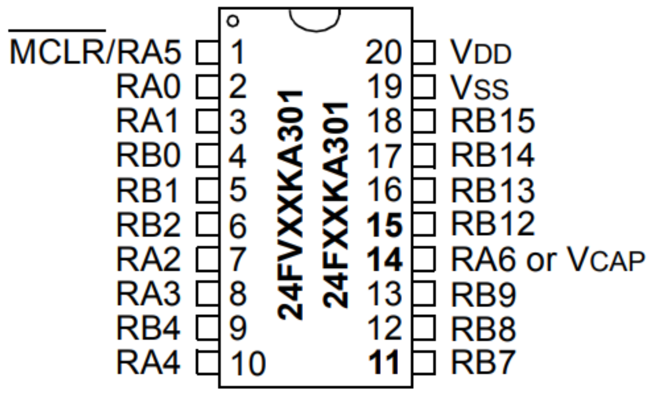
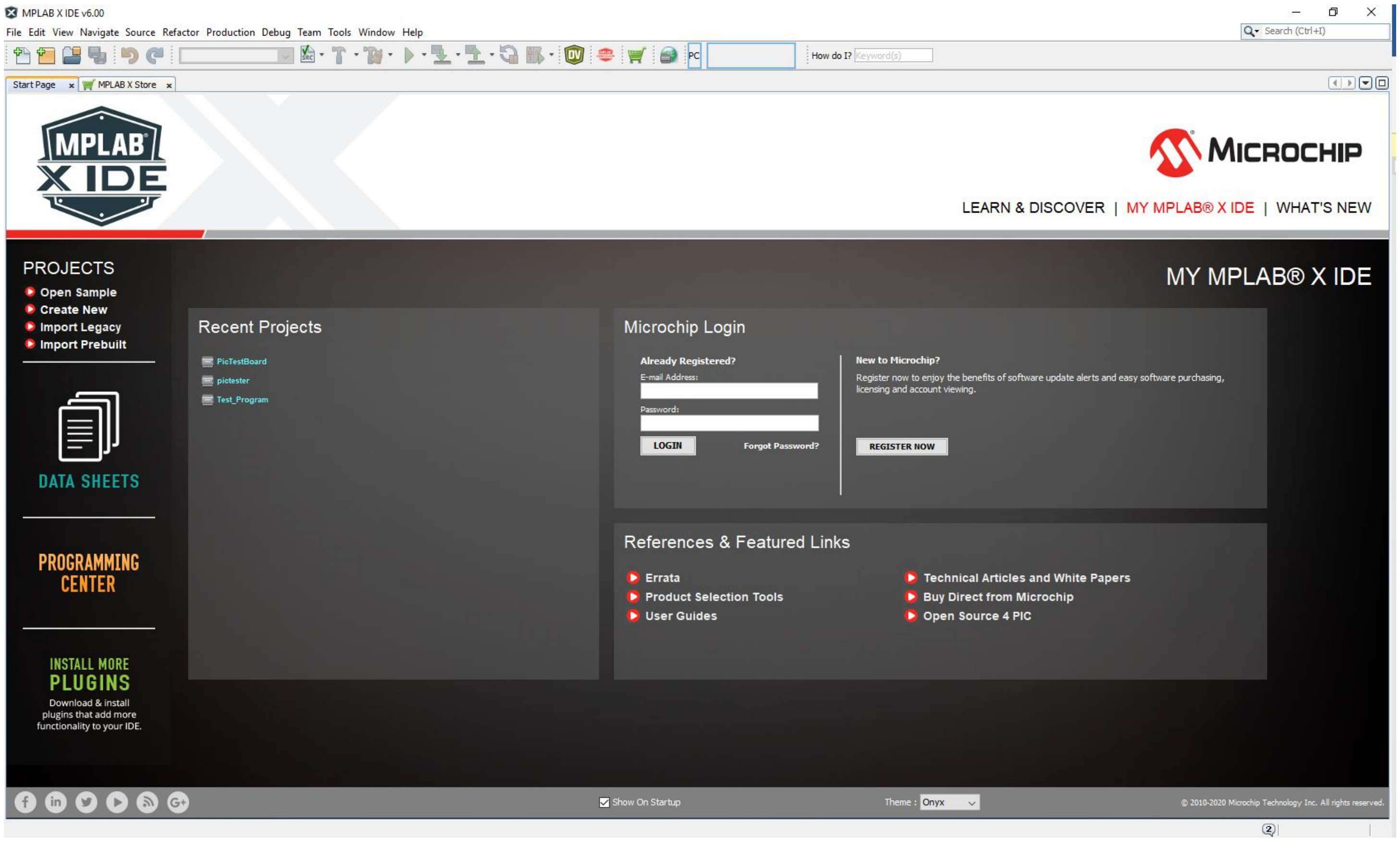
Programming a PIC24 in MPLAB X

Table of Contents



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

Making a Project and Navigating the IDE

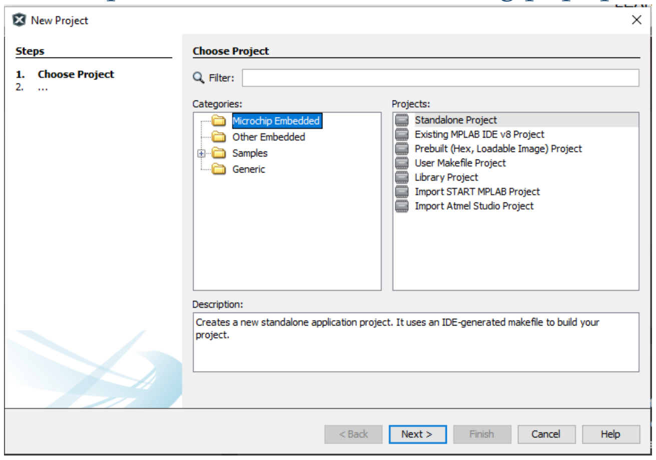


Initial Screen

Upon startup you should see the startup screen as shown above that has shortcuts to websites and functions of the IDE. From this start page we will either make or continue a previous project with the buttons in the top left corner. The manila folder with a plus on it creates a new project and the blue folder with a manila folder coming out opens a project.

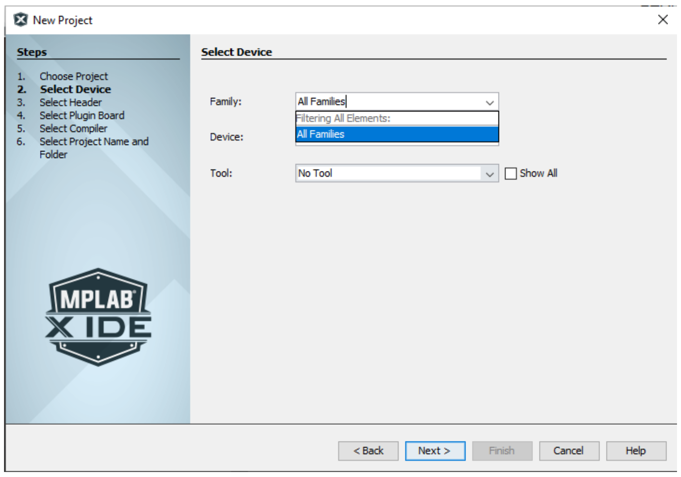
Creating a Project

Upon clicking the new project button, we are presented with the following popup:



Creates a new standalone application project. It uses an IDE-generated makefile to build your project.

From this point we select a standalone project and hit next at the bottom of the screen. The next window we see looks like the following:



New Project

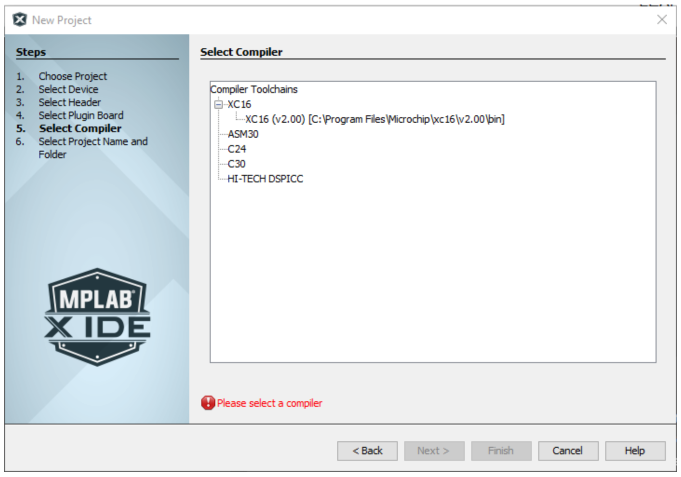
Steps
1. Choose Project
2. **Select Device**
3. Select Header
4. Select Plugin Board
5. Select Compiler
6. Select Project Name and Folder

Select Device

Family: All Families
Filtering All Elements:
All Families
Device:
Tool: No Tool
Show All

< Back | Next > | Finish | Cancel | Help

Here we need to define what device we are programming to. In the family drop down select 16-bit MCUs (PIC24) and in the device dropdown select PIC24F16KA301. If you have a PicKit3 plugged in you can select that under the tool drop down or you can just skip that section and hit next. The next menu looks like the following:



New Project

Steps
1. Choose Project
2. Select Device
3. Select Header
4. Select Plugin Board
5. **Select Compiler**
6. Select Project Name and Folder

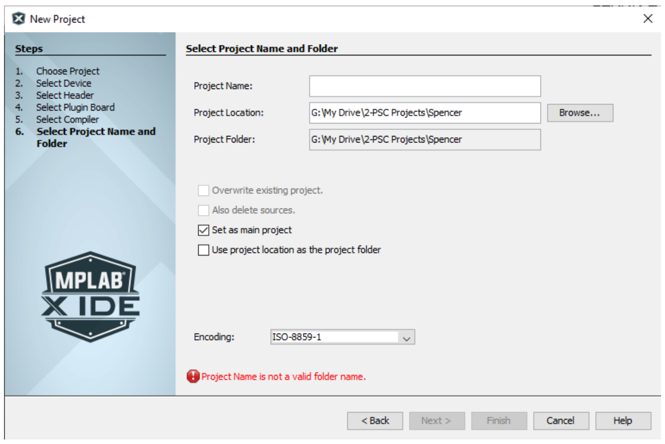
Select Compiler

```
Compiler Toolchains
⊟ XC16
    XC16 (v2.00) [C:\Program Files\Microchip\xc16\v2.00\bin]
  ASM30
  C24
  C30
  HI-TECH DSPICC
```

Please select a compiler

< Back | Next > | Finish | Cancel | Help

This is VERY IMPORTANT. If you do not see an XC16 compiler option then you must go and download one from online. This compiler is in charge of converting your code into instructions for the PIC24. A google search should provide you the options needed to get one. Select the most recent XC16 compiler and hit next. The final screen looks like the following:



New Project

Steps
1. Choose Project
2. Select Device
3. Select Header
4. Select Plugin Board
5. Select Compiler
6. **Select Project Name and Folder**

Select Project Name and Folder

Project Name:
Project Location: G:\My Drive\2-PSC Projects\Spencer | Browse...
Project Folder: G:\My Drive\2-PSC Projects\Spencer

☐ Overwrite existing project.
☐ Also delete sources.
☑ Set as main project
☐ Use project location as the project folder

Encoding: ISO-8859-1

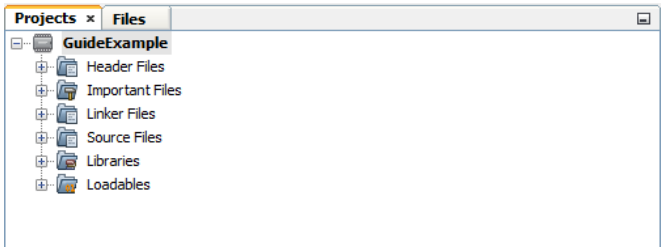
Project Name is not a valid folder name.

< Back | Next > | Finish | Cancel | Help

In this menu you choose where your project will be saved and what the name of the project will be. Do not use special characters or spaces in the name to avoid compiler issues. When done hit finish and your project should open.

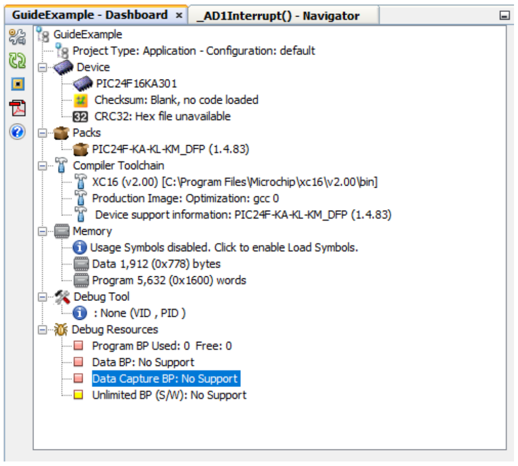
Navigating the IDE

When done the front screen will have some new side panels. The top left panel is the project view and shows you the files that make up your project. These will include source and header files. It looks like the following:



```
Projects ×   Files
⊟ GuideExample
   ⊞ Header Files
   ⊞ Important Files
   ⊞ Linker Files
   ⊞ Source Files
   ⊞ Libraries
   ⊞ Loadables
```

The bottom panel is a summary of project resources and settings. It looks like the following:



```
GuideExample - Dashboard ×   _AD1Interrupt() - Navigator
GuideExample
  Project Type: Application - Configuration: default
⊟ Device
    PIC24F16KA301
    Checksum: Blank, no code loaded
    CRC32: Hex file unavailable
⊟ Packs
    PIC24F-KA-KL-KM_DFP (1.4.83)
⊟ Compiler Toolchain
    XC16 (v2.00) [C:\Program Files\Microchip\xc16\v2.00\bin]
    Production Image: Optimization: gcc 0
    Device support information: PIC24F-KA-KL-KM_DFP (1.4.83)
⊟ Memory
    Usage Symbols disabled. Click to enable Load Symbols.
    Data 1,912 (0x778) bytes
    Program 5,632 (0x1600) words
⊟ Debug Tool
    : None (VID , PID )
⊟ Debug Resources
    Program BP Used: 0  Free: 0
    Data BP: No Support
    Data Capture BP: No Support
    Unlimited BP (S/W): No Support
```

The top tool bar contains the commands to compile, build, program, and debug your code with your PIC24. For more details on these features refer to the sections on the basics of a PIC24 program and the set up for the debugging tool.

How Programming a PIC24 Differs from Writing Code in C++

Variable Limitations

If you have had previous experience programming in C++ you are probably accustomed to the wide varieties of variables and STL containers that are available on PC devices. While most of these capabilities carry over to the PIC24 most of them are not inherent and must be included as header files. However, it is important to note that the PIC24 does not have much memory. This means that as a general rule of thumb dynamic memory allocation should be avoided as much as possible. Another consideration is that global variables should only be used if they are used frequently. A good rule of thumb for global variable use is that if it will be used in more than one function repeatedly then it is ok to have. While in most cases these will hardly be problems for robots and small projects, any attempt to be fancy should keep in mind these limitations.

Registers

Unlike most code in C++, programming a PIC24 requires you to access and write specific bytes of data to storage areas called registers. These registers and what each byte of data controls are all explained in the datasheet. When programming a register, you will have to use special calls to the specific register you want to change and then specific bits you want to write. Some registers have shortcuts to specific bits as well. Further on in this document we will give example code that will show you how to perform common tasks on the PIC24 with their associated registers.

No Delay or Wait Function

The PIC24 has no in-built wait function. Often times in coding in C++ or coding an Arduino we are used to being able to tell the device to wait before doing something else. This is still possible on the PIC24 but requires the programmer to make use of a timer and timer interrupt as well as a global variable which will get set to a specific value when to timer is done to let the code know to continue. See page 41 for more information on how to use timers and timer interrupts.

Interrupts

Putting a while loop in your main loop is a common way to loop through repeated actions and respond to inputs in normal code. This leads to problems when programming a microcontroller however because we are dealing with peripheral devices and sensors which interact with the world around them. A while loop would keep the code stuck until its conditions are met and this could lead to us not responding to potential hazards for our device or more important tasks that arise. While loops also do not have a sense of time and this can lead to problems when dealing with timers, PWM signals, or time sensitive inputs. The use of interrupts allows the PIC24 to respond in the moment to different inputs and cleans up the main loop significantly. Interrupts are code that stops the main loop temporarily and runs code in response to a specific event. This could be a timer reaching a certain time, a PWM signal counting a certain amount, a digital input going high, etc. These interrupts have to be enabled but make coding much more fluid and real time than just writing code. In a practical application this allows you to respond to sensors on your system quickly and correctly. It also helps to prevent code from interrupting each other as interrupts enter a queue with priorities.

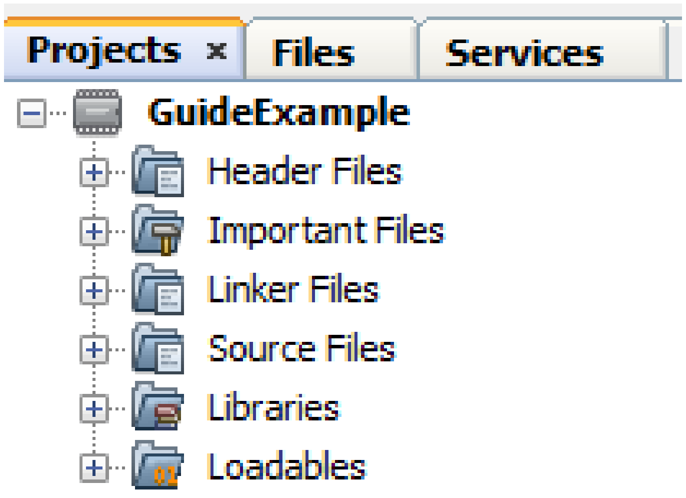
Voltages vs. Numbers

When using the PIC24 most of your sensors will not output a simple digital high or low. Often times you deal with voltage values that must be converted to digital numbers. We discuss this further on pages 24 and 31 but the complication of programming a PIC24 is that your inputs are no clean numbers like 2V or 1.25V but instead are converted into values like 900 and 2400.

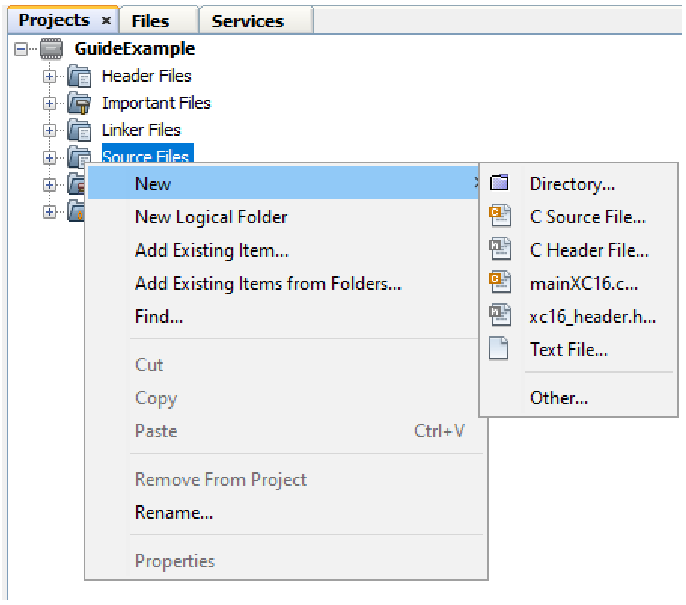
Basics of a PIC24 Program

Making a Main File

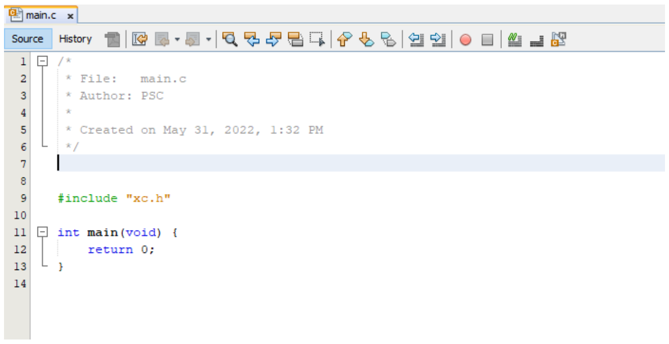
After loading your project your side panel should look like this:



Currently we have no files within our project. To remedy this, we need to make a main.c file. If we right click on the Source Files folder we get the following options:



Selecting the mainXC16.c option we can rename the file to just main.c and hit finish. From there we should have a file open that looks like the following image:



```c
/*
 * File:   main.c
 * Author: PSC
 *
 * Created on May 31, 2022, 1:32 PM
 */


#include "xc.h"

int main(void) {
    return 0;
}

```

It already has the xc.h header included so that we can program the registers on our PIC24.

Choosing an Oscillator Frequency

An important thing about programming a PIC24 is the frequency of your oscillator. The frequency of your oscillator divided by two is called your cycle frequency. Each iteration of the cycle frequency runs one command of code. Each line of code can translate to different amounts of commands but often it is about six commands per line. When programming timers or PWM signals this is much more important since you are limited in the PIC24's memory to how many cycles you can count and thus how long a timer or PWM period can be. The options you have are as follows:

FRC – This is the internal 8 MHz oscillator

FRCDIV – This refers to the internal 8 MHz oscillator with a postscaler divider. A postscaler divider means the frequency of the oscillator will be reduced.

LPRC – This is the 31.25 kHz Low-Power internal oscillator

LPFRC – This is the 500 kHz Low Power internal oscillator with a postscaler divider.

To set this up in the code simply place the following line before your main loop:

```
#pragma config FNOSC = /**/;
```

In place of the /**/ simply place one of the 4 options from above. If you chose any of the clock options with a postscaler divider you will need an additional line of code. While the clock can only be defined once out of the main loop. This line defines the divider and can be changed whenever in the code. This is useful if you need to change clock signals to temporarily be able to make a PWM signal work or get a timer to wait for a certain period of time. For the divider use the following line:

```
_RCDIV = 0bXXX;
```

Where the XXX is you must choose one of the following values when you are using the FRCDIV 8MHz oscillator:

111 = 31.25 kHz (divide-by-256)
110 = 125 kHz (divide-by-64)
101 = 250 kHz (divide-by-32)
100 = 500 kHz (divide-by-16)
011 = 1 MHz (divide-by-8)
010 = 2 MHz (divide-by-4)
001 = 4 MHz (divide-by-2) (default)
000 = 8 MHz (divide-by-1)

When you are using the LPFRC 500 kHz oscillator

111 = 1.95 kHz (divide-by-256)
110 = 7.81 kHz (divide-by-64)
101 = 15.62 kHz (divide-by-32)
100 = 31.25 kHz (divide-by-16)
011 = 62.5 kHz (divide-by-8)
010 = 125 kHz (divide-by-4)
001 = 250 kHz (divide-by-2) (default)
000 = 500 kHz (divide-by-1)

After setting the oscillator for your PIC24 your code is now ready to be written.

Writing in the Main Loop

The main loop will loop repeatedly after it finishes all commands. While at times this may be advantageous often we want it to run once and loop only certain parts. To prevent this from happening we just insert the following code into our main loop at the end before the return

```
while(1);
```

This is an infinite loop that has our main loop idle while we may have things going on with interrupts or other code. Now any code we insert before this command will run once and then the PIC24 will stay in that infinite loop until it times out after a few minutes on inaction or until we reset it.

General Good Practices

Global Variables Header

To keep your main.c file as clean and neat as possible it is a good idea to make a dedicated header file for your global variables. To do so, right click in your project window on the Header Files folder, select new, and then select C Header File. Name your file something obvious for another code reader and select finish. Now include the file in your main.c file with the following line of code:

```
#include “your_file_name.h”
```

In this file you can define all your global variables and initialize them. Include this file where you use these variables.

Interrupts Header

It is highly recommended you make a dedicated header file for the code needed to handle the interrupts when they occur. Make a header file like before but name it interrupts.h or ISRs.h. Include it in your main.c file and write out all the code to handle interrupts in this file. Use switch statements in your handlers so you can make cases for different functions or sections of code that use this interrupt and need to behave differently.

Using Header Files for Other Code

Often times you may write long functions or work on code as a group. In these times using header files to hold code that will be used in main.c file but would make it much harder to read is best. A good rule of thumb for functions is that if it will take 10 or more lines, place it in a separate header file. If you are having each member of your group write their own code make a header file for each person and have them write their functions in that header file. Make sure to include all your files in your main.c file.

Comment Everything

PIC24 code is hard to read. Making many comments to save your life and your project in the inevitable future. Having more comments is always better than having less.

Using Custom Definitions to Improve Code Legibility and Avoid Magic Numbers

PIC24 code is often nonintuitive to read. We are setting registers and variables to many different values and this can be hard to come back to and review when we need to fix code. To avoid this becoming an issue try using the following line of code to help out:

```
#define /*register*/ /*new name*/
```

This assigns the register an alias of your choice. As an example, if we had a photodiode on pin 3 we would have to call ADC1BUF1 to read its value. With a custom definition, anywhere we would type ADC1BUF1 to read the data of a photodiode on pin 3, we could replace with something like frontPhotodiode which is a lot easier to understand and diagnose. Make sure to place these definitions in your global variables header file for ease of reading and to keep your main.c file clean and clear. In addition to improving your code’s readability, custom definitions should be used to avoid magic numbers. Magic numbers are repeatedly used numbers that have a meaning to the programmer but would be hard or impossible to understand for another reader. As an example, let’s say you have a PWM signal whose frequency you change between 3 values in your code. Instead of just writing these numbers every time you could use a custom definition like the following:

```
#define fiftyHz 4999
```

So that anywhere you would place 4999 you can use fiftyHz instead. The compiler will automatically make the substitutions for you before downloading to the PIC24. This makes your code much easier to read and can reduce the number of variables you use if you are smart about your definitions.

Clear All Pins at the Start of your Code

The PIC24 requires all unused I/O pins to be set to digital outputs set to low when not in use. To ensure your code behaves how you expect it to insert the following lines of code at the beginning of your main loop.

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

This will ensure every pin is a digital output set to low until you reassign them later on in your code.

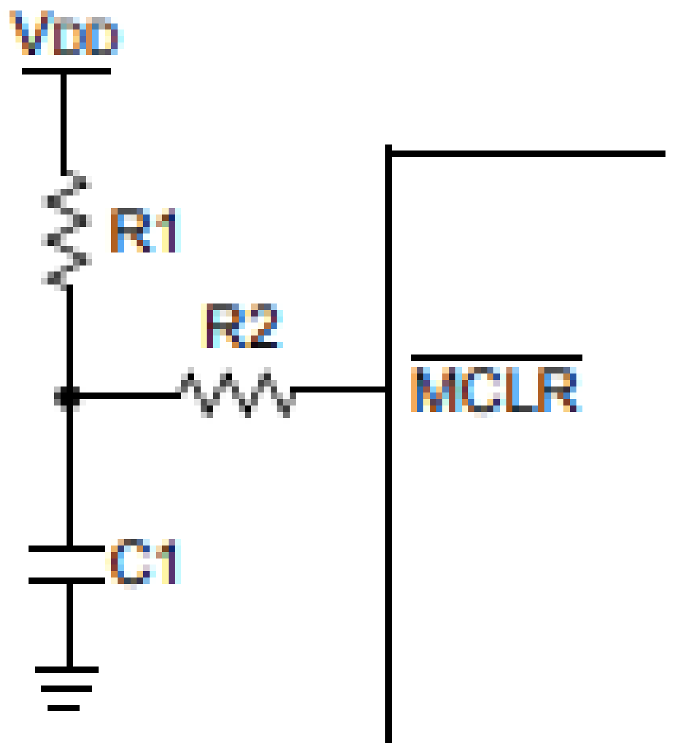
Create Code Templates

It is extremely helpful if you take the time to make code templates or template functions that you can use later. A good way to do this is to make a header file of functions that you can copy and use in different projects. Examples of functions that would be useful are a function to setup the analog to digital converter, a function to create a timer, a function to setup a PWM signal, etc. If you add plenty of comments to these functions it can save you a tremendous amount of time when you need to change or redo parts of your code. Just make sure you call those functions in your main loop so that the settings are actually made in your PIC24.

How to Setup Basic PIC24 Connections

MCLR Pin

The MCLR pin is the master clear pin for the PIC24. It must be held at a constant high voltage for it to be inactive and the device to run. This pin is pin 1 on the PIC24 and the required circuit is as shown below:



VDD needs to be 3.3V, R1 is a 10 kΩ resistor, R2 is a resistor between 100 Ω and 470 Ω, and C1 is a .1μF capacitor.

VSS and VDD Pins

Pin 19 is the VSS pin and pin 20 is the VDD pin. You will need to place a .1μF capacitor between the two pins. This helps to reduce any fluctuations in the power supply and ensure the PIC24 outputs a consistent power supply. The max current output on the VDD pin is 250 mA. This is shared with all the I/O ports which can only output 25mA each. Be careful not to draw too much current or you risk burning out your PIC24.

Programming Pins

Pins 9 and 10 are your programming pins. It is VERY IMPORTANT that you remove any sensors or other connections before connecting your PICKit 3 to prevent interference. After programming the PIC24 you can then use these pins freely. The programming pins can be changed if needed but before doing so, make sure to review the datasheet to see what pins have what capabilities. These programming pin options are as follows:

PGx3 = Pins 9 and 10 (default)
PGx2 = Pins 3 and 2
PGx1 = Pins 4 and 5

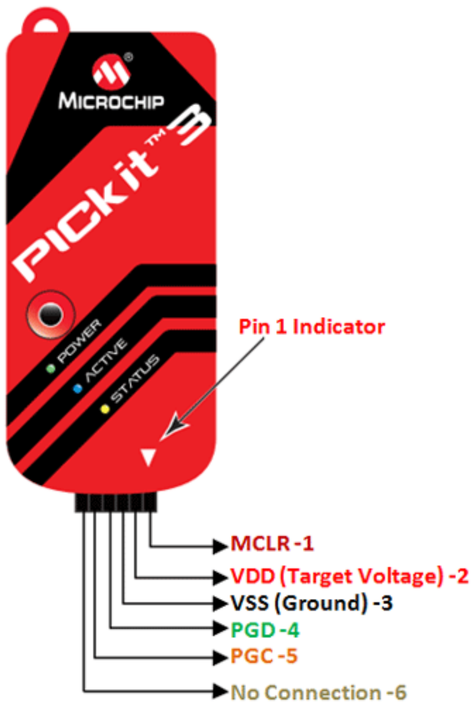
The options are listed in the order that the pins will be connected to the PICKit 3 programmer. The first pin listed is the PGD pin and the second pin is the PGC pin. On the PIC24 PCB available at the PSC pins 9 and 10 are preconnected to the PICKit 3 connection on board. If you intend on changing the pins you should not use the PICKit 3 connection on that PCB and instead use individual wires going to the proper connections from the PICKit 3. To change the programming pins, use the following optional line of code in your main.c file outside of the main loop:

```
#pragma config ICS = XXXX
```

Where XXXX is replaced by one of the options above.

Wiring the PICKit 3

To wire the PICKit 3 follow this diagram and the paragraph that follows:



MCLR connects to pin 1, VDD pin 20, VSS to pin 19, PGD to pin 9, and PGC to pin 10. If you changed the programming pins make sure you refer to the previous paragraph to ensure that your PGD and PGC pins are properly connected.

Powering the PIC24 from the PICKit 3

If you do not have a voltage regulator circuit or any other power source hooked up to your PIC24 you can use the PICKit 3 to provide the power you need to run your PIC24. To do this, got to file>>project properties and in the menu that pops up select the PICKit 3 on the side bar. Then at the top drop-down menu select Power and set the voltage level to 3.25V. Then check the box that says power target circuit from the PICKit 3. If you have a voltage regulator circuit attached you should turn this option off since the PICKit 3 will not be able to supply enough power for your circuit in that case. Use the voltage regulator circuit to get the external power down to 3.3V instead.

How to Configure and Use the Debugging Tool

Programming Pins

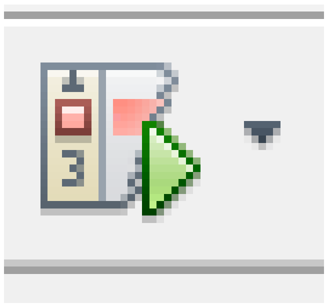
Refer to the section on these pins on page 10 to ensure your programming pins are properly selected. Make sure even if you don't change pins to include the following line of code outside your main loop:

#pragma config ICS = PGx3

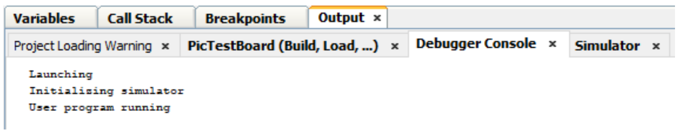
So that the default pins 9 and 10 are designated as the programming and debugging pins. After including that line of code and wiring the PICKit 3 to its proper connections you are now ready to use the debugging tool.

Programming the PIC24 for Debugging

At the top of the MPLAB X window you should see the following icon:



This icon programs your PIC24 and opens the Debugging tool. Upon programming your PIC24 you should see the following window open up at the bottom of your screen:

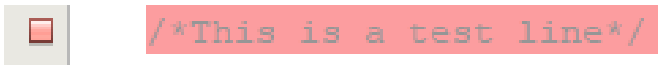


```
Launching
Initializing simulator
User program running
```

The output tab is your standard project warnings. The Breakpoints tab refers to points that you would like the code to pause at. We will talk about these further on. The Call Stack tab shows the current functions that have been called at this point in the code. The Variables tab contains information on both watches you set and local variables.

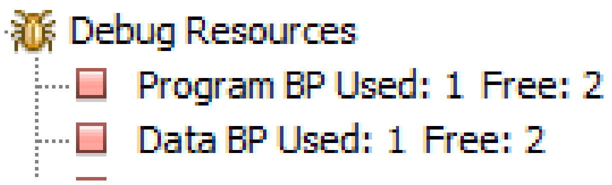
Setting and Using Breakpoints

Breakpoints are generally used when you want to verify that a value is met at a certain pint in the code, or to verify that a register is set properly when you arrive at a part of the code. To set a breakpoint. Find a line of code of code you would like to stop at. Now go to that line and click the number of that line. You should see the following:



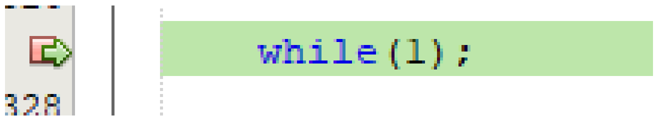
```
/*This is a test line*/
```

This is a visual indicator of a breakpoint. In the bottom left hand of your screen in the dashboard you should see the following as well:



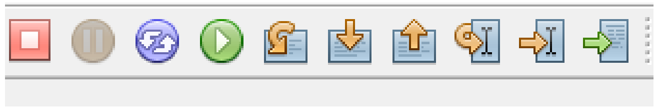
```
Debug Resources
    Program BP Used: 1  Free: 2
    Data BP Used: 1  Free: 2
```

This tells you how many breakpoints you have available for use with your debugging tool. The PICKit 3 supports three breakpoints. However, it is important to note that using all three will result in the inability to reset the PIC24 programmatically as well as other features. If you run your debugged code and reach your breakpoint you should see the two things occur. First the screen should show a green line like this:



```
while(1);
```

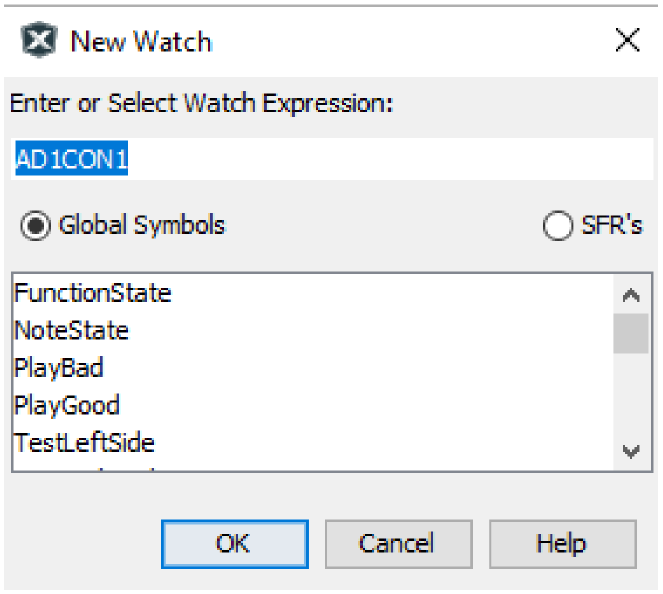
This is indicative of where the code is stopped at. The other thing you should see is this at the top:



This shows you the code is paused and you have some options of how to proceed. The green arrow means continue. The blue cycle means reset. And the rest stop means end debugging. The other arrows are used to in order: step over the current step, step into the code, step out of the code, run to cursor, set pc at cursor, and focus cursor at pc. These tools help you to navigate code more slowly. While paused you can view variables and the call stack and make sure everything is running the way it should

Setting and Using Variable Watches

Whenever you pause the code with the pause button at the top of the screen or you hit a breakpoint. The code will update the variable tab with a snapshot of any local variables at that point in the code and a snapshot of any watches you have set. Watches can be set to view global variables or registers in the PIC24. To set a watch go to the variables tab and click on the diamond with a green plus in the corner. You should get a window that looks like the following:



New Watch

Enter or Select Watch Expression:

AD1CON1

[✓ Global Symbols] SFR's

FunctionState
NoteState
PlayBad
PlayGood
TestLeftSide

In this window select global symbols for global variables or SFR's for registers. Search with the bar at the top or scroll till you find what you need and hit ok. Now you should see you watch in the variables tab. To delete it right click on it and select delete. To change what information you see, right click on the headers and select the information you're interested in and a column will appear. Now whenever you pause the code these will be updated.

Configuring Pins as Digital I/O Pins without Change Interrupts

PORTS

The PIC24 consists of two ports. Ports are groups of pins that are tied to a register for data. These ports on the PIC24 are ports A and B. Referring to the pin diagram on Page II of this document you can see which pins belong to what port and what pin number they are by looking for an RX## symbol on the diagram. The X represents what port the pin belongs to and the ## refers to what bit in the registers below the pin corresponds to.

ANSX Registers

The ANSX registers are in charge of telling the PIC24 if a pin is an analog or digital pin. Only 12 pins on the PIC24 are analog enabled pins and require this line of code. These are pins 2, 3, 4, 5, 6, 7, 8, 9, 15, 16, 17, and 18. To set them as digital pins you must use the following line of code for each pin you want to use:

```
_ANSX## = 0;
```

Where X refers to the port that the pin belongs to and ## refers to the corresponding bit in that port.

TRISX Registers

The TRISX registers are in charge of defining whether a pin a digital input or output pin. To set each pin you must use the following line of code for each pin you want to use:

```
_TRISX## = Y;
```

Where X refers to the port of the pin and the ## refers to the corresponding bit in that register. Replace the Y with a 0 for output and 1 for input.

LATX Registers

The LATX Registers control the output of the pins. Use the following command to set a digital output to high for each pin you want to set:

```
_LATX## = 1;
```

Where X is the pins port and ## is the bit in that port.

Reading from a Digital Pin

To read from a pin we need to read from the PORTX register. As an example, if we wanted to read a digital input from pin 6 which is RB2 and set a variable equal to its value I would use the following line of code:

```
int var = PORTBbits.RB2;
```

If I were calling a pin from port A I would have to make sure to change PORTBbits to PORTAbits and then change the ending to the corresponding pins.

Max Voltage Input on any Digital Pin

The max voltage on any digital pin is 3.6V. Exceeding this voltage can lead to potential failure of your PIC24. Verify inputs with a multimeter or voltage regulator before inputting them to the PIC24.

Max Current Out on any Digital Pin

The current output limit for the PIC24 is 25mA. Exceeding this current draw can lead to potential failure of your PIC24.

Code Example - Configuring Digital I/O Pins without Change Interrupts

```
/*
* File: main.c
* Author: Spencer Mosley
* Created on May 31, 2022
* Description: In this code I set pin 2 as a digital output and pin 16 as a digital input
* This document is provided as a reference material
*/
#include "xc.h"
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        /*pin 2 is RA0*/
        /*pin 16 is RB13*/
        _RCDIV = 0b100; // sets a 16 post scaler
        /*Reset All Registers We will be Using*/
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Set Pins 2 and 3 as Digital Outputs*/
        _ANSA0 = 0; //sets pin 2 as a digital pin
        _TRISA0 = 0; //sets pin 2 as digital output
        _LATA0 = 1; //sets pin 2 output to high
        /*Set Pin 16 as a Digital Input*/
        _ANSB13 = 0; //sets pin 16 as a digital pin
        _TRISB13 = 1; //sets pin 16 as a digital input
        /*Keep the Code Running as is*/
        while(1);
        return 0;
}
```

Configuring Pins as Digital I/O Pins with Change Interrupts

PORTS

The PIC24 consists of two ports. Ports are groups of pins that are tied to a register for data. These ports on the PIC24 are ports A and B. Referring to the pin diagram on Page II of this document you can see which pins belong to what port and what pin number they are by looking for an RX## symbol on the diagram. The X represents what port the pin belongs to and the ## refers to what bit in that register the pin corresponds to.

ANSX Registers

The ANSX registers are in charge of telling the PIC24 if a pin is an analog or digital pin. Only 12 pins on the PIC24 are analog enabled pins and require this line of code. These are pins 2, 3, 4, 5, 6, 7, 8, 9, 15, 16, 17, and 18. To set them as digital pins you must use the following line of code for each pin you want to use:

```
_ANSX## = 0;
```

Where X refers to the port that the pin belongs to and ## refers to the corresponding bit in that port.

TRISX Registers

The TRISX registers are in charge of defining whether a pin a digital input or output pin. To set each pin you must use the following line of code for each pin you want to use:

```
_TRISX## = Y;
```

Where X refers to the port the pin belongs to and the ## refers to the corresponding bit in that register. Replace the Y with a 0 for output and 1 for input.

LATX Registers

The LATX Registers control the output of the pins. Use the following command to set a digital output to high for each pin you want to set:

```
_LATX## = 1;
```

Where X is the pins port and ## is the bit in that port.

Reading from a Digital Pin

To read from a pin we need to read from the PORTX register. As an example, if we wanted to read a digital input from pin 6 which is RB2 and set a variable equal to its value I would use the following line of code:

```
int var = PORTBbits.RB2;
```

If I were calling a pin from port A I would have to make sure to change PORTBbits to PORTAbits and then change the ending to the corresponding pins.

Max Voltage Input on any Digital Pin

The max voltage on any digital pin is 3.6V. Exceeding this voltage can lead to potential failure of your PIC24. Verify inputs with a multimeter or voltage regulator before inputting them to the PIC24.

Max Current Out on any Digital Pin

The current output limit for the PIC24 is 25mA. Exceeding this current draw can lead to potential failure of your PIC24. This is also the max current out for a pin which has a pull up resistor enabled for its change interrupt

Change Interrupts

The PIC24 has the capability of detecting when a digital input pin changes state. This is great for responding to a button being pressed, a switch being thrown, or a digital sensor going high. Each of the digital I/O pins has a built-in pull up and pull-down resistor that can be enabled to eliminate the need for an external resistor. In your interrupt handler you will need to write code to handle changes from low to high AND from high to low. You can do this by checking what the value of the pin is, high or low, to respond accordingly. It is VERY IMPORTANT to note that the change interrupt is shared between all pins so you will need to check each pin in your handler to find which pin had changed

Enabling the Change Interrupt on Pins

Each I/O pin has its port number RBXX but it also has its own change interrupt number CNXX that corresponds to it. Look at the diagram on page II of this document to find the number for the pin you are configuring. After finding that number use the following line of code for each pin you want to enable the change interrupt on that pin:

_CNYYIE =1;

Where YY is replaced with the number of the pin you are configuring.

Optional: Pull-Up and Pull-Down Resistors

Each of the pins that has a change interrupt capability also has the ability to use an internal pull-up or pull-down resistor to eliminate the need for an external resistor. A pull-up resistor keeps the voltage of the pin high until the pin is connected to ground and a pull-down resistor keeps the pin low until it is connected to power. If your circuit calls for one of these you could remove it and enable the resistor with one of these lines of code:

_CNXPDE = 1;

or

_CNXPUE = 1

Where X is the CNX number of the pin you are configuring. It is important to note that only one of these may be enabled at a time so make sure you do not enable both to avoid errors or hurting the PIC24. Use these lines of code for each pin you are configuring.

Configuring the Change Interrupt

After defining which pins will use the change interrupt we need to configure how the change interrupt should work. Use the following lines of code to do this:

_CNIP = XX;
_CNIF = 0;
_CNIE = 1;

Where XX represents the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The second line clears the interrupt flag. When we handle the interrupt, we must always clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with how the pins change.

Handling the Change Interrupt

When the interrupt is called we need to make a handler that clears the interrupt flag and reacts to the interrupt. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__((interrupt, no_auto_psv))
_CNInterrupt(void){
        _CNIF = 0;
        /*your code here*/
}
```

The function name must be as given above for the code to work properly. The first line clears the interrupt flag and must always be included. The rest is up to you. You should check the values of pins here and respond according to your design.

Code Example - Configuring Digital I/O Pins with Change Interrupts

```
/*
* File: main.c
* Author: Spencer Mosley
* Created on May 31, 2022
* Description: In this code I set pins 2 and 4 as digital inputs and pins 3 and 5 as a
* digital inputs with change interrupts. I am only concerned in pins 3 and 5 go high*/
* This document is provided as a reference material
*/
#include "xc.h"
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
void __attribute__((interrupt, no_auto_psv)) _CNInterrupt(void){
        _CNIF = 0; //clears interrupt flag
        if(PORTAbits.RA1 == 1){
                /*react to pin 3 going high here*/
        }
        if(PORTBbits.RB1 == 1){
                /*react to pin 5 going high here*/
        }
}
main(void) {
        /*pin 2 is RA0*/
        /*pin 3 is RA1 and CN3*/
        /*pin 4 is RB0*/
        /*pin 5 is RB1 and CN5*/
        _RCDIV = 0b100; // sets a 16 post scaler
        /*Reset All Registers We will be Using*/
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Set Pins 2 and 4 as Digital Outputs*/
        _ANSA0 = 0; //sets pin 2 as a digital pin
        _ANSB0 = 0; //sets pin 4 as a digital pin
        _TRISA0 = 0; //sets pin 2 as digital output
        _TRISB0 = 0; //sets pin 4 as digital output
        _LATA0 = 1; //sets pin 2 output to high
        _LATB0 = 1; //sets pin 4 output to high
        /*Set Pins 3 and 5 as Digital Inputs*/
        _ANSA1 = 0; //sets pin 3 as a digital pin
        _ANSB1 = 0; //sets pin 5 as a digital pin
        _TRISA1 = 1; //sets pin 3 as a digital input
        _TRISB1 = 1; //sets pin 5 as a digital input
        /*Enable Change Interrupt on Pins 3 and 5*/
        _CN3IE = 1; //enables change interrupt for pin 3
        _CN5IE = 1; //enables change interrupt for pin 5
        /*Enable Internal Pull-Down Resistors*/
        _CN3PDE = 1;
        _CN5PDE = 1;
        /*Configure the Change Interrupt*/
        _CNIP = 4; //sets change interrupt priority to 4;
        _CNIF = 0; //clears the change interrupt flag
        _CNIE = 1; //enables the change interrupt
        /*Let the Interrupts Control the Rest of the Code*/
        while(1);
        return 0;
}
```

About the Analog to Digital Converter with Threshold Detect

Registers

There is a total of 11 registers that are used to configure the analog to digital converter. 2 of these are not used on the PIC24 and 2 of them are only used if you intend to measure capacitance on the analog to digital converter. The other 7 are used to configure the analog to digital converter, the pins to check, and the threshold detection feature.

Max Voltage Input on the Pins

The maximum voltage you can input on any analog pin is 3.6V. Exceeding this can potentially cause PIC24 failure so take care to check all signals you intend to read with a multimeter or oscilloscope before inputting them into the PIC24.

Resolution

When using the analog to digital converter it is important to remember how precise your measurements can be. The PIC24 has two operating modes for the analog to digital converter. These are the 10-bit and 12-bit operating modes. Assuming you will be using the PIC24 operating 3.3V as the voltage reference these are the smallest measurements you can get in each operating mode

10-bits:

$$\frac{3.3}{2^{10}-1} = .0032\text{V}$$

12-bits:

$$\frac{3.3}{2^{12}-1} = .00081\text{V}$$

You can also use a reference voltage with pin 2 being the positive input and pin 3 being the negative input. The reference voltage maximum is 3.3V and its minimum is 1.7V. If you use the smallest voltage reference, the smallest voltage measurement you could have would be the following:

10-bits:

$$\frac{1.7}{2^{10}-1} = .0017\text{V}$$

12-bits:

$$\frac{1.7}{2^{12}-1} = .00042\text{V}$$

However, if you do this, anything above 1.7V will be read as 1.7V. The analog to digital converter also does not output the voltage as a decimal value. Instead it outputs it as a multiple of the values given above. In the 10-bit operating mode your voltage will be represented as a value between 0 and 1023. In the 12-bit operating mode you value will be represented as a value between 0 and 4095. To calculate what the value you should expect to receive, given you know what voltage should be inputted, use the following equation:

$$\frac{\text{Your Voltage}}{3.3\text{V}} \times (1023 \text{ or } 4095)$$

The value you multiply by depends on your bit operation.

Analog to Digital Converter Clock Cycle

The analog to digital converter has its own internal counter that is responsible for creating a new frequency for the analog to digital converter. This frequency must have a minimum period of 600nS however for the sampling time a minimum of 750nSis required. The largest period that the analog to digital converter can have is 64 times the oscillator period. Alternatively, you can use one of the 5 timers on the PIC24 to act as the timing for the analog to digital converter. This will have to be done if you plan on using the threshold detect feature.

Analog to Digital Converter Interrupt Rate

The analog to digital converter throws an interrupt after a set number of samples have been taken. You will need to make sure that this interrupt occurs after you have sampled all of your pins. If it does not you will not see what happens on certain pins because the scan will reset after each interrupt.

Threshold Detect

The analog to digital converter has a built-in feature for detecting when a pin rises above a certain value, falls below a certain value, is within a certain window of values, or is outside a certain window of values. This feature can throw an interrupt if desired so that you can address these changes or can make it so that you don’t need to store the value the sensor is at or run any comparison logic on your pins and rather just look at what pins tripped.

Configuring the Analog to Digital Converter without Threshold Detect

Clear All Converter Registers

To start you need to clear the registers so you can have proper settings for your analog to digital converter. To do this use the following lines of code to clear the relevant registers:

```
AD1CON1 = 0;
AD1CON2 = 0;
AD1CON3 = 0;
AD1CON5 = 0;
AD1CSSL = 0;
AD1CSSH = 0;
```

Setting the Pins for Analog Use

We need to tell the pins we are going to use that they are analog inputs. To do this we use the following lines of code for each pin we want to use:

```
_ANSYXX = 1;
_TRISYXX = 1;
```

Where Y is the letter corresponding to the port of the pin we use. The XX refers to the bit in that port. Look at page II of this document for the pin out diagram and find the RYXX value of the pin you want to use for that information.

Setting Voltage and Ground References

The analog to digital converter needs to know with regards to what voltage and ground it should base its conversion on. Refer to the Resolution section in About the Analog to Digital Converter with Threshold Detect to understand what your options are. To set the positive voltage choose from the following options and use the following line of code:

11 = 4 * Internal VBG
10 = 2 * Internal VBG
01 = External VREF+
00 = AVDD

```
_PVCFG = 0bXX
```

Where the XX is exchanged for the option you choose above. The VBG can be ignored since it falls outside the scope of the projects you will do. To set the ground reference choose from the following options and use the following line of code:

1 = External VREF-
0 = AVSS

```
_NVCFG = X;
```

Where X is exchanged for the option you chose above.

Set the Analog to Digital Conversion Timing

When not using the threshold detect you can have the analog to digital converter take care of its own timing. Simply use the following line of code:

```
_SSRC = 0b0111;
```

This means the converter automatically converts after finishing sampling.

Choose Bit Mode

You need to decide if you would like to use the 10-bit or 12-bit resolution mode. Refer to the Resolution section in About the Analog to Digital Converter with Threshold Detect to understand what your options are. Use the following line of code:

_MODE12 = X;

Where X is 0 for 10-bit mode or 1 for 12-bit mode.

Choose Output Format

The PIC24 has 4 output formatting options you can choose from. They are as follows:

11 = Fractional result, signed, left-justified
10 = Absolute fractional result, unsigned, left-justified
01 = Decimal result, signed, right-justified
00 = Absolute decimal result, unsigned, right-justified

If you use the signed fractional result you will get a value between -0.500 and .500. The inverse of your reference voltage is equal to -.500 and your reference voltage is .500. To find what voltage you are at you would need to multiply that number by your reference voltage. An unsigned fractional result will be represented as a value between 0 and 1. This is in absolute form so a negative voltage reading would still be read as simply its value. To find what voltage you are at take the value and multiply that by your reference voltage. A signed decimal result would give you a value between -2048 and 2047 for 12-bit mode or -512 and 511 in 10-bit mode. To find what voltage you are at take the value and divide it by 2048 for 12-bit mode and 512 for 10-bit mode. Then multiply that value by your reference voltage. An unsigned decimal result would be a value between 0 and 4095 for 12-bit mode and 0 and 1023 for 10-bit mode. Take the value and divide it by 4095 in 12-bit mode or 1023 in 10-bit mode and multiply the result by your reference voltage. It is highly unlikely that you will encounter negative voltages in your readings and since your PIC24 does not care for the difference between and integer and a float it is recommended to use the unsigned integer results for ease of programming. Some math will be required on you part to understand what you are looking for or what the output means but it will be easier to deal with in the future rather than fractions. To set this use the following line of code:

_FORM = 0bXX:

Where XX is replaced by the value corresponding to the option you decide to use. Remember that if you choose a fractional form any attempt to store in a variable must use a float variable.

Auto Sampling

Auto sampling means after the converter is done converting the last value it read, it will begin reading another value. If you are sampling from multiple pins you should use this option. If you are only reading from one analog pin you should still set this. If you want to sample on demand read the data sheet for more information on how that is done. Use the following line of code to set it:

_ASAM = X;

Where X = 1 for enabled and 0 for disabled;

Setting Output Location

There are options for where to output your converted data but it is HIGHLY recommended you set the following option to enabled. The reason for this is that it prevents data from being overwritten and simplifies accessing your data. If you desire to do otherwise this is at your own discretion and will require some through reading of the data sheet starting at page 210. Use the following line of code:

_BUFREGEN = 1;

Change the value to 0 to disable this at your own risk. The following only applies to if this option is enabled. To read the value you must access it with this value:

ADC1BUFXX

Where XX represents the ANXX value that corresponds to the pin you are reading from. For example, if you are reading from pin 2 it corresponds to AN0, see page II of this document for a diagram of the pinout, and I would access its converted value with ADC1BUF0. I could set a variable equal to that value or you can compare that value in a comparison statement.

Auto Scan Inputs

In most cases you should have the PIC24 iterate through the analog pins. If you do not iterate through the pins you will be in charge of telling the PIC24 which pin it needs to look at for each sample iteration. Look at the datasheet for more information on how to do that. To turn on the auto scanning use the following line of code:

```
_CSCNA = 1;
```

Change the 1 to 0 to disable auto scanning. If you are auto scanning it is VERY IMPORTANT that you follow the next step to tell the PIC24 what pins should be included in the auto scan.

Defining Which Pins to Auto Scan

The PIC24 does not look at the ANSX registers to determine which pins it should scan when auto scanning. Instead it looks at the ADC1CSSL register. Having reset the register, it will not scan any pin currently. To tell it which pins to scan you must use the following line of code for each pin you want to scan:

```
_CSSXX = 1;
```

Where XX represents the ANXX number of the pin. Look at the pinout on page II of this document next to the pin you intend to use for its ANXX number.

Defining the Interrupt Rate

The analog to digital converter throws an interrupt after it has converted a certain number of samples. With the auto scan enabled it is VERY IMPORTANT that you properly change this setting. Even without the interrupt enabled this must be set. Take the number of pins you are scanning, subtract one, and convert that number to binary. That is the input for the interrupt rate you will want to use. Set the interrupt rate with the following line of code:

```
_SMPI = 0bXXXXX
```

Where XXXXX is the number you calculated. You need to add the leading zeros so that it is five digits long. For example, if I was converting eight pins, I would need to use binary seven which is 111. I would then add the two leading zeros to get 00111 which is what I would use as my setting.

Defining the Analog to Digital Converter Clock Cycle

You must always set the conversion clock setting. To do this you must choose from the following options and use the following line of code:

11111111-01000000 = Reserved
00111111 = 64·TCY = TAD
•
00000001 = 2·TCY = TAD
00000000 = TCY = TAD

```
_ADCS = 0bXXXXXXXX;
```

Where TCY is your chosen Oscillator period, or one divided by your Oscillator Frequency, and

TAD is your converter period. This period must be longer than 600nS for converting. Replace XXXXXXXX with your chosen option.

Defining the Auto Sample Time

After setting the converter clock you need to set how long the converter should sample the pin value for. This must be longer than 750nS. To set it choose one of the following options and use the following line of code:

11111 = 31 TAD
•
•
•
00001 = 1 TAD
00000 = 0 TAD

```
_SAMC = 0bXXXXX;
```

Where XXXXX is replaced by the option you chose and TAD is the converter clock cycle.

Configuring the Analog to Digital Converter Interrupt

This step is optional but the interrupt rate must still be set even if this is not used. After sampling all your pins, you can throw an interrupt to pause your main loop and run specific code in response to that data. To do this we need the following three lines of code:

```
_AD1IP = X;
_AD1IF = 0;
_AD1IE = 1;
```

Where X represents the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _ADC1IF command refers to the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with responding to the data acquisition or thresholds.

Handling the Analog to Digital Converter Interrupt

If we have enabled the interrupt we must handle that interrupt. To do so we must create a handler in our code. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_ADC1Interrupt(void){
        _AD1IF = 0;
        /*your code here*/
}
```

The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Enabling the Analog to Digital Converter

The last thing to do is enable the analog to digital converter. It can be turned on and off as needed and is all done with the following line of code:

```
_ADON = X;
```

Where X is 1 for on and 0 for off.

Code Example – Configuring the Analog to Digital Converter without Threshold Detect

```
/*
* File: main.c
* Author: Spencer Mosley
* Created on June 2, 2022
* Description: In this code I set pins 17 and 18 as analog inputs. I also configure the A/D
* converter without threshold detect and an interrupt for when the values are available.
* This document is provided as a reference material
*/
#include "xc.h"
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void){
_AD1IF = 0; //clears interrupt flag
        /*code would go here based on what I was doing*/
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
/*pin 17 is RB14 and AN10*/
/*pin 18 is RB15 and AN9*/
_RCDIV = 0b100; // sets a 16 post scaler
/*Reset All Registers We will be Using*/
AD1CON1 = 0;
AD1CON2 = 0;
AD1CON3 = 0;
AD1CON5 = 0;
AD1CSSL = 0;
AD1CSSH = 0;
LATA = 0;
LATB = 0;
TRISA = 0;
TRISB = 0;
ANSA = 0;
ANSB = 0;
/*Set Pins we want to use as Analog Pins and Inputs*/
_ANSB14 = 1; //sets pin 17 as an analog pin
_ANSB15 = 1; //sets pin 18 as an analog pin
_TRISB14 = 1; //sets pin 17 as an input
_TRISB15 = 1; //sets pin 18 as an input
/*Set Positive and Negative Voltage References*/
_PVCFG = 0b00; //selects voltage reference
_NVCFG = 0; //selects ground reference
/*Set Conversion Timing*/
_SSRC = 0b0111; //sets the converter as managing its own time
/*Set Bit Mode*/
_MODE12 = 1; //sets converter to the 12-bit operating mode
/*Set Output Format*/
_FORM = 0b00; //sets converter to absolute decimal format
/*Set Auto Sampling*/
_ASAM = 1; //enables auto sampling
/*Set Output Location*/
_BUFREGEN = 1; //sets output to buffers corresponding to pins
/*Set Auto Scanning of Inputs*/
_CSCNA = 1; //auto scanning enabled
/*Define which Pins to Scan*/
_CSS10 = 1; //enables scanning of pin 17
_CSS9 = 1; //enables scanning of pin 18
/*Choose Interrupt Rate*/
//(2 – 1) in binary is 00001
_SMPI = 00001; // interrupts every other sample
/*Define the Analog to Digital Converter Clock Cycle*/
_ADCS = 0b00000000; //sets clock to be the same as the system clock
/*Define the Auto Sample Time*?
_SAMC = 0b00001; //sets the sample time to be one converter clock cycle
/*Configure the Analog to Digital Converter Interrupt*/
_AD1IP = 4; //sets interrupt priority to 4
_AD1IF = 0; //clears interrupt flag
_AD1IE = 1; //enables interrupt
/*Turn on Analog to Digital Converter*/
_ADON = 1; //turns analog to digital converter on
/*Let the Interrupts Control the Rest of the Code*/
while(1);
return 0;
}
```

Configuring the Analog to Digital Converter with Threshold Detect

Setting the Pins for Analog Use

We need to tell the pins we are going to use that they are analog inputs. To do this we use the following lines of code for each pin we want to use:

```
_ANSYXX = 1;
_TRISYXX = 1;
```

Where Y is the letter corresponding to the port of the pin we use. The XX refers to the bit in that port. Look at page II of this document for the pin out diagram and find the RYXX value of the pin you want to use for that information.

Clear All Converter Registers

To start you need to clear the registers so you can have proper settings for your analog to digital converter. To do this use the following lines of code to clear the relevant registers:

```
AD1CON1 = 0;
AD1CON2 = 0;
AD1CON3 = 0;
AD1CON5 = 0;
AD1CSSL = 0;
AD1CSSH = 0;
```

Setting Voltage and Ground References

The analog to digital converter needs to know with regards to what voltage and ground it should base its conversion on. Refer to the Resolution section in About the Analog to Digital Converter with Threshold Detect to understand what your options are. To set the positive voltage choose from the following options and use the following line of code:

11 = 4 * Internal VBG
10 = 2 * Internal VBG
01 = External VREF+
00 = AVDD

```
_PVCFG = 0bXX
```

Where the XX is exchanged for the option you choose above. The VBG can be ignored since it falls outside the scope of the projects you will do. To set the ground reference choose from the following options and use the following line of code:

1 = External VREF-
0 = AVSS

```
_NVCFG = X;
```

Where X is exchanged for the option you chose above.

Choose Bit Mode

You need to decide if you would like to use the 10-bit or 12-bit resolution mode. Refer to the Resolution section in About the Analog to Digital Converter with Threshold Detect to understand what your options are. Use the following line of code:

```
_MODE12 = X;
```

Where X is 0 for 10-bit mode or 1 for 12-bit mode.

Choose Output Format

The PIC24 has 4 output formatting options you can choose from. They are as follows:

11 = Fractional result, signed, left-justified

10 = Absolute fractional result, unsigned, left-justified
01 = Decimal result, signed, right-justified
00 = Absolute decimal result, unsigned, right-justified

If you use the signed fractional result you will get a value between -0.500 and .500. The inverse of your reference voltage is equal to -.500 and your reference voltage is .500. To find what voltage you are at you would need to multiply that number by your reference voltage. An unsigned fractional result will be represented as a value between 0 and 1. This is in absolute form so a negative voltage reading would still be read as simply its value. To find what voltage you are at take the value and multiply that by your reference voltage. A signed decimal result would give you a value between -2048 and 2047 for 12-bit mode or -512 and 511 in 10-bit mode. To find what voltage you are at take the value and divide it by 2048 for 12-bit mode and 512 for 10-bit mode. Then multiply that value by your reference voltage. An unsigned decimal result would be a value between 0 and 4095 for 12-bit mode and 0 and 1023 for 10-bit mode. Take the value and divide it by 4095 in 12-bit mode or 1023 in 10-bit mode and multiply the result by your reference voltage. It is highly unlikely that you will encounter negative voltages in your readings and since your PIC24 does not care for the difference between and integer and a float it is recommended to use the unsigned integer results for ease of programming. Some math will be required on you part to understand what you are looking for or what the output means but it will be easier to deal with in the future rather than fractions. To set this use the following line of code:

```
_FORM = 0bXX:
```

Where XX is replaced by the value corresponding to the option you decide to use. Remember that if you choose a fractional form any attempt to store in a variable must use a float variable.

Auto Sampling

Auto sampling means after the converter is done converting the last value it read, it will begin reading another value. If you are sampling from multiple pins you should use this option. If you are only reading from one analog pin you should still set this. If you want to sample on demand read the data sheet for more information on how that is done. Use the following line of code to set it:

```
_ASAM = X;
```

Where X = 1 for enabled and 0 for disabled;

Auto Scan Inputs

In most cases you should have the PIC24 iterate through the analog pins. If you do not iterate through the pins you will be in charge of telling the PIC24 which pin it needs to look at for each sample iteration. Look at the datasheet for more information on how to do that. To turn on the auto scanning use the following line of code:

```
_CSCNA = 1;
```

Change the 1 to 0 to disable auto scanning. If you are auto scanning it is VERY IMPORTANT that you follow the next step to tell the PIC24 what pins should be included in the auto scan.

Defining Which Pins to Auto Scan

The PIC24 does not look at the ANSX registers to determine which pins it should scan when auto scanning. Instead it looks at the ADC1CSSL register. Having reset the register, it will not scan any pin currently. To tell it which pins to scan you must use the following line of code for each pin you want to scan:

_CSSXX = 1;

Where XX represents the ANXX number of the pin. Look at the pinout on page II of this document next to the pin you intend to use for its ANXX number.

Defining the Interrupt Rate

The analog to digital converter throws an interrupt after it has converted a certain number of samples. With the auto scan enabled it is VERY IMPORTANT that you properly change this setting. Even without the interrupt enabled this must be set. Take the number of pins you are scanning, subtract one, and convert that number to binary. That is the input for the interrupt rate you will want to use. Set the interrupt rate with the following line of code:

_SMPI = 0bXXXXX

Where XXXXX is the number you calculated. You need to add the leading zeros so that it is five digits long. For example, if I was converting eight pins, I would need to use binary seven which is 111. I would then add the two leading zeros to get 00111 which is what I would use as my setting.

Defining the Analog to Digital Converter Clock Cycle

You must always set the conversion clock setting. To do this you must choose from the following options and use the following line of code:

11111111-01000000 = Reserved
00111111 = 64 ·TCY = TAD
•
00000001 = 2 ·TCY = TAD
00000000 = TCY = TAD

_ADCS = 0bXXXXXXXX;

Where TCY is your chosen Oscillator period, or one divided by your Oscillator Frequency, and TAD is your converter period. This period must be longer than 600nS for converting. Replace XXXXXXXX with your chosen option.

Defining the Auto Sample Time

Since we are using threshold detect we cannot auto sample so this setting can be ignored.

Set the Analog to Digital Conversion Timing

Since we are using threshold detect we will need to use a timer to control when the sampling should end and when the conversion should begin. To do this select a timer from one of the following options:

0101 = Timer 1 event ends sampling and starts conversion
0011 = Timer5 event ends sampling and starts conversion
0010 = Timer3 event ends sampling and starts conversion

All the timers on the PIC24 are 16 bits but timers 2 and 3 are linked as well as 4 and 5. These links make it so you can use timers 2 and 4 as 32-bit timers. However, if you intend on using timers 3 or 5 for this timing you must turn off this link so that you can access timers 3 or 5. For this to work properly you need to do some math on what your timer period will be. The timer period must encompass our conversion time which is TAD and our sampling time as well as the time needed to start conversion. The data sheet states that there is a delay of 3 TAD periods after conversion ends for sampling to begin. Therefore, our minimum timer period must be

4 TAD periods with an additional 750nS for sampling. To setup the timer, look at our instructions on page 41 to properly initialize and set the timer. Some trial and error will be needed here to find a timer period that works well with your setup. Use the following line of code to then enable this setting:

```
_SSRC = 0bXXXX;
```

Where XXXX is replaced by the option that you chose from the list above

Enabling Threshold Detect

To use the threshold detect feature we need to enable it with the following line of code:

```
_ASEN = 1;
```

Defining Threshold Detect Interrupt Operation

The threshold detect feature shares the same interrupt with the normal analog to digital converter interrupt. This setting will modify how the interrupt occurs and requires a more involved version of handling. In our handling the interrupt section later on we discuss how to recognize in the interrupt handler if a threshold detect feature occurred or if it was a standard conversion interrupt. The options for the threshold detect are as follows:

11 = Interrupt after a Threshold Detect sequence completed and a valid compare has occurred
10 = Interrupt after a valid compare has occurred
01 = Interrupt after a Threshold Detect sequence completed
00 = No interrupt

The interrupt after a threshold detect sequence completes and a valid compare completes changes the interrupt to where it will not occur at all until it has scanned all the pins and one of those pins met our threshold. The interrupt after a valid compare changes the interrupt to only occur after a pin has met our threshold. The interrupt after a threshold detect sequence completes means will behave like normal, interrupting after a certain amount of conversions but it will also interrupt immediately if it detects that our threshold is met. This is the feature you will use 99% of the time. The last feature disables the interrupt for the threshold detect and it behaves like a normal analog to digital converter with the added feature of the PIC24 keeping track of what pins met a predefined threshold. After choose your desired operation use the following line of code to properly enable this setting:

```
_ASINT = 0bXX;
```

Where XX is replaced by the code for the operation you would like to use.

Setting Output Location

There are options for where to output your converted data but it is HIGHLY recommended you set the following option to enabled. The reason for this is that it prevents data from being overwritten and simplifies accessing your data. When using threshold detect this feature must almost always be enabled to prevent your thresholds from being overwritten accidentally. Use the following line of code:

```
_BUFREGEN = 1;
```

Change the value to 0 to disable this at your own risk. If you desire to do otherwise this is at your own discretion and will require some through reading of the data sheet starting at page 210. The following only applies to if this option is enabled. To read the value you must access it with this call:

```
ADC1BUFXX
```

Where XX represents the ANXX value that corresponds to the pin you are reading from. For example, if you are reading from pin 2 it corresponds to AN0, see page II of this document for a diagram of the pinout, and I would access its converted value with ADC1BUF0. I could set a variable equal to that value or you can compare that value in a comparison statement.

Defining Our Threshold Type

To use threshold detect we need to define what kind of threshold we would like to use. The options are as follows:

11 = Outside Window mode (valid match occurs if the conversion result is outside of the window defined by the corresponding buffer pair)
10 = Inside Window mode (valid match occurs if the conversion result is inside the window defined by the corresponding buffer pair)
01 = Greater Than mode (valid match occurs if the result is greater than the value in the corresponding buffer register)
00 = Less Than mode (valid match occurs if the result is less than the value in the corresponding buffer register)

If you choose either of the window modes you are limited to 5 pins which correspond to pins AN0, AN1, AN2, AN3, and AN4. These are pins 2, 3, 4, 5, and 6. This is because they use the buffers of other pins to write their upper window limit. The pins and buffer pairs are as shown below:

ADC1BUF0(pin 2) + ADC1BUF9(pin 18)
ADC1BUF1(pin 3) + ADC1BUF10(pin 17)
ADC1BUF2(pin 4) + ADC1BUF11(pin 16)
ADC1BUF3(pin 5) + ADC1BUF12(pin 15)
ADC1BUF4(pin 6) + ADC1BUF13(pin 7)

Where the first pin is the pin you would measure on and the second pin would be the one that is disabled. Any of the other options writes their threshold only to the buffer of the pin it uses. You can only have one type of threshold for all your pins defined at one time. With some creative programming loops and control it is possible to use different threshold types for different pins but this requires some more advanced thought than this guide will mention. After you have selected an option from the list above use the following lines of code:

```
_CM1 = X;
_CM0 = Y;
```

Where X is the first digit of the code you selected and Y is the second digit.

Setting Our Thresholds

We can only set the thresholds when the analog to digital converter is off. If we want to change them we must turn of the converter then turn it back on. To set a threshold for a pin we need to write the threshold to the output register of that pin. If we are using any of the windowed thresholds we write the lower limit to the threshold of the pin we are reading from and we write the upper limit to a corresponding register of another pin. A list of the paired pins are as follows:

ADC1BUF0(pin 2) + ADC1BUF9(pin 18)
ADC1BUF1(pin 3) + ADC1BUF10(pin 17)
ADC1BUF2(pin 4) + ADC1BUF11(pin 16)
ADC1BUF3(pin 5) + ADC1BUF12(pin 15)
ADC1BUF4(pin 6) + ADC1BUF13(pin 7)

Where the first pin is the one that is read and the other pin is the one that does not function during this operation. Make sure you do not scan the pins that are used to set the upper limit since they have no corresponding upper pin. For any other threshold mode any pin can be used by setting its threshold to its register.

To set the threshold use the following line of code:

```
ADC1BUFXX = YYYY;
```

Where XX is the corresponding analog register number given by the ANXX of the pin we want to use and YYYY is the value of the threshold. To calculate what the threshold should be use the following equation:

$$\frac{V_{threshold}}{V_{reference}} \times \text{Max Value}$$

Where the max value is 4095 for 12-bit operation and 1023 for 10-bit operation.

Determining What to Do with the Converted Voltage

When we use the threshold detect feature we need to decide what to do with the converted voltage value. The options we have are as follows:

10 = Auto-compare only (conversion results not saved, but interrupts are generated when a valid match as defined by CM and ASINT bits occurs)
01 = Convert and save (conversion results saved to locations as determined by register bits when a match as defined by CM bits occurs)
00 = Legacy operation (conversion data saved to location determined by buffer register bits)

The first option does not save the voltage value and is only concerned if our threshold is met. The second option saves our data according to our previous output settings, only when the threshold value is met. The last option saves the data according to our previous output settings after every conversion. If you used the recommended output setting the last two options are problematic however because our threshold value gets overwritten. If you didn't use the recommended output setting it's even more problematic because you won't know which output buffer was written to and therefore which threshold may have been overwritten. If you use one of these options you should disable the converter and timer in your interrupt handler, process the value, reset the threshold, and reenable your converter and timer when you're done. It is recommended you use the first option for simplicity sake since there is a dedicated register that will tell you which pin met its threshold so you don't have to worry about its value. After choosing a setting use the following lines of code:

```
_WM1 = X;
_WM0 = Y;
```

Where X is the first digit of your option and Y is the second digit of your option.

Configuring the Analog to Digital Converter Interrupt

This step is optional but the interrupt rate must still be set even if this is not used. After sampling all your pins, you can throw an interrupt to pause your main loop and run specific code in response to that data. To do this we need the following three lines of code:

```
_AD1IP = X;
_AD1IF = 0;
_AD1IE = 1;
```

Where X represents the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _ADC1IF command refers to the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with responding to the data acquisition.

Handling the Analog to Digital Converter Interrupt

THIS IS VERY IMPORTANT WITH THRESHOLD DETECT. If we have enabled the interrupt we must handle that interrupt. With threshold detect it is HIGHLY recommended that you have the interrupt enabled. Also, when using threshold detect this operation changes based on the settings we changed in the Defining Threshold Detect Interrupt section. If we used either of the first two options an interrupt will only occur after a value that meets a threshold is detected. The third option will have interrupts when after it scans all the pins or when a threshold is met. It is important to note with the third option interrupts can be either a threshold was met or it scanned all the pins. In the next section we address how to distinguish which pins met their thresholds. We must create a handler for the interrupt in our code. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_ADC1Interrupt(void){
        _AD1IF = 0;
        /*your code here*/
}
```

The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Identifying Which Pins Have Met their Thresholds

In the code to handle your interrupt you need to identify which pins may have met their threshold. If a pin meets a threshold the AD1CHITL register bit corresponding to the pin is thrown high. This must be cleared by the user so it makes the most sense in your interrupt handler to check which pins are high, clear them, act on the information, repeat. To check and write to the corresponding bit use the following call:

```
_CHHXX
```

Where XX is replaced by the corresponding XX value for the ANXX value of the pin we are using. If set that bit equal to zero then we have cleared it. If we check that bit and see it is equal to one then we know that the pin met our threshold requirements. It is important to note however that for the Outside window mode operation this bit is written as one is the data was written to the ADC1BUFXX register or if a match occurred. If you have turned of data saving it will only occur when a match has happened. See the code example below to understand what I would do in the handler.

Enabling the Analog to Digital Converter

The last thing to do is enable the analog to digital converter with the following line of code:

```
_ADON = X;
```

Where X is 1 for on and 0 for off. You can turn it on and off freely so you are not always converting if not needed.

Code Example – Configuring the Analog to Digital Converter with Threshold Detect

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 7, 2022, 1:32 PM
* Description: In this code I set pins 17 and 18 as analog inputs and configure the A/D
* converter with threshold detect to know when they become greater than 1.61V. I also
* configure an interrupt for when the data is available or when a pin is above the
* threshold. In my interrupt handler I process data and check which pins exceed 1.61V.
* This document is provided as a reference material
*/
#include "xc.h"
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt(void){
        _AD1IF = 0; //clears interrupt flag
        if(_CHH10 == 1){ //if pin 17 was greater than 1.61V
                _CHH10 = 0; //clears threshold detected flag
                /*code would go here for pin 17*/
        }
        else{
                /*what to do if pin 17 is not above 1.61V*/
        }
        /*DO NOT use else if statements. If the first pin meets the condition all the else if statements
        are ignored. Instead each pin should be an if statement of its own to ensure we are checking
        each pin*/
        if(_CHH9 == 1){ //if pin 18 was greater than 1.61V
                _CHH9 = 0; //clears threshold detected flag
                /*code would go here for pin 18*/
        }
        else{
                /*what to do if pin 18 is not above 1.61V*/
        }
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
```

```
/*pin 17 is RB14 and AN10*/
/*pin 18 is RB15 and AN9*/
_RCDIV = 0b100; // sets a 16 post scaler
/*Reset All Registers We will be Using*/
AD1CON1 = 0;
AD1CON2 = 0;
AD1CON3 = 0;
AD1CON5 = 0;
AD1CSSL = 0;
AD1CSSH = 0;
LATA = 0;
LATB = 0;
TRISA = 0;
TRISB = 0;
ANSA = 0;
ANSB = 0;
/*Set Pins we want to use as Analog Pins and Inputs*/
_ANSB14 = 1; //sets pin 17 as an analog pin
_ANSB15 = 1; //sets pin 18 as an analog pin
_TRISB14 = 1; //sets pin 17 as an input
_TRISB15 = 1; //sets pin 18 as an input
/*Set Positive and Negative Voltage References*/
_PVCFG = 0b00; //selects voltage reference
_NVCFG = 0; //selects ground reference
/*Set Bit Mode*/
_MODE12 = 1; //sets converter to the 12-bit operating mode
/*Set Output Format*/
_FORM = 0b00; //sets converter to absolute decimal format
/*Set Auto Sampling*/
_ASAM = 1; //enables auto sampling
/*Set Auto Scanning of Inputs*/
_CSCNA = 1; //auto scanning enabled
/*Define which Pins to Scan*/
_CSS10 = 1; //enables scanning of pin 17
_CSS9 = 1; //enables scanning of pin 18
/*Choose Interrupt Rate*/
//(2 – 1) in binary is 00001
_SMPI = 00001; // interrupts every other sample
/*Define the Analog to Digital Converter Clock Cycle*/
_ADCS = 0b00000000; //sets clock to be the same as the system clock
/*Set Conversion Timing*/
_SSRC = 0b0101; //sets timer one as managing the timing for the converter
/*Configure Timer 1*/
```

```
        T1CON = 0; //clears timer 1 configuration register
        T1CONbits.TCKPS = 0b00; //sets a prescaler value of 1
        T1CONbits.TCS = 0; //sets the clock to the internal oscillator
        PR1 = 100; //sets the timer period
        T1CONbits.TON = 1; //turn on timer 1
        /*Define the Auto Sample Time*?
        _SAMC = 0b00001; //set to one clock cycle but not needed with threshold detect
        /*Enable Threshold Detect*/
        _ASEN = 1;  //enables threshold detect feature
        /*Define Threshold Detect Interrupt Operation*/
        _ASINT = 0b01; //sets interrupt to occur when input scanning completes
        /*Set Output Location*/
        _BUFREGEN = 1; //sets output to buffers corresponding to pins
        /*Define Threshold Type*/
        _CM1 = 0; //sets the threshold type to greater than
        _CM0 = 1;
        /*Setting the Thresholds*/
        ADC1BUF10 = 2000; //sets the voltage threshold to 1.61V
        ADC1BUF9 = 2000; //sets the voltage threshold to 1.61V
        /*What to do with the Converted Value*/
        _WM1 = 1; //sets the PIC24 to discard converted values
        _WM0 = 0;
        /*Configure the Analog to Digital Converter Interrupt*/
        _AD1IP = 4; //sets interrupt priority to 4
        _AD1IF = 0; //clears interrupt flag
        _AD1IE = 1; //enables interrupt
        /*Turn on Analog to Digital Converter*/
        _ADON = 1; //turns analog to digital converter on
        /*Let the Interrupts Control the Rest of the Code*/
        while(1);
        return 0;
}
```

Configuring a Timer and Timer Interrupt

About the Timers on the PIC24

The PIC24 has five 16-bit timers on board, four of which are linked to make two 32-bit timers. You can unlink one or both of the 32-bit timers if you so desire. The bits determine how many cycles you can count and thus how long your timer may be without a prescaler. A 16-bit timer can count 65,535 cycles and a 32-bit timer can count 4.29 X 10^9 cycles. To convert this into time use the following equation:

$$\frac{1}{f_{\text{oscillator}}} \times \text{Prescaler} \times \text{Cycles} = \text{Time}$$

Where $f_{\text{oscillator}}$ is the frequency of your oscillator after the postscaler and the cycles is how many cycles you intend to count. Alternatively, you can solve the equation with a time in mind and find how many cycles you need. If you set a prescaler the timer will only count every so many cycles effectively increasing the time you can count. A prescaler of two means every other cycle it counts, of four every fourth cycle, etc. Bear this in mind for your designs because timers are used to operate different features such as the threshold detect on the analog to digital converter and input capture. Timers are also useful for ensuring something happens every so often even if other code may be running.

Clear the Timer Register

To start clear the Register of the timer you intend to use. TO do this sue the following line of code:

```
TXCON = 0;
```

Where X is replaced with the value of the timer you wish to use, being 1-5. If you are using the 32-bit timers, timers 2 and 4 are the timers whose registers correspond to those controls.

Optional: Unlink Timers

If you are unlinking timer 3 from timer 2 or timer 5 from timer 4 to have more 16-bit timers you need to use the following line of code:

```
TXCONbits.T32 = 0;
```

Where X is 2 or 4 depending on which timer you are resetting. The first line clears the parent timer settings and the second unlinks the child timers. After doing this you can configure both timers normally.

Set the Timer Clock Source

While there are different options for the clock source, it is always recommended to use the internal oscillator as the clock. If you desire to choose otherwise refer to the manual for instructions on how and what to do for this. To set the clock sue the following line of code:

```
TXCONbits.TCS = 0;
```

Where X is the number of the timer you are configuring.

Set the Timer Prescaler

We need to decide which prescaler to use for our timer to match the time we want to count. The options are as follows:

11 = 1:256
10 = 1:64
01 = 1:8
00 = 1:1

Where the value after the colon is the value of your prescaler. IMPORTANT: The prescaler value is reset whenever we write to the TMRX register, we turn off the timer, or the device resets. If you plan on turn off and turning on the timer or prematurely resetting the timer due to some event make sure you set the prescaler again. To set it use the following line of code:

```
TYCONbits.TCKPS = XX;
```

Where Y is the number of the timer you're configuring and XX is the code for the prescaler you would like to use.

Set the Timer Period

In order for the timer to function correctly we need to tell it how long to count for. After calculating how many cycles you will need use the following line of code to set the period of the timer:

```
PRX = Y;
```

Where X is the number of the timer you are configuring and Y is the number of cycles you intend to count.

Configuring the Timer Interrupt

To know when the timer has finished counting we need to configure an interrupt that will be called when that happens. To do so we use the following lines of code:

```
_TXIP = Y;
_TXIF = 0;
_TXIE = 1;
```

Where X is the number of the timer which you are configuring. Y represents the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _TXIF command clears the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with responding to the timer. Alternatively, you could turn the timer off.

Handling the Timer Interrupt

When the interrupt is called we need to make a handler that clears the interrupt flag and reacts to the interrupt. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_T$Interrupt(void){
        _T$IF = 0;
        /*your code here*/
}
```

Where the $ in the handler name is replaced with the number of the timer you are configuring. The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Turning on the Timer

The last step is to turn on the timer with the following line of code:

```
TXCONbits.TON = 1;
```

Where X is the number of the timer you are configuring

Optional: Gated Timer Operation Mode

The timers have an additional functionality that may or may not be useful to you. This function known as gated accumulation makes use of preset pins which must be set as digital inputs. The pins and the timers they are associated with are as follows:

T1CK (pin 13) = Timer 1
T2CK (pin 18) = Timer 2
T3CK (pin 18) = Timer 3
T4CK (pin 6) = Timer 4
T5CK (pin 6) = Timer 5

Gated accumulation means that when the corresponding pin goes high the timer will start counting and throw an interrupt when the signal goes low or when the timer counts its full period, whichever comes first. This is great for if you are receiving a signal that you need to know if it stays on longer or shorter than a certain time period. An example would be if you wanted to know if a limit switch had been pressed for more than half a second and do something if it was. You would enable this and then in your while loop or other control structure check the TMRX value to see how long it has been. It is important to note that the time that it was on does not get saved anywhere when the interrupt is thrown and is lost. Use input capture to track how long something has been on and use the time for calculations or inputs. To enable this feature, use this line of code:

```
TXCONbits.TGATE = 1;
```

Where X is replaced with the number of the timer you are configuring.

Code Example – Configuring Timer 2 with a Timer Interrupt

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 7, 2022
* Description: In this code I configure timer 2 to count for half a second and then throw
* an interrupt. I also configure a handler for that interrupt that would run code every
* half second.
* This document is provided as a reference material
*/
#include "xc.h"
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void){
        _T2IF = 0; //clears interrupt flag
        /*your code here*/
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        _RCDIV = 0b100; //sets a 16 post scaler
        /*Reset All Registers We will be Using*/
        T2CON = 0;
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Unlink Timer 2 from Timer 3*/
        T2CONbits.T32 = 0; //unlinks timer 2 from timer 3 so it is a 16-bit timer
        /*Set the Timer 2 Clock Source*/
        T2CONbits.TCS = 0; //sets timer 2 clock to the internal oscillator
        /*Set Timer 2 Prescaler Value*/
        /*I want to be able to count for half a second so my math would be ((.5/65535)*250000) =
        Prescaler. This equals 1.9 and I can only go up so I set a prescaler value of 8 in order to be able
        to count half a second*/
        T2CONbits.TCKPS = 0b01;
        /*Set the Timer Period*/
        /*With my prescaler selected I need to solve now for the cycles I need to count in order to get
```

```
    half a second. My math for this would be as follows: ((.5/8)*250000) which gives me 15625*/
    PR2 = 15625 //sets the timer period to half a second
    /*Configure the Timer 2 Interrupt*/
    _T2IP = 4; //sets the timer 2 interrupt priority to 4
    _T2IF = 0; //clears the timer 2 interrupt flag
    _T2IE = 1; //enables the timer 2 interrupt flag
    /*Turn on Timer 2*/
    T2CONbits.TON = 1;
    /*Let the Interrupt Control the Rest of the Code*/
    while(1);
    return 0;
}
```

Configuring a PWM Signal and Counter Interrupt without Fault Inputs

Output Compare Registers

The PIC24 has 3 built in PWM signals that are available for use. Each of these has their own set of control registers that are called the output compare registers. The PWM pins and their associated registers are as listed:

OC1CON1
OC1CON2: Pin 14
OC2CON1
OC2CON2: Pin 4
OC3CON1
OC3CON2: Pin 5

Depending on which pin you plan on using refer to this list to know which register to call and configure.

Clear Relevant Registers

After clearing all the ANSX, TRISX, and LATX registers to make sure the pins are reset, you should also clear the output compare registers associated with the pin you plan on using. To do this use these lines of code for each pin you plan on using:

```
OCXCON1 = 0;
OCXCON2 = 0;
```

Where the X is replaced with the associated value for the pin you are using.

Select the Output Compare Clock

Each output compare pin has its own dedicated timer in the output compare module so that it can time its period and duty cycle. We need define which source the internal timer uses and the options are as follows:

111 = System clock
110 = Reserved
101 = Reserved
100 = Timer1
011 = Timer5
010 = Timer4
001 = Timer3
000 = Timer2

If you use any of the timer options you need to have that timer configured and on before you enable the output compare module. Unless the period you want to use is more than a 16-bit timer could count, you should use the system clock. If you want to use another option refer to the manual for more instructions. Another option to look at in the manual if you have a very long period is cascading OC1 and OC2 to create a 32-bit PWM counter. This also can increase the period but will not be covered further in this guide but is covered in the manual. To set the clock use the following line of code:

```
OCXCON1bits.OCTSEL = 0bYYY;
```

Where X is the number corresponding to the pin you are using and YYY is the value of the clock you intend on using.

Select the Output Compare Operating Mode

When using the output compare module there are 8 operating modes that can be used. 6 of them deal with generating a pulse or a delayed output. For more information refer to the manual. The two we are concerned with are the following options:

111 = Center-Aligned PWM on OCx
110 = Edge-Aligned PWM on OCx

Where center-aligned PWM means that the high portion of the PWM is set in the middle of the period rather at the beginning. The edge-aligned PWM mode has the signal start high and go low after the duty cycle is reached, then stays low until the period is over. If you intend on using the optional fault pin capabilities mentioned later on you must use the center-aligned PWM option. Besides that, either or is fine for a PWM signal. Most times defaulting to an edge-aligned PWM is easier for people to understand and diagnose with an oscilloscope. To set the operating mode use the following line of code:

```
OCXCON1bits.OCM = 0bYYY;
```

Where X is the corresponding number for the pin you are configuring and YYY is the code for the option you chose.

Set the Output Compare Module to Sync Mode

The output compare module can be triggered by or synced by a module. Trigger mode is used for the other operating modes of the output compare module. For PWM applications it should be set to be synced by a source to keep the module timed properly. To set it to sync mode use this line of code:

```
OCXCON2bits.OCTRIG = 0;
```

Where X is the corresponding number for the pin you are configuring.

Set the Output Compare Module Sync Source

The output compare module timer counts according to the clock that we defined earlier but the module must know when the end of a period occurs. There are many options that can be used to sync with but for PWM applications we should have the module sync with its own timer to ensure the duty cycle and frequency are properly set. To look at other options look at the manual for more information. To set the sync source to the output compare module itself use this line of code:

```
OCXCON2bits.SYNSEL = 0b11111;
```

Where X is the corresponding number for the pin you are configuring.

Set the PWM Period

To set the period of the PWM signal we need to calculate what value the output compare module needs to count to. Use the following equation to find that value:

$$\frac{\frac{1}{f_{\text{PWM}}}}{\frac{1}{f_{\text{oscillator}}}} - 1 = \text{Period}$$

Where f_{PWM} is your desired frequency and $f_{\text{oscillator}}$ is your set clock frequency. After you have that value use the following line of code to set the period:

```
OCXRS = Period;
```

Where X is the corresponding number for the pin you are configuring.

Set the PWM Duty Cycle

To set the PWM duty Cycle value use the following equation:

$$\text{Period} \times \frac{\%\ \text{Duty Cycle}}{100} = \text{Duty Cycle}$$

After finding your duty cycle value use this line of code to set your duty cycle:

```
OCXR = Duty Cycle;
```

Where X is the corresponding number for the pin you are configuring.

Configuring a Counter Interrupt

A counter interrupt occurs after every pulse from a PWM signal. It can be used to count the pulses for things like counting steps on a stepper motor, counting time, or reacting to sensors while a PWM signal is being driven. To configure it use the following lines of code:

```
_OCXIP = Y;
_OCXIF = 0;
_OCXIE = 1;
```

Where X represents the module which you are configuring. Y is the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _OCXIF command clears the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with counting the PWM cycles.

Handling a Counter Interrupt

When the interrupt is called we need to make a handler that clears the interrupt flag and reacts to the interrupt. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_OC$Interrupt(void){
        _OC$IF = 0;
        /*your code here*/
}
```

Where the $ in the handler name is replaced with the number of the module you are configuring. The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Optional: Using the Counter Interrupt without a PWM Output

The output compare module can be used in PWM mode like a timer interrupt to run code at consistent intervals and leave the pin open for other uses. This is done by using the following line of code:

```
OCXCON2bits.OCTRIS = 1;
```

Where X is the number of the module which you are configuring. This sets the output of the compare module to be tri-stated, meaning a high impedance state that effectively removes it from the other circuits on the PIC24. This means we can still use the pin as an analog input or digital I/O pin while using the output compare module.

Optional: Inverting the PWM Output

The output compare module can be inverted where the duty cycle setting is treated as the low time percentage of the period. To set this use this line of code:

```
OCXCON2bits.OCINV = 1;
```

Where X is the number of the module which you are configuring.

Code Example – Configuring a PWM Signal and Counter Interrupt without Fault Inputs

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 8, 2022
* Description: In this code I configure the PWM signal on pin 14 to run at 50Hz and
* have a duty cycle of 50%. I also configure a counter interrupt that will increment a
* global variable for each pulse of the PWM.
* This document is provided as a reference material
*/
#include "xc.h"
int count = 0; //a global variable I want to increment in my counter interrupt
void __attribute__((interrupt, no_auto_psv)) _OC1Interrupt(void){
        _OC1IF = 0; //clears interrupt flag
        count++; //increments my global variable
        /*your code here*/
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        _RCDIV = 0b100; //sets a 16 post scaler
        /*Reset All Registers We will be Using*/
        OC1CON1 = 0;
        OC1CON2 = 0;
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Set Output Compare Module Clock*/
        OC1CON1bits.OCTSEL = 0b111; //tells the OC1 module to use the system clock
        /*Set the Output Compare Operating Mode*/
        OC1CON1bits.OCM = 0b110; //sets the OC1 module to Edge-Aligned PWM mode
        /*Set the Output Compare Module to Sync Mode*/
        OC1CON2bits.OCTRIG = 0; //sets the OC1 module to sync mode and not trigger mode
```

```
	/*Set the Output Compare Module Sync Source*/
	OC1CON2bits.SYNSEL = 0b11111; //sets the OC1 module to sync to its own counter
	/*Set the PWM Period*/
	/*To get 50Hz on the PWM my math would be ((1/50)/(1/250000)) – 1. This gives me 4999
	which is in the range of the PWM 16-bit counter*/
	OC1RS = 4999; //sets OC1 period to 50Hz
	/*Set PWM Duty Cycle*/
	/*For a 50% duty cycle I divide the value I got for the period by two which is just under
	2500*/
	OC1R = 2500; //sets a 50% duty cycle
	/*Configuring the Counter Interrupt*/
	_OC1IP = 4; //sets the OC1 counter interrupt priority to 4
	_OC1IF = 0; //clears the OC1 counter interrupt flag
	_OC1IE = 1; //enables the OC1 counter interrupt
	/*Let the Interrupts Control the Rest of the Code*/
	while(1);
	return 0;
}
```

Configuring a PWM Signal and Counter Interrupt with Fault Inputs

Output Compare Registers

The PIC24 has 3 built in PWM signals that are available for use. Each of these has their own set of control registers that are called the output compare registers. The PWM pins and their associated registers are as listed:

OC1CON1
OC1CON2: Pin 14
OC2CON1
OC2CON2: Pin 4
OC3CON1
OC3CON2: Pin 5

Depending on which pin you plan on using refer to this list to know which register to call and configure.

Clear Relevant Registers

After clearing all the ANSX, TRISX, and LATX registers to make sure the pins are reset, you should also clear the output compare registers associated with the pin you plan on using. To do this use these lines of code for each pin you plan on using:

```
OCXCON1 = 0;
OCXCON2 = 0;
```

Where the X is replaced with the associated value for the pin you are using.

Select the Output Compare Clock

Each output compare pin has its own dedicated timer in the output compare module so that it can time its period and duty cycle. We need define which source the internal timer uses and the options are as follows:

111	= System clock
110	= Reserved
101	= Reserved
100	= Timer1
011	= Timer5
010	= Timer4
001	= Timer3
000	= Timer2

If you use any of the timer options you need to have that timer configured and on before you enable the output compare module. Unless the period you want to use is more than a 16-bit timer could count, you should use the system clock. If you want to use another option refer to the manual for more instructions. Another option to look at in the manual if you have a very long period is cascading OC1 and OC2 to create a 32-bit PWM counter. This also can increase the period but will not be covered further in this guide but is covered in the manual. To set the clock use the following line of code:

```
OCXCON1bits.OCTSEL = 0bYYY;
```

Where X is the number corresponding to the pin you are using and YYY is the value of the clock you intend on using.

Select the Output Compare Operating Mode

When using the output compare module there are 8 operating modes that can be used. 6 of them deal with generating a pulse or a delayed output. For more information refer to the manual. The two we are concerned with are the following options:

111 = Center-Aligned PWM on OCx
110 = Edge-Aligned PWM on OCx

Where center-aligned PWM means that the high portion of the PWM is set in the middle of the period rather at the beginning. The edge-aligned PWM mode has the signal start high and go low after the duty cycle is reached, then stays low until the period is over. If you intend on using the optional fault pin capabilities mentioned later on you must use the center-aligned PWM option. Besides that, either or is fine for a PWM signal. Most times defaulting to an edge-aligned PWM is easier for people to understand and diagnose with an oscilloscope. To set the operating mode use the following line of code:

OCXCON1bits.OCM = 0bYYY;

Where X is the corresponding number for the pin you are configuring and YYY is the code for the option you chose.

Set the Output Compare Module to Sync Mode

The output compare module can be triggered by or synced by a module. Trigger mode is used for the other operating modes of the output compare module. For PWM applications it should be set to be synced by a source to keep the module timed properly. To set it to sync mode use this line of code:

OCXCON2bits.OCTRIG = 0;

Where X is the corresponding number for the pin you are configuring.

Set the Output Compare Module Sync Source

The output compare module timer counts according to the clock that we defined earlier but the module must know when the end of a period occurs. There are many options that can be used to sync with but for PWM applications we should have the module sync with its own timer to ensure the duty cycle and frequency are properly set. To look at other options look at the manual for more information. To set the sync source to the output compare module itself use this line of code:

OCXCON2bits.SYNSEL = 0b00000;

Where X is the corresponding number for the pin you are configuring.

Set the PWM Period

To set the period of the PWM signal we need to calculate what value the output compare module needs to count to. Use the following equation to find that value:

$$\frac{\frac{1}{f_{\mathrm{PWM}}}}{\frac{1}{f_{\mathrm{oscillator}}}} - 1 = \mathrm{Period}$$

Where f_{PWM} represents your desired frequency and $f_{\mathrm{oscillator}}$ represents your set clock frequency. After you have that value use the following line of code to set the period:

OCXRS = Period;

Where X is the corresponding number for the pin you are configuring.

Set the PWM Duty Cycle

To set the PWM duty Cycle value use the following equation:

$$\mathrm{Period} \times \frac{\%\ \mathrm{Duty\ Cycle}}{100} = \mathrm{Duty\ Cycle}$$

After finding your duty cycle value use this line of code to set your duty cycle:

OCXR = Duty Cycle;

Where X is the corresponding number for the pin you are configuring.

Using Fault Inputs

The PIC24 has 3 fault options that can be used with the output compare modules. Depending on the fault configuration you choose you could have it so if the fault pin goes low the PWM signal pin stops outputting a signal and instead is either forced to be low or high, until the fault pin goes high or until you programmatically reset the fault signal. The fault options are as follows:

Comparator Module (Comparator 1 for OC1 and OC2 and Comparator 2 for OC3)
OCFA = Pin 17;
OCFB = Pin 16;

The first option uses one of the 3 on board comparator modules for its operation. You must enable this fault on the module you want it to affect after you have configured and enabled the associated comparator. Refer to the manual for more information on configuring the comparator module and using it. The second and third options are two active low pins that are shared for all output compare modules.

Enabling Fault Inputs

To use fault inputs, use the line of code below that corresponds to the fault you want to use:

```
OCXCON1bits.ENFLT2 = 1;
OCXCON1bits.ENFLT1 = 1;
OCXCON1bits.ENFLT0 = 1;
```

Where X is the corresponding number for the output compare module, you are configuring. The first line enables the comparator fault for the module of your choice, the second the OCFB fault, and the third the OCFA fault.

Determining Fault Input Mode

After setting which modules will use what faults you need to define how these faults will impact the modules and how to respond to the fault. The following options deal with the fault operating mode:

1 = Fault mode is maintained until the Fault source is removed and the corresponding OCFLTx bit is cleared in software
0 = Fault mode is maintained until the Fault source is removed and a new PWM period starts

Where the first option states that after a fault is removed the user must clear the fault bit to resume the PWM signal and the second states that after the fault is removed the PWM resumes by itself. After selecting the operating mode you would like to enable, use this line of code to set it:

```
OCXCON2bits.FLTMD = Y;
```

Where X is the corresponding number for the pin you are configuring. And Y is the option you chose.

Determining Fault Input Effect

The next option determines how the PWM pin reacts to a fault and the options are as follows:

1 = Pin is forced to an output on a Fault condition
0 = Pin I/O condition is unaffected by a Fault

In the first option you decide which output should be forced on the PWM pin. The second option leaves it wherever the PWM signal was at be it high or low. To set this option use this line of code:

```
OCXCON2bits.FLTTRIEN = Y;
```

Where X is the corresponding number for the pin you are configuring. And Y is the option you chose. If you chose the first option you also must change the following setting which determines what to output on a fault condition:

1 = PWM output is driven high on a Fault
0 = PWM output is driven low on a Fault

After making a choice set it with this line of code:

```
OCXCON2bits.FLTOUT = Y;
```

Where X is the corresponding number for the pin you are configuring. And Y is the option you chose.

Resetting Fault Input Status

The last thing you may need to do is programmatically reset the fault status bit if you set the fault input mode, FLTMD, to 1. If you did you need a line of code from the list below somewhere in your code when you want the PWM to start again after the fault condition is resolved:

```
OCXCON1bits.OCFLT2 = 0;
OCXCON1bits.OCFLT1 = 0;
OCXCON1bits.OCFLT0 = 0;
```

Where X is the corresponding number for the pin you are configuring. The first line is for if a comparator fault was tripped, the second if OCFB fault was tripped, and the third for if the OCFA fault was tripped.

Configuring a Counter Interrupt

A counter interrupt occurs after every pulse from a PWM signal. It can be used to count the pulses for things like counting steps on a stepper motor, counting time, or reacting to sensors while a PWM signal is being driven. To configure it use the following lines of code:

```
_OCXIP = Y;
_OCXIF = 0;
_OCXIE = 1;
```

Where X represents the module which you are configuring. Y is the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _OCXIF command clears the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with counting the PWM cycles.

Handling a Counter Interrupt

When the interrupt is called we need to make a handler that clears the interrupt flag and reacts to the interrupt. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_OC$Interrupt(void){
        _OC$IF = 0;
        /*your code here*/
}
```

Where the $ in the handler name is replaced with the number of the module you are configuring. The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Optional: Using the Counter Interrupt without a PWM Output

The output compare module can be used in PWM mode like a timer interrupt to run code at consistent intervals and leave the pin open for other uses. This is done by using the following line of code:

```
OCXCON2bits.OCTRIS = 1;
```

Where X is the number of the module which you are configuring. This sets the output of the compare module to be tri-stated, meaning a high impedance state that effectively removes it from the other circuits on the PIC24. This means we can still use the pin as an analog input or digital I/O pin while using the output compare module.

Optional: Inverting the PWM Output

The output compare module can be inverted where the duty cycle setting is treated as the low time percentage of the period. To set this use this line of code:

```
OCXCON2bits.OCINV = 1;
```

Where X is the number of the module which you are configuring.

Code Example – Configuring a PWM Signal and Counter Interrupt with Fault Inputs

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 8, 2022
* Description: In this code I configure the PWM signal on pin 14 to run at 50Hz and
* have a duty cycle of 50%. I then configure a counter interrupt that will increment a
* global variable for each pulse of the PWM. Finally, I configure a fault interrupt on
* OCFB (pin 16) that will drive the PWM pin to a low output if it goes low.
* This document is provided as a reference material
*/
#include "xc.h"
int count = 0; //a global variable I want to increment in my counter interrupt
void __attribute__((interrupt, no_auto_psv)) _OC1Interrupt(void){
        _OC1IF = 0; //clears interrupt flag
        count++; //increments my global variable
        /*your code here*/
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        _RCDIV = 0b100; //sets a 16 post scaler
        /*Reset All Registers We will be Using*/
        OC1CON1 = 0;
        OC1CON2 = 0;
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Set Output Compare Module Clock*/
        OC1CON1bits.OCTSEL = 0b111; //tells the OC1 module to use the system clock
        /*Set the Output Compare Operating Mode*/
        OC1CON1bits.OCM = 0b110; //sets the OC1 module to Edge-Aligned PWM mode
        /*Set the Output Compare Module to Sync Mode*/
```

```
	OC1CON2bits.OCTRIG = 0; //sets the OC1 module to sync mode and not trigger mode
	/*Set the Output Compare Module Sync Source*/
	OC1CON2bits.SYNSEL = 0b11111; //sets the OC1 module to sync to its own counter
	/*Set the PWM Period*/
	/*To get 50Hz on the PWM my math would be ((1/50)/(1/250000)) – 1. This gives me 4999
	which is in the range of the PWM 16-bit counter*/
	OC1RS = 4999; //sets OC1 period to 50Hz
	/*Set PWM Duty Cycle*/
	/*For a 50% duty cycle I divide the value I got for the period by two which is just under
	2500*/
	OC1R = 2500; //sets a 50% duty cycle
	/*Enable OCFB Fault Input*/
	OC1CON1bits.ENFLT1 = 1; //enables the OCFB (pin 16) fault input
	/*Set Fault Input Mode*/
	OC1CON2bits.FLTMD = 0; //PWM starts automatically after the fault goes high
	/*Set Fault Input Effect*/
	OC1CON2bits.FLTTRIEN = 1; //forces pin 14 to an output when faulted
	/*Set PWM Output after Fault Input*/
	OC1CON2bits.FLTOUT = 0; //forces pin 14 to be low when faulted
	/*Configuring the Counter Interrupt*/
	_OC1IP = 4; //sets the OC1 counter interrupt priority to 4
	_OC1IF = 0; //clears the OC1 counter interrupt flag
	_OC1IE = 1; //enables the OC1 counter interrupt
	/*Let the Interrupts Control the Rest of the Code*/
	while(1);
	return 0;
}
```

Configuring Input Capture and Input Capture Interrupts

About Input Capture

The PIC24 has 3 input compare modules onboard that can measure how long a pin has been on or off. Each of these modules has its own in built 16-bit timer. The first two can also be linked to make a 32-bit timer if needed. This however can most often be avoided with careful planning on the part of the user. If you intend on using a 32-bit timer see the manual for more details on how this is done. Use input capture when measuring pulses from devices like the HC-SR04 ultrasonic range finder whose output is a pulse whose width tells you how far away it sensed an obstacle. See the Code Example below on page 62 for more information on how to do that. The pins that these modules are connected to are as follows:

IC1 = pin 14
IC2 = pin 13
IC3 = pin 15

Clear Input Capture Control Registers

Each of the input capture modules on the PIC24 has two registers that control the settings for its use. You should clear these registers when you start your code with these lines of code:

```
ICXCON1 = 0;
ICXCON2 = 0;
```

Where X is replaced with the number of the module you are configuring.

Select Input Capture Clock

The input capture modules need a clock to count off of for their timers. It is important to note that the input capture module timers have no prescaling capabilities so if you anticipate a longer period than a 16-bit timer with your clock frequency would read you should plan on using one of the timers that is properly setup with a prescaler value to divide your clock frequency. The options for the clock selection are as follows:

111 = System clock (FOSC/2)
110 = Reserved
101 = Reserved
100 = Timer1
011 = Timer5
010 = Timer4
001 = Timer2
000 = Timer3

Use the following line of code after having made your selection:

```
ICXCON1bits.ICXTSEL = 0bYYY;
```

Where X is replaced with the value of module you are configuring and YYY is replaced with the option you chose.

Select Input Capture Mode

The input capture module needs to know when its needs to write its timer value to its FIFO buffer. The mode options and a definition of each are as follows:

101 = Prescaler Capture mode: Capture on every 16th rising edge – This means that you will have started the ICX timer earlier and when it has counted 16

rising edges whatever the value of that timer is gets saved to the ICX FIFO buffer.

100 = Prescaler Capture mode: Capture on every 4th rising edge – This means that you will have started the ICX timer earlier and when it has counted 4 rising edges whatever the value of that timer is gets saved to the ICX FIFO buffer.

011 = Simple Capture mode: Capture on every rising edge – This means that you will have started the ICX timer earlier and whenever a rising edge occurs the value of that timer gets saved to the ICX FIFO buffer.

010 = Simple Capture mode: Capture on every falling edge – This means that you will have started the ICX timer earlier and whenever a falling edge occurs the value of that timer gets saved to the ICX FIFO buffer.

001 = Edge Detect Capture mode: Capture on every edge (rising and falling); ICI<1:0> bits do not control interrupt generation for this mode – This means that you will have started the ICX timer earlier and whenever a rising edge or a falling edge occurs the value of that timer gets saved to the ICX FIFO buffer. Also, the interrupt will go off after every change in the pins value and this cannot be changed.

000 = Input capture module is turned off

If you are trying to read a pulse coming in you should use the Edge Detect Capture mode because it makes it easy to start and end the timer as the signal comes in. The other options are more useful for trying to count incoming pulses and time the interval between them. Later we will discuss timing options and how they relate to the options above.

Select Input Capture Timing Method

The input capture modules have three options on how they can tell when to start their timer and when to reset it. These are the triggered, synchronous, and normal operation modes. The triggered mode means that the software or a hardware event starts the timer and the user must end it programmatically. These events could be as follows: another input capture module saving its data, a comparator returning true, an analog to digital converter finishing its conversions, an output compare module finishing one of its periods, or a timer reaching its period. The synchronous operating mode means that the timer resets at the same time that another module begins on the PIC24. The options for this mode are an output compare module, an input capture module, or a timer. The normal operating mode means when you turn the module on it will count up to its limit and reset only after it reaches that value. To summarize these options, look at this list:

Normal Operation Mode – The timer operates as a standard timer that rolls over when it reaches its max value.

Synchronous Operation Mode – The timer synchronizes itself with another timer so that both roll over simultaneously

Software Triggered Operation Mode – The timer starts when the user sets the TRIGSTAT bit and ends when the user clears the TRIGSTAT bit

Hardware/Software Triggered Operation Mode – The timer starts when a hardware event occurs or when the user sets the TRIGSTAT bit. It ends when the user clears the TRIGSTAT bit.

If you are using the Edge Detect Capture mode, I suggest using the Software Triggered Operation Mode since it will make your life easier to measure the time between the two edges. Once you have chosen an option from above hop down to section that follows which explains how to configure that mode.

Normal Operation Mode

In normal operation mode the timer is always running and when an edge is detected that matches your input capture mode the time is written to the FIFO buffer. This mode can work well for timing the length of a pulse or the time between pulse but it requires more thought in the interrupt handler to properly find the correct time. To use this mode, add the following lines of code:

```
ICXCON2.ICTRIG = 0;
ICXCON2.SYNCSEL = 0b00000;
```

Where X is replaced with the value of the input capture module you're configuring.

Synchronous Operation Mode

In synchronous operation mode the falling edge of a sync input signal resets the timer. You can use this to ensure that the timer of the input capture module counts in sync with another timer or module. This can be useful for if you want to only have the max period of the input capture module be smaller than normal and match another 16-bit timer without prescaler. Similar to the normal operation though it may be difficult to time a pulse width or time how long before another signal occurs since the timer is not reset upon getting a signal. To set this mode use the following lines of code:

```
ICXCON2.ICTRIG = 0;
ICXCON2.SYNCSEL = 0bYYYYY;
```

Where X is replaced with the value of the input capture module you're configuring. YYYYY is replaced with one of the following options which determines what the timer should sync to:

10111 =	Input Capture 4
10110 =	Input Capture 3
10101 =	Input Capture 2
10100 =	Input Capture 1
01111 =	Timer5
01110 =	Timer4
01101 =	Timer3
01100 =	Timer2
01011 =	Timer1
01010 =	Input Capture 5
00101 =	Output Compare 5
00100 =	Output Compare 4
00011 =	Output Compare 3
00010 =	Output Compare 2
00001 =	Output Compare 1

Software Triggered Operation Mode

In software triggered operation mode, the timer is held in reset till the TRIGSTAT bit is set high by the programmer. Then, until the TRIGSTAT bit is set low, the timer will count up and roll over at its max period. This is very useful because we can use the edge detect input capture mode to set the TRIGSTAT bit at the start of a pulse and turn it off at the end of a pulse. This is very helpful for timing the length of a pulse. For timing the amount of time between pulses we could set the TRIGSTAT bit when we get the first rising edge and clear it when we get the second rising edge. In both cases this will ensure our timer only counts when we want to measure and not while idle. To set this mode use the following lines of code:

```
ICXCON2.ICTRIG = 1;
ICXCON2.SYNCSEL = 0b00000;
```

Where X is replaced with the value of the input capture module you're configuring.

Hardware/Software Triggered Operation Mode

In hardware/software triggered operation mode an external module event can start the timer and have it run until the programmer clears it. It also can function like the normal software triggered operation mode. This can be useful if you anticipate an input signal at the end of a timer or other module that you want to measure. To set this mode use the following lines of code:

```
ICXCON2.ICTRIG = 1;
ICXCON2.SYNCSEL = 0bYYYYY;
```

Where X is replaced with the value of the input capture module you're configuring. YYYYY is replaced with one of the following options which determines what the timer should be triggered by:

11100 = CTMU
11011 = A/D
11010 = Comparator 3
11001 = Comparator 2
11000 = Comparator 1
10111 = Input Capture 4
10110 = Input Capture 3
10101 = Input Capture 2
10100 = Input Capture 1
01111 = Timer5
01110 = Timer4
01101 = Timer3
01100 = Timer2
01011 = Timer1
01010 = Input Capture 5
00101 = Output Compare 5
00100 = Output Compare 4
00011 = Output Compare 3
00010 = Output Compare 2
00001 = Output Compare 1

Interrupt Rates

The input capture module can create interrupts when it saves data to its buffer. If you want it to save multiple measurements before setting an interrupt use this line of code:

```
ICXCON1bits.ICI = 0bYY;
```

Where X is replaced with the number of the input capture module you are configuring and YY is replaced with one of these options:

11 = Interrupt on every fourth capture event
10 = Interrupt on every third capture event
01 = Interrupt on every second capture event
00 = Interrupt on every capture event

It is important to note however that if you are using edge detect mode these bits will not affect the interrupt. In that mode it interrupts on every edge it finds.

Getting Data from the FIFO Buffer

Each input capture module has a FIFO buffer with space for four measurements. Here is a diagram to visualize it:

MOST RECENT DATA- FIRST READ
X
X
FURTHEST DATA – LAST READ

This buffer can present complications when reading data because fi your module is running at really fast speeds your data can get buried or lost. One thing you can do before making an important measurement is clear the buffer. To do this you have to read all the data in it. Writing a value to the buffer only adds data to it. To tell when you the buffer is clear there is a bit that the hardware sets high when the buffer is not empty. To reset the buffer, use the following code:

```
while(ICXCON1bits.ICBNE == 1){
        int temp = ICXBUF;
}
```

Where X is replaced with the value of the input capture module you are reading from. Each read will clear one value from the buffer so the while loop ensures enough reads are made to clear it all. With careful programming you can avoid getting improper data and only getting relevant values.

Configuring an Input Capture Interrupt

An input capture interrupt occurs after a certain number of signals have been captured. To configure it use the following lines of code:

```
_ICXIP = Y;
_ICXIF = 0;
_ICXIE = 1;
```

Where X represents the module which you are configuring. Y is the priority of the interrupt and needs a value between 1 and 7. 1 being least important 7 being most important. This only matters if interrupts will potentially occur at the same time and one has to be done before the other. I always set them as 4. The _ICXIF command clears the interrupt flag. When we handle the interrupt, we must clear the flag so that the code can continue running normally after the interrupt handler or we will be infinitely stuck in the handler. The last line enables the interrupt. You can enable and disable the interrupt throughout your code which can be useful if there are times where you are not concerned with capturing the input signals timing.

Handling an Input Capture Interrupt

When the interrupt is called we need to make a handler that clears the interrupt flag and reacts to the interrupt. This can be done in the main.c file or if you are following the good practices guidelines in your interrupts header file. The handler function is written as follows:

```
void __attribute__ ((interrupt, no_auto_psv))
_IC$Interrupt(void){
        _IC$IF = 0;
        /*your code here*/
}
```

Where the $ in the handler name is replaced with the number of the module you are configuring. The handler must be written with that name in order for the PIC24 to properly function. The interrupt flag line must also always be at the top of the interrupt to clear the flag.

Code Example – Configuring Input Capture with an Input Capture Interrupt

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 13, 2022
* Description: In this code I configure the input capture module on pin 13 to tell me the
* total period of a PWM signal being inputted on that pin. I use only rising edge
* detection and software triggered operation in order to get the proper data. I then
* configure the interrupt to occur every edge it detects and handle the interrupt in my
* code.
* This document is provided as a reference material
* Since I am trying to calculate the period of a PWM signal being inputted I
* should know the limits of my system. To do this I can look at the maximum
* resolution of the input capture module and my oscillator. I know that I cannot
* measure anything less than 6 oscillator cycles due to limitations on the
* interrupts so the fastest PWM frequency I could get would be FOSC/6 which in
* this case would be a max frequency of 41.67 kHz.
*/
#include "xc.h"
float frequency = 0.0; //global variable that stores the value of the incoming frequency
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt(void){
        _IC2IF = 0; //clears interrupt flag
        if(IC2CON2bits.TRIGSTAT == 0){ //At the start of a new PWM period
                IC2CON2bits.TRIGSTAT = 1; //starts IC2 timer
                while(IC2CON1bits.ICBNE == 1){ //clears IC2 buffer
                        int temp = IC2BUF;
                }
        }
        else{
                IC2CON2bits.TRIGSTAT = 0; //stops timer at start of next PWM period
                frequency = (250000/IC2BUF) //gives me the input PWM frequency
        }
}
#pragma config ICS = PGx3
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        _RCDIV = 0b100; //sets a 16 post scaler
        /*
        /*Reset All Registers We will be Using*/
        IC2CON1 = 0;
        IC2CON2 = 0;
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Configure Input Capture 2 Module Clock*/
        IC2CON1bits.IC2TSEL = 0b111; //sets IC2 module clock to the system clock
        /*Configure Input Capture 2 Module Capture Mode*/
        IC2CON1bits.ICM = 0b011; //sets the IC2 module mode to capture rising edges
        /*Configure Input Capture 2 Module in Software Triggered Operation Mode*/
        IC2CON2bits.ICTRIG = 1; //sets the IC2 module to trigger mode
        IC2CON2bits.SYNCSEL = 0b00000; //sets the IC2 module as software triggered
        /*Set Input Capture 2 Module Interrupt Rate*/
        IC2CON1bits.ICI = 0b00; //sets interrupt to occur on every capture event
        /*Configure the Input Capture 2 Module Interrupt*/
        _IC2IP = 4; //sets the IC2 interrupt priority to 4
        _IC2IF = 0; //clears the IC2 interrupt flag
        _IC2IE = 1; //enables the IC2 interrupt
        /*Let the Interrupts Control the Rest of the Code*/
        while(1);
        return 0;
}
```

Code Example – Configuring Input Capture with a HC-SR04 Ultrasonic Range Finder

```
/*
* File:   main.c
* Author: Spencer Mosley
* Created on June 13, 2022
* Description: In this code I configure the input capture module on pin 13 to tell me the
* total length of the echo pulse from the HC-SR04 on that pin. I use the edge detect
* mode to find both the rising and falling edges. I then use the interrupt handler to stop
* and end the timer and calculate the distance it saw. Finally, I configure timer 2 so that
* it will send the 10uS output pulse on pin 12 for the HC-SR04 trigger input and wait .5
* sec between each trigger pulse
* This document is provided as a reference material
* Since I am trying to calculate the length of a pulse being inputted I should
* know the limits of my system. To do this I can look at the maximum length of
* the echo pulse output. From the datasheet I know that it can measure 4 meters
* and it says that uS/58 = cm or uS/148 = in. Therefore, 400 * 58 is the max pulse
* width in uS I can measure. This ends up being 23200uS. At a clock frequency of
* 8MHz we can measure increments of .25uS which means our 16-bit timer
* would need to count 92800 for the full distance possible. Since this is out of
* range we select a postscaler of 2 which gives us a max count of 46400 and a
* resolution of .5uS. That means our distances will have a very small error in
* them but it is a maximum of .5/58 = .0086cm which is negligible in our
* application of not touching a wall.
*/
#include "xc.h"
float distance = 0.0; //global variable that stores the value of the measured distance
void __attribute__((interrupt, no_auto_psv)) _IC2Interrupt(void){
	_IC2IF = 0; //clears interrupt flag
	if(IC2CON2bits.TRIGSTAT == 0){ //At the start of a new pulse
		IC2CON2bits.TRIGSTAT = 1; //starts IC2 timer
		while(IC2CON1bits.ICBNE == 1){ //clears IC2 buffer
			int temp = IC2BUF;
		}
	}
	else{
```

```
                IC2CON2bits.TRIGSTAT = 0; //stops timer at end of pulse period
                distance = IC2BUF/2000000; //converts timer value to seconds
                distance*= 1000000; //converts to uS
                distance/= 58; //converts to cm
                IC2TMR = 0; resets timer for next pulse;
        }
}
void __attribute__((interrupt, no_auto_psv)) _T2Interrupt(void){
        _T2IF = 0; //clears interrupt flag
        if(PR2 == 20){ //if sending the 10uS pulse
                if(_LATB8 == 0){ //if pin 12 is off
                        _LATB8 = 1; //turns pin 12 on
                }
                else{ //if pulse is sent
                        T2CONbits.TON = 0; //turn timer 2 off
                        _LATB8 = 0; //turn off pin 12
                        T2CONbits.TCKPS = 0b10; //sets a timer prescaler value of 64
                        PR2 = 15625; //sets a period of half a second
                        TRM2 = 0; //resets timer 2 value
                        T2CONbits.TON = 1; //turns timer 2 back on
                }
        }
        else if(PR2 == 15625){ //if we waited half a second
                T2CONbits.TON = 0; //turn off timer 2
                T2CONbits.TCKPS = 0b00; //sets a timer prescaler value of 1
                PR2 = 20; //sets timer 2 period to 10uS
                TMR2 = 0; //resets timer 2 value
                T2CONbits.TON = 1; //turns timer 2 back on;
        }
}
#pragma config ICS = PGx3 //configures programming and debugging pins
#pragma config FNOSC = FRCDIV //Sets clock to 8 MHZ oscillator with post scaler
int main(void) {
        _RCDIV = 0b001; //sets a 2 post scaler
        /*Pin 12 is RB8*/
        /*Pin 13 is RB9*/
        /*Reset All Registers We will be Using*/
        IC2CON1 = 0;
        IC2CON2 = 0;
        T2CON = 0;
        LATA = 0;
        LATB = 0;
        TRISA = 0;
        TRISB = 0;
        ANSA = 0;
        ANSB = 0;
        /*Configure Pin 12 as a Digital Output*/
        _TRISB8 = 0; //sets pin 12 as a digital output
        _LATB8 = 0; //sets pin 12 output to low
        /*Configure Pin 13 as a Digital Input*/
        _TRISB9 = 1; //sets pin 13 as a digital input
        /*Configure Input Capture 2 Module Clock*/
        IC2CON1bits.IC2TSEL = 0b111; //sets IC2 module clock to the system clock
        /*Configure Input Capture 2 Module Capture Mode*/
        IC2CON1bits.ICM = 0b001; //sets the IC2 module mode to capture every edge
        /*Configure Input Capture 2 Module in Software Triggered Operation Mode*/
        IC2CON2bits.ICTRIG = 1; //sets the IC2 module to trigger mode
        IC2CON2bits.SYNCSEL = 0b00000; //sets the IC2 module as software triggered
        /*Set Input Capture 2 Module Interrupt Rate*/
        IC2CON1bits.ICI = 0b00; //sets interrupt to occur on every capture event
        /*Configure the Input Capture 2 Module Interrupt*/
        _IC2IP = 4; //sets the IC2 interrupt priority to 4
        _IC2IF = 0; //clears the IC2 interrupt flag
        _IC2IE = 1; //enables the IC2 interrupt
        /*Configure Timer 2*/
        T2CONbits.TCS = 0; //sets timer 2 clock source to system clock
        T2CONbits.TCKPS = 0b00 //sets a prescaler value of 1
        T2CONbits.T32 = 0; //unlinks timers 2 and 3
        PR2 = 20 //sets the period to 10uS
        TMR2 = 0; //resets the timer 2 value
        /*Configure Timer 2 Interrupt*/
        _T2IP = 4; //sets the timer 2 interrupt priority to 4
        _T2IF = 0; //clears the timer 2 interrupt flag
        _T2IE = 1; //enables the timer 2 interrupt
        /*Turn on Timer 2*/
        T2CONbits.TON = 1; //turns on timer 2
        /*Let the Interrupts Control the Rest of the Code*/
        while(1);
        return 0;
}
```