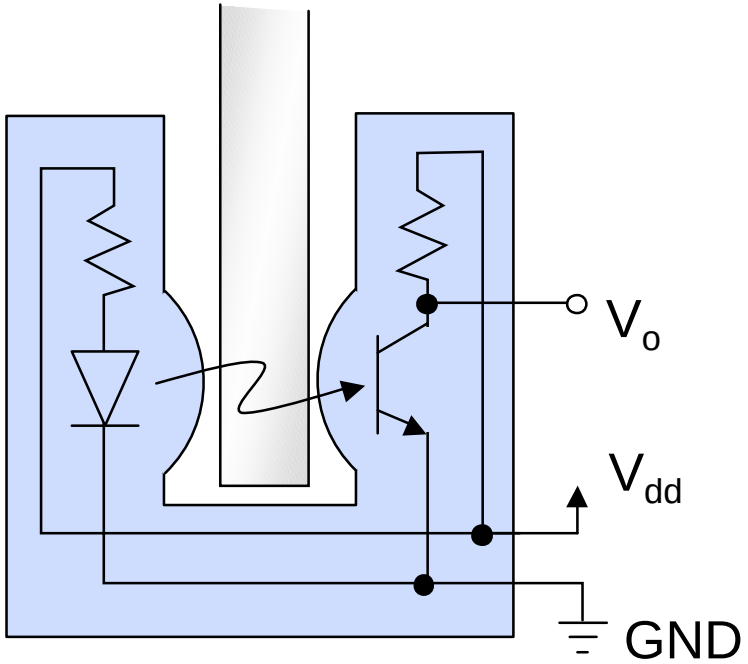
- **Available in individual emitter/detector pairs or in slot packages**
- **Many components require external resistors to set current**
- **Set R_1 and R_2 such that current ratings are not exceeded**

Component type #1



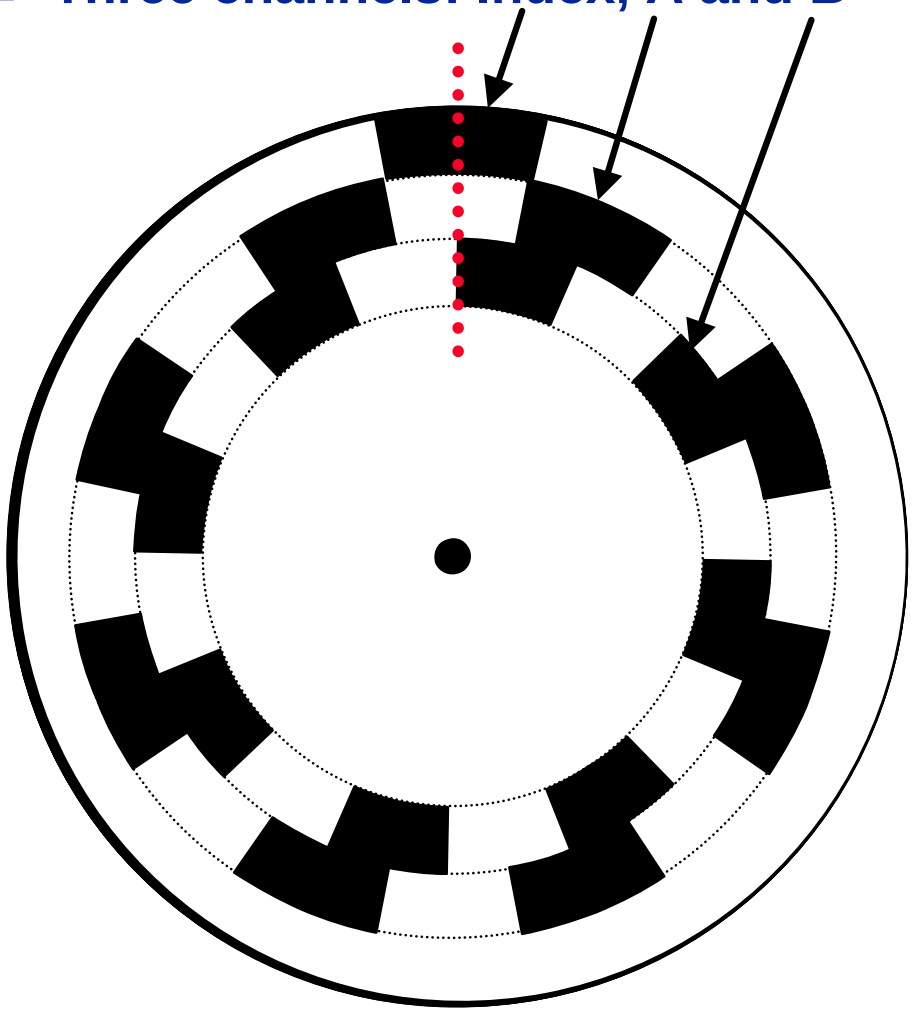
$I_{LED} = (V_{dc1} - V_{LED}) / R_1$

Component type #2

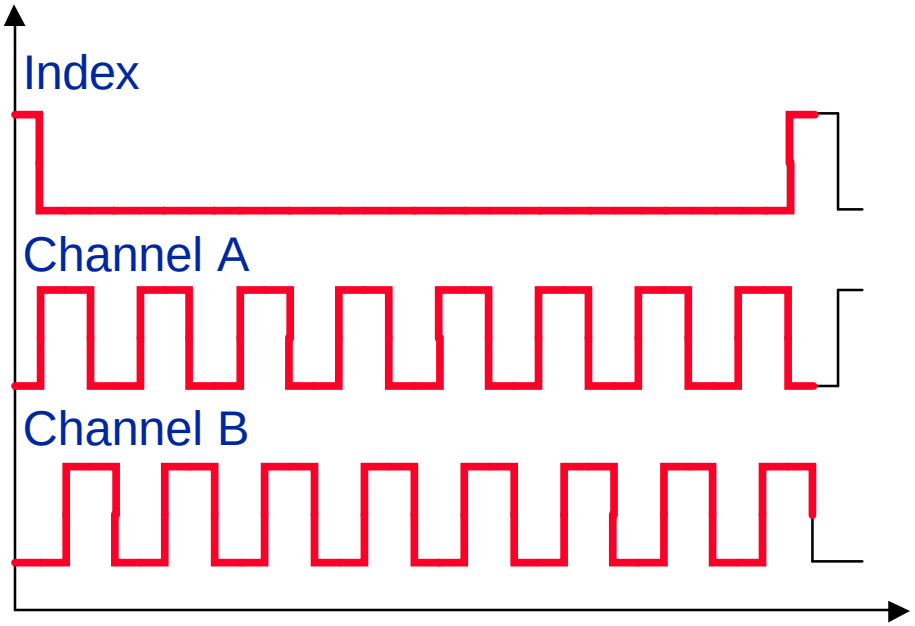


Circular Encoder Operation

- Three channels: Index, A and B

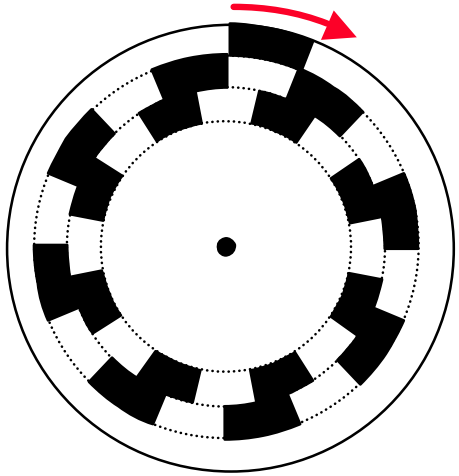


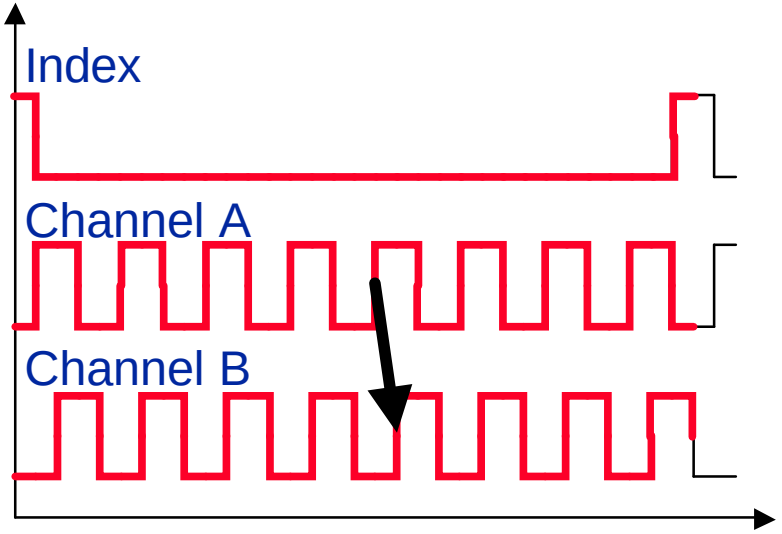
- Dark is logic high (circuit dependent)
- Frequency gives speed



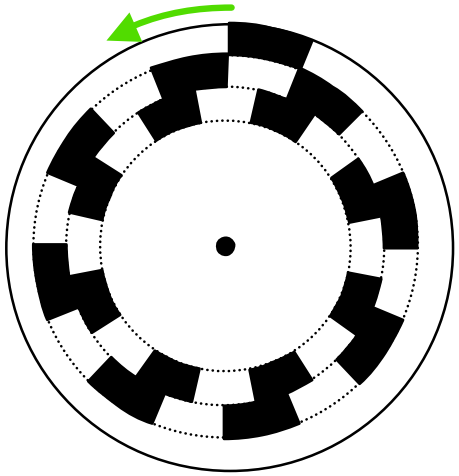
Quadrature Encoder

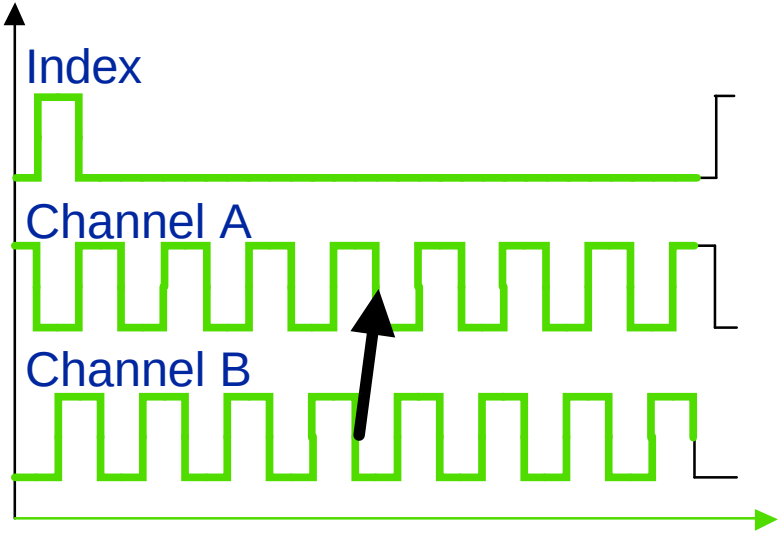
- A and B are in quadrature (90° phase difference) to detect direction
- Clockwise: A leads B





- Counterclockwise: B leads A

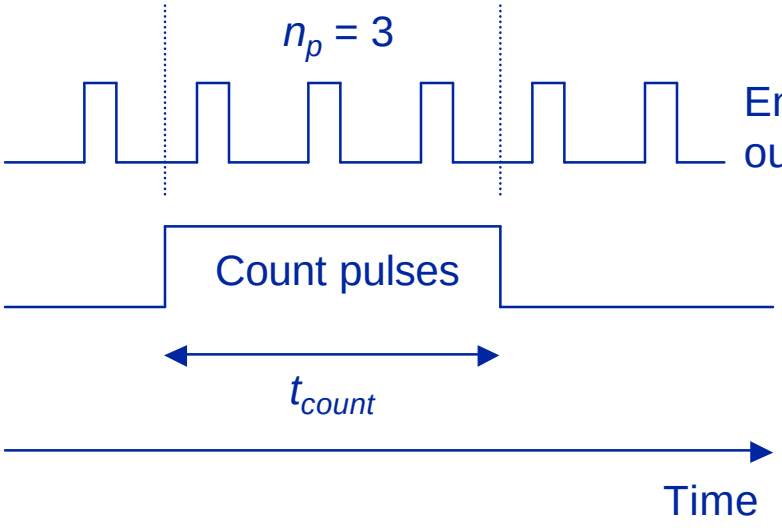




Measuring Shaft Speed

Method # 1

- ***Use timer in counting mode***
- ***Count the number of encoder pulses during a fixed time interval***
- ***Number of counted pulses is directly proportional to rotational speed***
- ***Resolution = 1 / (t_{count} • N • v_m)***



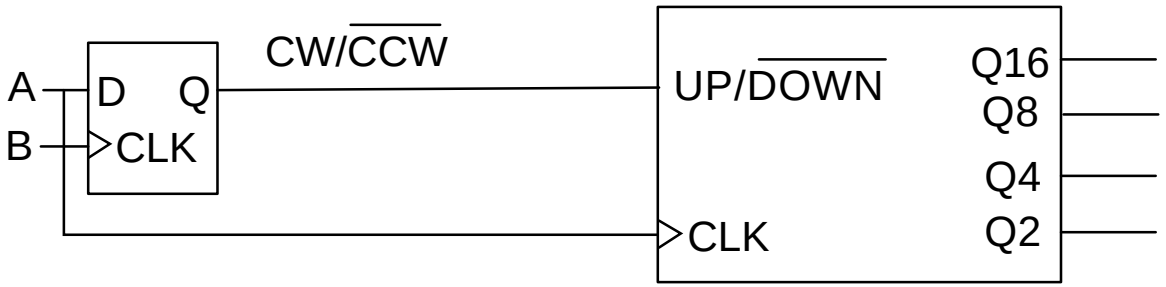
$$v_m = \frac{60 \cdot n_p}{N \cdot t_{count}}$$

N = encoder counts per rev.
n_p = number of pulses counted
v_m = motor speed in rpm
t_{count} = duration of counting interval

Counting with Interrupts

- Encoder signal can be fast
- For example…
  - HP HEDS-9X00 encoder has up to 512 counts/revolution
  - At 10 rev/sec, output is 5120 counts/sec = 5120 Hz
  - If interrupts were used to count pulses, microcontroller would be interrupted every 195 μs; no time for much else..
  - Use overflow interrupt; then 256 * 195 μs = 50 ms

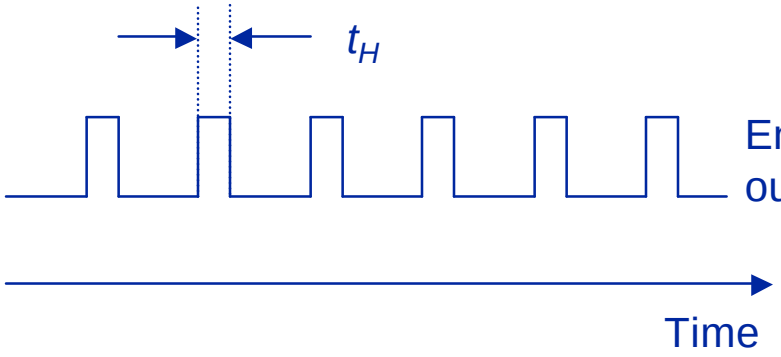
- Consider counting in external hardware
  - Can use divide by N digital counters
  - D flip-flop can be used for up/down count indicator

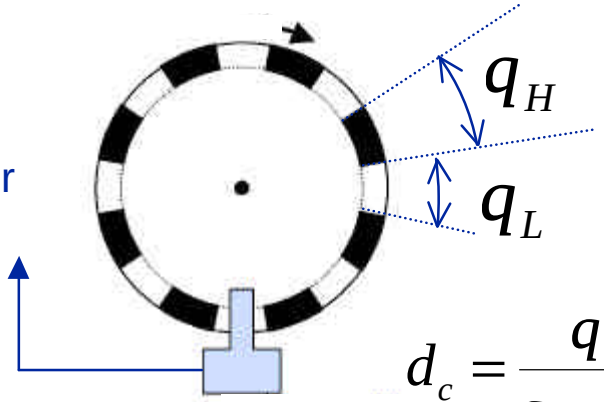


Measuring Shaft Speed (cont.)

Method # 2

- ***Use input capture***
- ***Measure the pulse width of the encoder pulses***
- ***Pulse width is inversely proportional to rotational speed***
- ***Resolution = timer clock period / pulse width***





$$d_c = \frac{q_H}{q_H + q_L}$$

N = encoder counts per rev.
d_c = duty cycle of encoder signal
v_m = motor speed in rpm

$$v_m = \frac{60 \cdot d_c}{N \cdot t_H}$$

Pulse Width Measurement Limitations

- For example…
  - HP HEDS-9X00 encoder has up to 512 counts/revolution
  - At 10 rev/sec, output is 5120 counts/sec = 5120 Hz
  - Pulse width = 98 μs
  - Fastest timer clock = 1 μs (with 4 MHz crystal)
  - Resolution = 1/98 = 1 % accuracy

- Turn on interrupts at some extended interval
  - Otherwise no time to do anything else..

Quadrature Clock Converters

- **LSI Computer Systems (lsicsi.com)**
  - **LS7082,3,4: Quadrature Clock Converters**
  - **LS7166: 24-Bit Quadrature Pulse Counter**
  - **LS7266R1: 24-Bit Dual-Axis Quadrature Pulse Counter**

8-bit data

A, B inputs

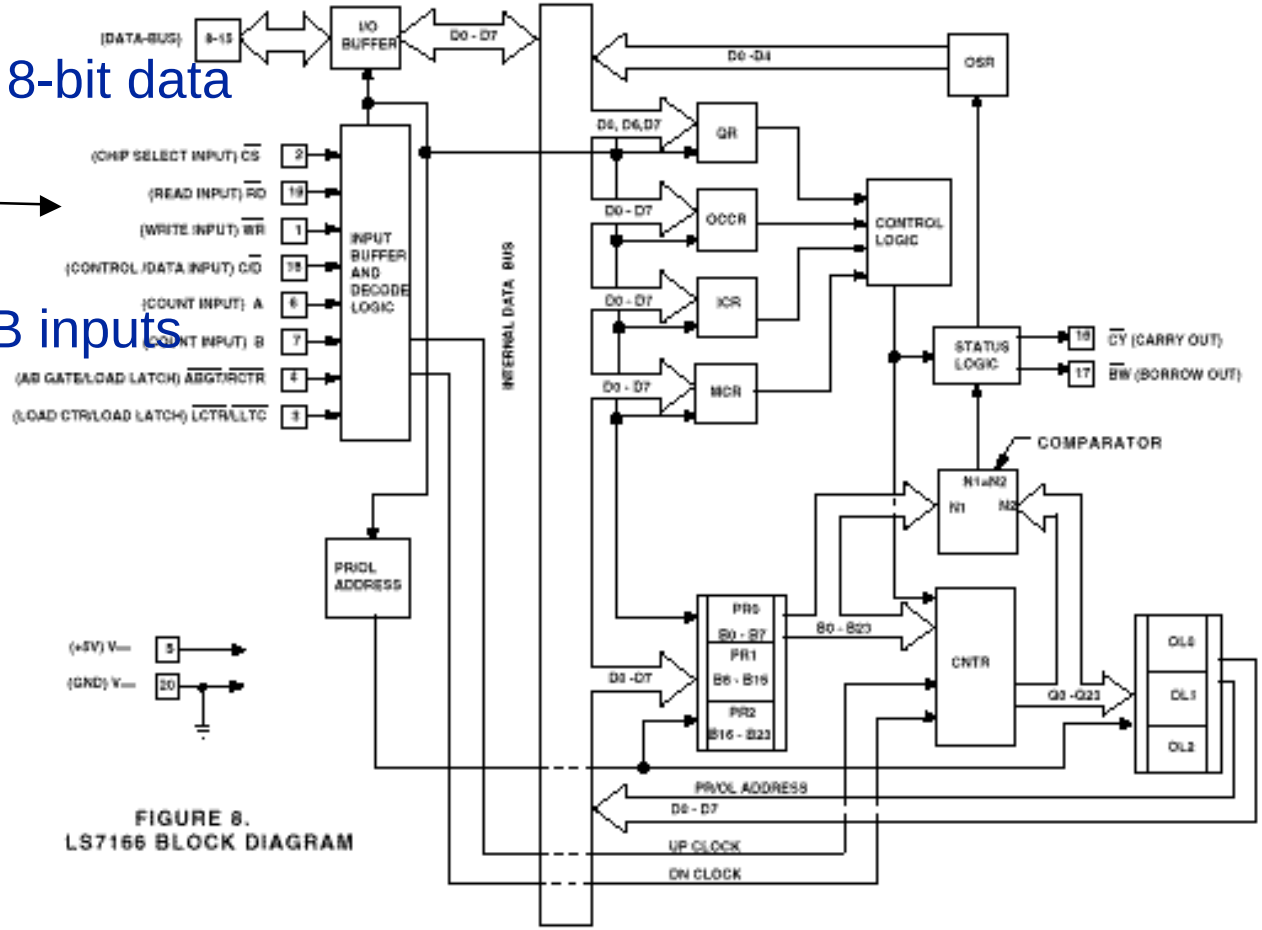


FIGURE 8.
LS7166 BLOCK DIAGRAM

24-bit count