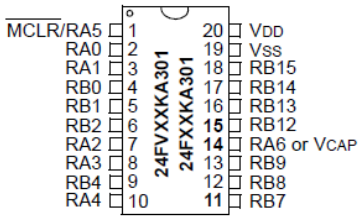
ME EN 330 Mechatronics Cheat Sheet

James Wade

May 7, 2025

Pin Diagrams

20-Pin SPDIP/SSOP/SOIC(1)



Pin	Pin Features	
	PIC24FVXXKA301	PIC24FXXKA301
1	MCLR/VPP/RA5	MCLR/VPP/RA5
2	PGEC2/VREF+/CVREF+/AN0/C3INC/SCK2/CN2/RA0	PGEC2/VREF+/CVREF+/AN0/C3INC/SCK2/CN2/RA0
3	PGED2/CVREF-/VREF-/AN1/SDO2/CN3/RA1	PGED2/CVREF-/VREF-/AN1/SDO2/CN3/RA1
4	PGED1/AN2/ULPWU/CTCMP/C1IND/C2INB/C3IND/U2TX/SDI2/OC2/CN4/RB0	PGED1/AN2/ULPWU/CTCMP/C1IND/C2INB/C3IND/U2TX/SDI2/OC2/CN4/RB0
5	PGEC1/AN3/C1INC/C2INA/U2RX/OC3/CTED12/CN5/RB1	PGEC1/AN3/C1INC/C2INA/U2RX/OC3/CTED12/CN5/RB1
6	AN4/SDA2/T5CK/T4CK/U1RX/CTED13/CN6/RB2	AN4/SDA2/T5CK/T4CK/U1RX/CTED13/CN6/RB2
7	OSCI/AN13/C1INB/C2IND/CLKI/CN30/RA2	OSCI/AN13/C1INB/C2IND/CLKI/CN30/RA2
8	OSCO/AN14/C1INA/C2INC/CLKO/CN29/RA3	OSCO/AN14/C1INA/C2INC/CLKO/CN29/RA3
9	PGED3/SOSCI/AN15/U2RTS/CN1/RB4	PGED3/SOSCI/AN15/U2RTS/CN1/RB4
10	PGEC3/SOSCO/SCLKI/U2CTS/CN0/RA4	PGEC3/SOSCO/SCLKI/U2CTS/CN0/RA4
11	**U1TX/C2OUT/OC1/IC1/CTED1/INT0/CN23/RB7**	**U1TX/INT0/CN23/RB7**
12	SCL1/U1CTS/C3OUT/CTED10/CN22/RB8	SCL1/U1CTS/C3OUT/CTED10/CN22/RB8
13	SDA1/T1CK/U1RTS/IC2/CTED4/CN21/RB9	SDA1/T1CK/U1RTS/IC2/CTED4/CN21/RB9
14	**VCAP**	**C2OUT/OC1/IC1/CTED1/INT2/CN8/RA6**
15	**AN12/HLVDIN/SCK1/SS2/IC3/CTED2/INT2/CN14/RB12**	**AN12/HLVDIN/SCK1/SS2/IC3/CTED2/CN14/RB12**
16	AN11/SDO1/OCFB/CTPLS/CN13/RB13	AN11/SDO1/OCFB/CTPLS/CN13/RB13
17	CVREF/AN10/C3INB/RTCC/SDI1/C1OUT/OCFA/CTED5/INT1/CN12/RB14	CVREF/AN10/C3INB/RTCC/SDI1/C1OUT/OCFA/CTED5/INT1/CN12/RB14
18	AN9/C3INA/SCL2/T3CK/T2CK/REFO/SS1/CTED6/CN11/RB15	AN9/C3INA/SCL2/T3CK/T2CK/REFO/SS1/CTED6/CN11/RB15
19	VSS/AVSS	VSS/AVSS
20	VDD/AVDD	VDD/AVDD

Abstract

For my own sanity, I have compiled this quick reference sheet to help me code quicker in PIC24. I hope this is also useful for you too! This document contains an updated list of the pin functions, registers, and calculations that we have learned. There is also a FAQ section covering some questions. The order of the sections generally follows the order of coding in PIC24.

Contents

1 PIC24 Setup

1.1 Oscillator

In MPLabX, you need to configure which oscillator to use. This must be configured before the int main() function. The available oscillators are:

- 8 MHz Fast RC (FRC or 000)
- 8 MHz FRC with Postscalar (FRCDIV or 111) - this is the default with a postscalar of two
- 500 kHz FRC with Postscalar (LPFRC or 110)
- 31 kHz Low-Power RC (LPRC or 101)

For an oscillator with postscalar capabilities, Table 1 presents the possible postscalers and what number you would input into _RCDIV:

Postscalar	1	2	4	8	16	32	64	256
Binary code	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111

Table 1: Table showing the possible postscalar options

Configure Oscillator

```
#pragma config FNOSC = XXXX //replace this with one of the options above

int main() {
    _RCDIV = 0b101 //postscaler of 64. The default is 2
    ....
    return 0;
}
```

To calculate what F_{osc} (the oscillator frequency) and F_{cy} (the clock frequency) is, use Equations 1 and 2.

$$F_{osc} = \frac{F_{osc,raw}}{postscaler, n} \tag{1}$$

$$F_{cy} = F_{osc}/2 \tag{2}$$

$F_{osc,raw}$ denotes the "undivided" or "max" oscillator frequency if the postscaler was 1. Notice that the clock frequency F_{cy} is *always* one-half the oscillator frequency after the postscaler has been applied.

1.2 Timers

To count the number of clock ticks, you can use a timer. The basic setup requires 4 settings to be configured: the period (PRx), the prescaler (TxCONbits.TCKPS), timer clock source (TxCONBITS.TCS), and the on/off state (TxCONbits.TON). This is shown below.

Configure Timer1

```
int main() {
...
T1CONbits.TON = 1; // start timer
T1CONbits.TCS = 0; // select internal clock
T1CONbits.TCKPS = 0b10; // select prescale of 64
PR1 = 45,625; // set the period in clock ticks
TMR1 = 0; // reset Timer1 to zero;
...

if (TMR1 == 27000) { // this is how you read or "poll" a timer
    ...
    TMR1 = 0; // this is how you write to a timer
    // the timer now is set back to zero and will count up again
}
...
return 0;
}
```

For each individual timer, Table 2 the available prescalers and the associated binary code to input into .TCKPS (timer clock prescale select).

Prescalar	1	8	64	256
Binary code	0b00	0b01	0b10	0b11

Table 2: Table showing the possible prescalar options

To calculate what prescaler to implement, use equation 3, where t_{total} is the total time in seconds that the timer will count for, and "max_timer_counts" is the maximum clock ticks that the 16-bit timer can hold. For a 16-bit timer, the max number of ticks it can measure is 2^{16}, which is equal to 65, 536 counts. In solving equation 3 for the prescale value, odds are that the result will not be a nice even 1, 8, 64, or 256. In that case, you round up to the next closest prescale value. For example, if you calculated that a prescale value of 9.7 is needed, you must round up to the 64 prescale value. Because your timer is now counting slower than needed (a 64 prescale means the timer only increments every 64 clock ticks instead of 9.7), your timer period does not and should not be the max number of counts it can hold, because each count of the timer now represents more ticks of the clock. In other words, the timer is now operating at a slower rate, so it needs to count less because of that. Hence, after calculating the needed prescaler, plug that into equation 3 and solve for the max_timer_counts. That is now your period in clock ticks that you should input into your code (PR1).

$$t_{total} = \frac{max\_timer\_counts\ x\ prescale}{F_{cy}} \tag{3}$$

2 Input/Output

Note: For the digital IO and PWM section, I used pin 4, as it is able to do digital IO (RB0), analog input (AN2), and PWM (OC2). BUT, it is unable to do more than just one of these roles. Remember that there are 7 pins in the TRISA register and 11 pins in the TRISB register

Good Practice for Beginning of Any Program

```
ANSA = 0;
ANSB = 0;
TRISA = 0;
TRISB = 0;
LATA = 0;
LATB = 0;
```

2.1 Digital Input/Output

Digital Input

```
//Set input direction
TRISB = 0b00000000001; //TRISAbits.TRISB0 = 1 or _TRISB0 = 1;

ANSB = 0; // turns off analog mode for all pins in register B.

//To read from a pin(s) into a variable called x
int x;
x = PORTB; //reads the status of all B pins
x = PORTBbits.RB0 //reads from pin 4. Shorthand is x = _RB0;
```

Digital Output

```
//Set input or output direction
TRISB = 0b00000000000; //TRISAbits.TRISB0 = 0 or _TRISB0 = 0;

ANSB = 0; // turns off analog mode for all pins in register B.

//To write to pin 4, use the Latch register (LATx). 1 is high, 0 is low
LATBbits.LATB0 = 1; // or _LATB0 = 1;
```

2.2 PWM - Pulse Width Modulation

So far, most other peripherals (such as timers, digital input/output, etc) have only needed one register for us to change the appropriate settings. But since PWM is very complicated, two registers (OCxCON1 and OCxCON2) are needed. It is important to "clear" them all to zero so as not to have any residual settings from previous code hanging around in the memory.

PWM

```
OC1CON1 = 0; // clear OC1CON1 register
OC1CON2 = 0; // clear OC1CON2 register

/* the following four lines are standard and should be used without any problem */
OC1CON1bits.OCTSEL = 0b111; // set system clock as basis for PWM signal
OC1CON1bits.OCM = 0b110; // edge-aligned PWM signal
OC1CON2bits.SYNCSEL = 0b11111; // setting the synchronization source
OC1CON2bits.OCTRIG = 0; // triggers OC1 from the source designated by .SYNCSEL bits

OC1RS = 499; // set the period of the PWM signal in clock ticks
OC1R = 166; // set the duty cycle of the PWM signal in clock ticks
```

In order to find out what the period needs to be in terms of clock ticks, use equation 4.

$$PWM\_period = [OCxRS + 1]\ x\ T_{cy}\ x\ Prescale\_value \tag{4}$$

If OC1CON1bits.OCTSEL = 0b111, then the PWM signal is using the clock as its timing source (instead of a timer), so the prescale_value is equal to 1. PWM_period is in seconds, and OCxRS is in clock ticks.

3 Interrupt Pins/Event-Driven Programming

3.1 Interrupt Pins

Interrupts let the program instantly drop whatever it's doing and execute a specified section of code. This is useful for programs with complicated programs that can't always check the state of a pin normally. So, if a sensor is triggered and it "throws" (i.e. raises or yells out) a flag, then the program will pause momentarily and address the flag that's been thrown. There are multiple types of interrupts, such as timer interrupts (when the timer gets set back to zero) and event interrupts (when something of interest is measured to have happened, like an infrared light being detected). There are multiple types of interrupts.

- **Timer Interrupts - Tx** - These are triggered when that specific timer's period is reached. So if your timer counts up to 10ms, then the timer interrupt will trigger every 10 ms. Each timer (1-5) has their own interrupt function that is called.
- **Change Notification Interrupt - CNx** - These are called when any digital state change is detected on a pin. All of the I/O pins on the PIC24 have CN capabilities. If the pin goes high, the interrupt is triggered. And when that same pin goes low, the interrupt is triggered again. All of these interrupt

pins call the same interrupt function, leading to some complications in determining which pin was triggered.

- **External Interrupts - INTx** - These are exactly like CN interrupts, except for two major aspects: each external interrupt pin calls its own interrupt function (there are 3 on the pic24), and these pins can be configured to only trigger when a certain state is reached. Examples could be when the pin goes high (but not low), when the pin goes low (but not high), or when the pin is in a certain range. This provides more flexibility over CN interrupts, but they're limited in number.
- **Output Compare Interrupts - OCx** - Each output compare pin (OC1-3) has its own interrupt function that it calls. Each leading edge (when it goes high) or falling edge (when it goes low) of the PWM signal triggers the interrupt. Whether its the leading or falling edge is configured in the code using the _INTxEP bit. This is useful for counting steps in a stepper motor.

Timer Interrupts

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {
    _T1IF = 0; // Clear interrupt flag
    // Do something here
}

int main() {
    ...
    // Configure Timer1 in usual way
    ...
    // Configure Timer1 interrupt
    _T1IP = 4; // Select interrupt priority (1-7, 7 being highest)
    _T1IF = 0; // Clear interrupt flag
    _T1IE = 1; // Enable interrupt
    PR1 = 10000; // Set period for interrupt to occur (this value depends on
                 // the time that you want to elapse between interrupts)

    // Do nothing { or do something cool in the loop
    while(1);

    return 0;
}
```

Change Notification Interrupts

```
void __attribute__((interrupt, no_auto_psv)) _CNInterrupt(void) {
    _CNIF = 0; // Clear interrupt flag
    // Do something here
}

int main() {
    _CN5IE = 1; // Enable interrupt on pin 5
    _CN5PUE = 0; //Disable pull-up resistor
    _CNIP = 4; // Select interrupt priority (4 is default)
    _CNIF = 0; // Clear interrupt flag
    _CNIE = 1; // Enable CN interrupts
    while(1);

    return 0;
}
```

External Interrupts

```
void __attribute__((interrupt, no_auto_psv)) _INT2Interrupt(void) {
    _INT2IF = 0; // Clear interrupt flag
    // Do something here
}

int main() {
    _INT2IP = 4; // Select interrupt priority (4 is default)
    _INT2IF = 0; // Clear interrupt flag
    _INT2EP = 1; // Set edge detect polarity to positive edge
    _INT2IE = 1; // Enable interrupt
    while(1);

    return 0;
}
```

Output Compare (PWM) Interrupts

```
#include "xc.h"
#pragma config FNOSC = LPRC //31 kHz oscillator

void __attribute__((interrupt, no_auto_psv)) _OC2Interrupt(void) {
// triggers each PWM period
    _OC2IF = 0; // Clear interrupt flag
    // Do something here
}

int main() {
    ...
    OC2CON1 = 0;
    OC2CON2 =0;
    OC2CON1bits.OCTSEL = 0b111;
    OC2CON2bits.SYNCSEL = 0b11111;
    OC2CON2bits.OCTRIG = 0;
    OC2CON1bits.OCM = 0b110; //set to 0 to turn off PWM
    OC2RS = 38; //200 steps in 0.5 seconds
    OC2R = 19;

    //Setup interrupt on OC2
    _OC2IP = 4; //Select the interrupt priority
    _OC2IF = 0; //clear flag
    _OC2IE = 1; //enable the OC2 interrupt

    return 0;
}
```

3.2 Event-Driven Programming

Event-driven programming is a type of programming that allows the robot to react to its environment (or "events"). For example, if the robot encounters a blue ball instead of a red ball, it needs to react in a different way. There are three aspects of event-driven programming that must be met for it to be effective:

1. Event checkers must run continuously and often. Ex) The program should check if the ball is red or blue often instead of every 2 seconds.
2. Services must be quick so that the program can go back to scanning for other events. Ex) The program should move the ball into its respective container quickly so as not to miss another ball.
3. No blocking of code. Ex) Don't input a set amount of "wait time" which clogs up the processing speed of event checkers or services.

To accomplish this, a common way to implement event-driven programming is through a finite-state machine. It's called a "finite-state" machine because there are a finite number of modes or "states" that

we want the robot to be in (i.e. ball-moving state, line-following state, canyon-navigating state). To make a finite-state machine (FSM), we usually use switch statements and enumerate data types. The enumerate data type is used to hold what "state" the robot is in. Then within each "case" branch of the switch statement we have event checkers (such as lightSensedLeft()) which would check for an event (like a light sensed to the left of the robot) and would change the state of the robot if the event-checker turns true.

Finite-State Machine

```
enum {UP, DOWN, LEFT, RIGHT} dir;
// enum specifies that you're making an enumerate data type
// {UP, DOWN, LEFT, RIGHT} are all the possible values in this user-defined data type
// dir is the variable that is set to one of those values within {...}

int main() {
    ...

    switch (dir) {
        case UP:
            //go forward
            if (lightSensedLeft()) {
                dir = LEFT;
            }
            else if (lightSensedRight()) {
                dir = RIGHT;
            }
            else if (lightSensedBack()) {
                dir = DOWN;
            }
            break;
        case DOWN:
            //go backward
            break;
        case LEFT:
            //go left
            break;
        case RIGHT:
            //go right
            break;
    }
}
```

4 Diodes

Diodes are one-way latches. They allow current to follow only from the anode to the cathode. When voltage is applied as shown in figure 1, it is called to be "forward-biased". This allows current to flow.

When the applied voltage is reversed, no current flows. In order for the current to flow, a minimum voltage, V_F (forward voltage) has to be applied. If a strong enough reverse voltage is applied, the diode breaks down and current is allowed to flow in the other direction. This is not desired behavior. Diodes have 5 important characteristics to look for in a datasheet:

- **Forward Voltage Drop (V_F)**: Voltage drop in forward-biased configuration
- **Reverse Breakdown Voltage (V_R)**: Reverse voltage at which the diode will break. Also known as peak inverse voltage (PIV)
- **Forward current-handling capacity (I_{o_max})**: Maximum forward current that can pass through the diode
- **Response time (t_R)**: Time for diode to switch on and off
- **Reverse leakage current ($I_{R,max}$)**: Current that flows in reverse biased configuration

It is important to note that a small amount of current does flow in voltages smaller than V_F, and some current will flow when reversed-biased. The simplified model of the diode does not allow current to flow in either of these situations. So, just be aware of this behavior when using diodes. The datasheet for each diode has a graph similar to figure 2. The vertical dotted line represents V_F. Notice that the reverse-biased current is on the scale of nA, so very insignificant.

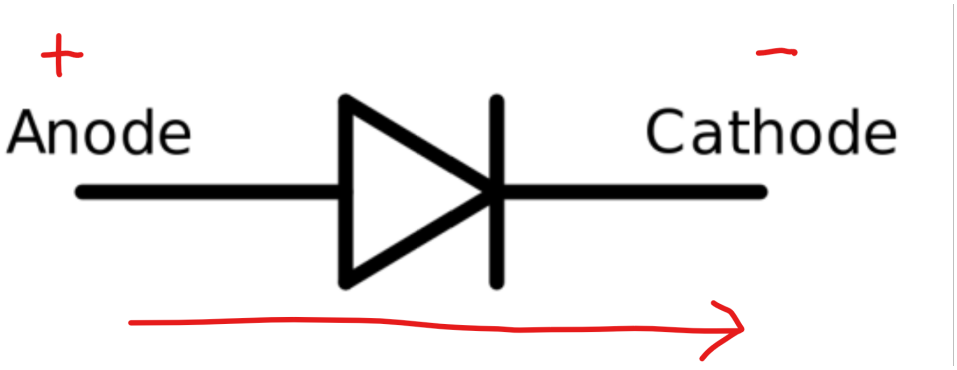


Figure 1: Diagram of diode

5 Transistors

Transistors are digital switches. They form the basis of all logic circuits, such as op-amps, H-bridges, and almost all other components that use bit logic. There are two main types of transistors, each with two subtypes.

- **BJTs (Bipolar Junction Transistors)**: A BJT has three pins - a collector, an emitter, and a base. In order for these to open, a current must either be added into or subtracted from the "base" pin. There are two types of BJTs, as listed below. BJTs are cheaper than MOSFETs, but the requirement to add or subtract current means that the power flowing in and out of the emitter and collector pins is affected (that extra current applied at the base has to go somewhere).

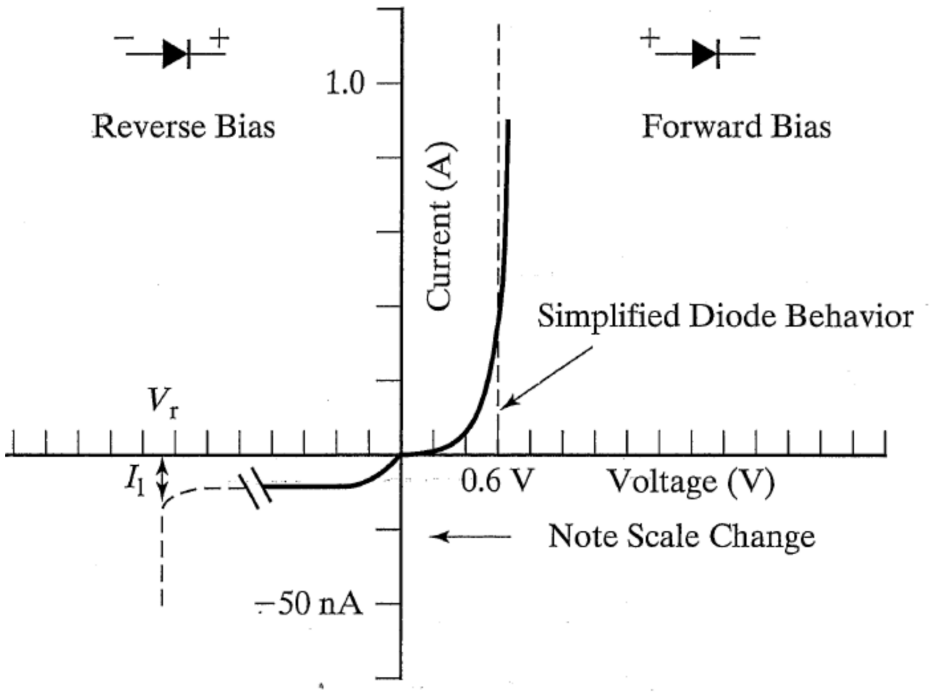


Figure 2: Graph showing the current flow through a diode based on applied voltage

  - NPN BJTs: These switch on when a high voltage is applied to the base pin, flowing current into the base pin. These are used "downstream" from the load (ie after your motor or your laser or whatever else), such as right before ground.
  - PNP BJTS: These switch on when a low voltage is applied, flowing current away from the base pin. These are used "upstream" from the load, such as right after the power supply.

- **MOSFETs**: A MOSFET has there pins - a source, a drain, and a drain, and a gate pin. To open these switches, a voltage (either high or low) must be applied to the gate pin. There are two types, as described below. While these are more expensive than BJTs, they are voltage-driven (rather than current-driven). This means that no extra energy is added to the current that flows between the drain and the source. These are generally better to use than BJTs.

  - NMOS (N-type MOSFET): A low voltage turns off this type of transistor. They are used after the load and typically right before ground. The PSC only has this type of MOSFET.
  - PMOS (P-type MOSFET): A high voltage turns off this type of transistor. They are used before the load, typically right after the voltage source.

Since MOSFETs are what we will use in this class, figure 3 shows the effect of low, intermediate, and high applied voltages to both NMOSs and PMOSs. Notice the varying brightness level that corresponds with the specific applied voltage. Also notice where the MOSFET is in relation to the load (ie the LED). The PMOSs are before the load, and the NMOSs are after the load.

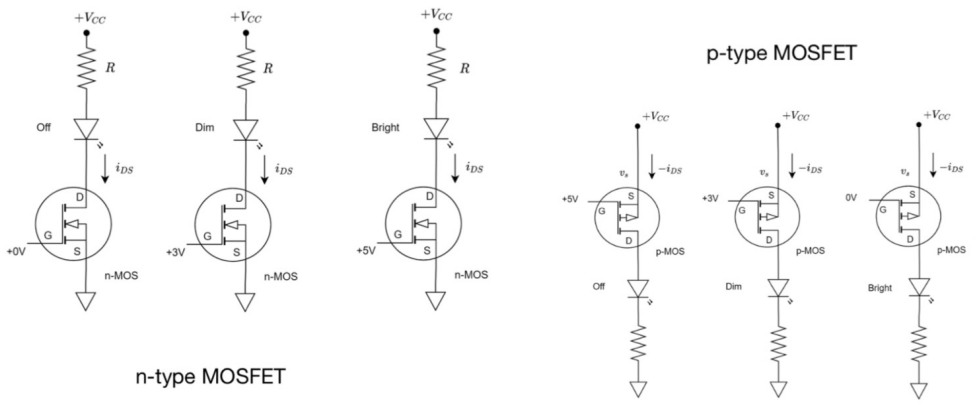


Figure 3: NMOS and PMOS behavior for various applied voltages

6 DC Motors

DC Motors work by translating incoming electrical current into a magnetic field. This magnetic field is pulled by magnets in the casing and it turns. As such, they are fast but they're not accurate if you're desiring specific lengths. To control one with an H-bridge, you need three pins: a digital output set to low, a digital output set to high, and a PWM pin with a frequency of 1500 Hz and a duty cycle that controls the speed. The higher the duty cycle, the faster the motor goes. The two digital output pins control the direction. To switch directions, change the polarity of the two digital output pins.

7 Stepper Motors

Due to the complicated nature of stepper motors, we use a stepper motor driver to control it. It's like an electric black box - we know what goes into it and we know what will come out, and we don't care about what's going on within the component. A stepper motor is controlled by the period of the PWM signal, NOT its duty cycle. Hence, always put the duty cycle at 50%. Equation 5 shows the necessary calculation for knowing how long in seconds (t) it will take to make the specified # of revolutions.

$$t = \frac{(\text{Steps per rev}) * (\#\text{ of rev})}{F_{pwm} * (\text{Stepping Mode})} \tag{5}$$

Note that t is the time in seconds to achieve the specified # of revolutions. This equation can be rearranged to get the more general form of revolutions/sec."Steps per rev" is a property of the stepper motor. This is the number of steps needed for one revolution in full-stepping mode. For most stepper motors, this number is 200, though checking the datasheet is always a good idea. The PWM frequency can be calculated from the equation in the PWM section.

8 Servo Motors

Servo motors require 3 wired connections: ground, power, and the PWM control signal. A servo motor NEEDS the PWM to operate a frequency of 50 Hz (unless the datasheet says otherwise). This is not negotiable. The servo motor will not work if you don't do this. This translates to a PWM period of 20ms. To turn the servo to 0°, the duty cycle is 1ms. To turn the servo to 180°, the duty cycle is 2ms. Some

servo motors, either due to manufacturing inconsistencies or a purposeful design, need duty cycles ranging from 0.5ms to 2.5ms. Be sure to check the datasheet and experiment with your servo motor. To control the servo motor, follow the instruction in the PWM section.

9 Motor Review

Here is a quick summary of how to control each of the three motors:

1. **DC Motors**
    - *Direction* - Two digital output signals, one high and one low
    - *Speed* - One PWM pin, usually at 1500 Hz period. Duty cycle changes the speed (higher the duty cycle, higher the speed)
2. **Stepper Motors**
    - *Direction* - One digital output pin (high or low)
    - *Speed* - One PWM pin, set at 50% duty cycle. The period changes the speed (the higher the frequency/the lower the period means higher the speed)
3. **Servo Motors**
    - *Direction* - One PWM pin, set at a frequency of 50 Hz (period of 20 ms). A stepper motor only has a moveable range of 180°. 0°typically corresponds to a duty cycle of 1ms. 180°typically corresponds to a duty cycle of 2ms. Though check the datasheet and experiment with the servo to verify
    - *Speed* - The only way to effect the speed is to send it either 1) smaller degree step sizes or 2) put it a delay

10 Batteries

Here are some key terms for batteries:

- **Cell:** one electrochemical unit to provide electrical power
- **Battery:** an assembly of cells in series or parallel
- **Primary:** non-rechargeable
- **Secondary:** rechargeable
- **Cathode:** positive terminal
- **Anode:** negative terminal

Here are some key performance and characteristic specs found on a datasheet:

- **Capacity (C):** Amount of energy that a battery is capable of storing, in units of Ah (Amp-hour) or mAh.
- **Cutoff voltage (end voltage):** The voltage at which a battery is "fully discharged".

- **C-rate:** Rate at which current is drawn relative to the capacity. C-rate = current/capacity. A C-rate of 1 means a discharge rate equal to the battery's capacity.
  - *Note:* For a battery with a nC rating, the battery will discharge in 1/2 hours. Ex) 2C means it will discharge in 1/2 hours. $\frac{1}{3}$C means it will discharge in 3 hours.
- **Energy Density**: Amount of energy per unit mass (Wh/kg) or volume (Wh/L)
- **Self-Discharge:** The rate at which the battery's capacity decreases when not in use
- **Internal Resistance:** Resistance to current flow within the battery. Lower means less heat loss and higher max current flow (lower output impedance)

It is also important to know that for Li-ion and Li-poly batteries, overcharging, physically damaging, or short-circuiting them can result in an explosion. Additionally, a deep discharge can also permanently ruin the battery as well. Hence, protection and monitoring circuits are needed to take care of the battery.

11 Op-Amps

Op-amps are considered a power element, as they require a power source to function (as opposed to a resistor, capacitor, or an inductor). The basic idea is $V_{out} = A(V_+ - V_-)$ with no other connections. In an ideal world, A is infinite (the voltage gain) and is in reality a really high number. There is infinite input impedance at the input terminals (no current flowing in), there is zero output impedance (no resistance as current exits), zero offset voltage (output voltage is zero when the two input voltages are the same), and infinite slew rate (ie the output can change instantaneously). Additionally, most of our applications use negative feedback (ie connecting the output back to the negative terminal), which makes $V_+ = V_-$.

12 IR-based Sensors

When it comes to IR-based sensors used in this class, there are two types of IR-sensing elements:

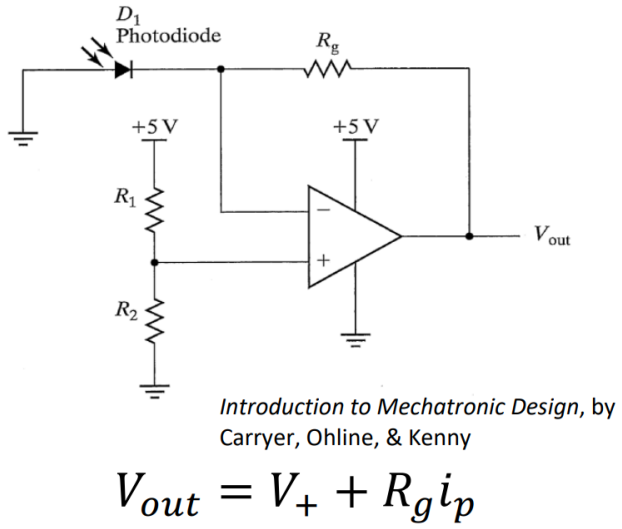
- **Photodiode:** Photodiodes act similar to normal diodes (section 4), in that they will let current flow when forward-biased and will block current when reverse-biased. *Except*, if the photodiode senses IR light, it will let current flow from its cathode to its anode when reverse-biased (ie opposite the normal flow you would see in a diode). This current is roughly linear to the light irradiance. Each photodiode is most sensitive within a certain range of wavelengths (we need to use photodiodes that accept 940 nm wavelength light). To convert this current across the photodiode, use a circuit as shown in figures 4a and 4b.
- **Phototransistor:** This is like a transistor opened by IR light. It can have larger output, but it also has a narrower range of wavelengths (which could be an advantage) and a slower response time. To use a phototransistor, use one of the circuits shown in figures 5a and 5b.

13 Frequently Asked Questions

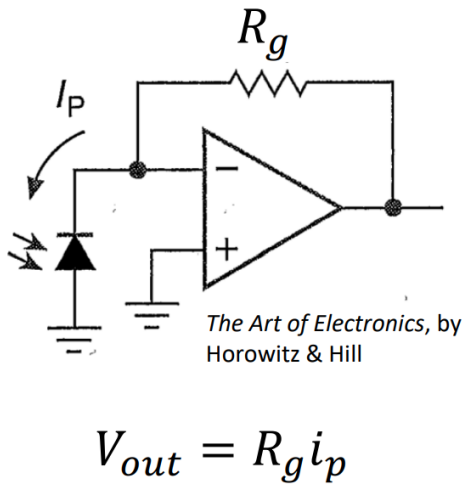
13.1 Hardware

What is an oscillator vs a clock?

Imagine a metronome (or look up a picture of one). It has a hand that rotates back and forth,



(a) Op-Amp and Photodiode Circuit 1



(b) Op-Amp and Photodiode Circuit 2

Figure 4: Two different op-amp circuits to use in converting photodiode-induced current to voltage

marking the passing of time in a steady manner. An oscillator is like every swing, counting "left, right, left, right", etc. A clock is half that speed, only counting when the hand reaches its starting position on the left-hand side, counting "left, left, left". Therefore, if the oscillator runs at 8 MHz, the clock runs at 4 MHz (half the speed). Another example is a heart. A heart has two motions for a completed cycle: bringing in blood and pushing it out (bum-bum, bum-bum). Therefore, the oscillator is the heart. One complete cycle (ie two beats of the oscillator) is the clock.

How does a postscaler and a prescaler work?

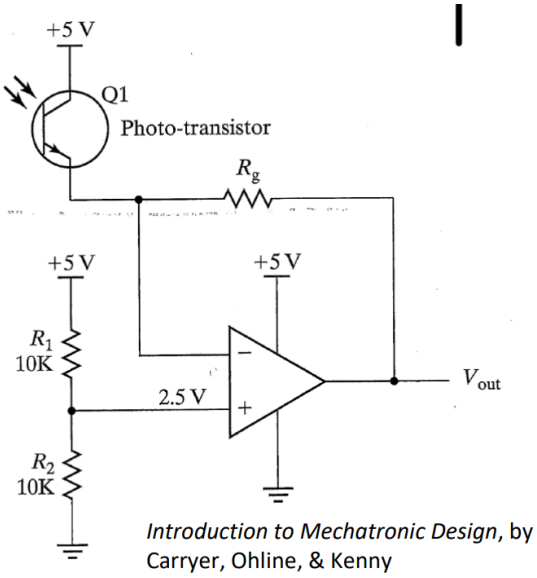
A postscaler and a prescaler are two things that fill similar needs. The postscaler slows down the oscillator, the heart of the microcontroller. Hence, everything on the microcontroller slows down. If you don't need to be super fast and have low-power constraints, then that's perfectly fine. But if you have a self-driving car that needs to detect things instantly, then a slower heart would not be sufficient. A prescaler slows down the timer. A prescaler of 64 means that for every 64 ticks of the clock, the timer increments (i.e. it adds 1 to itself).

What is digital input/output?

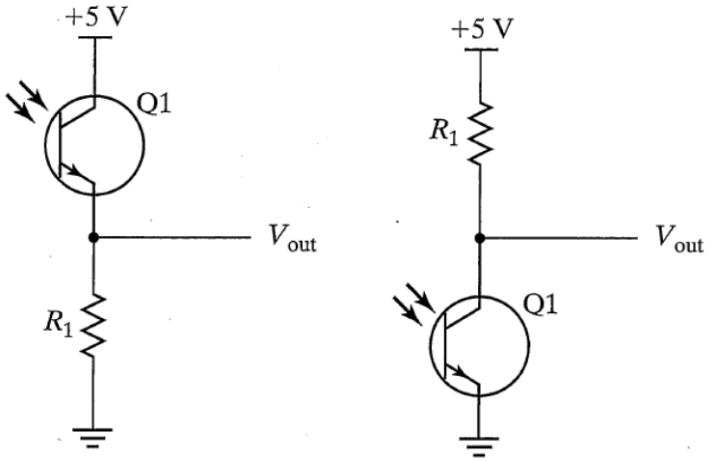
Digital input and output means "black and white" - on or off. An "off" voltage is a voltage between 0 and .6 volts. And "on" voltage is a voltage greater than 2.64 volts for the PIC24. Any voltage in between those ranges is undeterminable by digital input and will cause errors!

What is analog input/output?

Analog input means "not black and white" - i.e. the full range of voltages is possible. For example, sensor data from the real world would naturally fluctuate and give minute differences in the data returned. Imagine if this data could only be transported to the microcontroller in a "high" or "low" profile state. This data would be pretty useless as it would just look like a bunch of steps. Analog lets us produce and read more "natural" data curves. In other words, if a voltage is applied, the microcontroller is able to read exactly what level it is (1 volt, 2.3 volts, 1.999 volts, etc) and not just "high" or "low".



(a) Op-Amp and Phototransistor Circuit



(b) Pull-Up/Down Resistor and Phototransistor Circuit

Figure 5: Two different op-amp circuits to use in converting photodiode-induced current to voltage

What is a register?

In short, a register is a 16-bit location of memory which holds the settings for certain aspects of the PIC24. For example, to change the behavior/settings of Timer1, you would go to the register associated with Timer1, which is T1CON. A good analogy is that when two people get married, they have to give their wedding information to the wedding registrar. In other words, they have to register the data for it to be put into the system and become valid. Similarly, we have to manually put settings into the PIC24 for it to become valid

13.2 Coding

What is a compiler?

A compiler is a program that translates our english/human instructions (i.e. our written code) into something that the machine can read (one step closer to 1s and 0s). When we downloaded MPLabX, we downloaded the Xc16 compiler so that our computer could communicate with the PIC24.

What is #define and #pragma? And how are they different from a global variable?

All three of these must be declared before the int main() function. #pragma just indicates to the compiler that a non-C command (such as one specific to the PIC24) is about to happen. #define is a way of creating a global constant value that can have any name you like. For example, `#define _LATA6 LATAbits.LATA6` just tells the compiler to replace any instance of "_LATA6" with "LATAbits.LATA6". This #define statement is already present in the xc.h header file included at the top of your code. A global variable is a variable declared before the int main() function, and it can be accessed at any point in the code. It can also be assigned a different value during code execution, wheres a #define statement cannot be changed once the compiler has finished parsing the code.