EEE2008: Buggy Project

PIC16F18326 and MPLab X: Getting Started Guide

Authors: Ben Stainthorpe (original), Connor Bramwell (updated version 2025/26)

Module Leader: Andrew Smith

This guide is designed to help get started with using 'MPLab X' for code development and programming of the PIC device. The MPLab code generator MCC will be used to generate low level peripheral code. An example program and guide to debugging the code is also shown.

Contents

Introduction

This section provides an explanation of PICs and how they work. **Skim over this section to gain some background knowledge, the tutorial begins at 'Creating a Project on MPLab X'.**

This introduction may go into more detail than necessary as we will be using the MCC content manager wizard to generate most of our code. However, to gain a more in depth understanding of embedded programming, it will be useful to read.

What is a PIC?

PIC originally stood for 'Peripheral Interface Controller'. But now the name has changed to 'Programmable Intelligent Computer', as the device range has developed, and they are capable of more tasks. Essentially, it is a family of microcontrollers, not microprocessors.

Micro<u>processor</u> – contains only the computational part (the calculator). Does not contain memory, or any peripherals ADCs/GPIOs. They are used to construct microcontrollers.

Micro<u>controller</u> – contains a processor, reprogrammable memory storage, and peripherals for ADCs, DACs, GPIOs, Timers, PWM modules etc. It can control electronic devices such as DC/DC converters, Inverters etc.

Why are there so many different PICs?

There are a wide range of PICs because they can cater to a wide range of applications, and in industrial applications, 'every penny counts'. Therefore, manufacturers do not want to purchase a fancy expensive PIC with hundreds of I/Os when they only need 1 PWM and a few ADCs, and a high computational speed is not required. Therefore, they will find a device based on spec best suited for the application – at the cheapest price.

PIC Peripherals

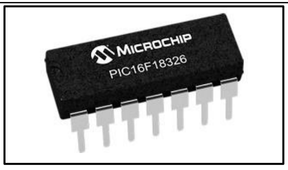
Several pins on a PIC are reserved for power supply, and the rest are available as General Purpose Input/Output (GPIO) pins. All I/O pins are capable of acting as a digital in or digital out. This means it can perform digital tasks such as reading a switch status or turning on/off an LED. However, more specialist tasks such as PWM out, ADC reads, or DAC outputs, are only available on specific pins, as the module has been wired internally onto those pins. In the code, we have to specify what this pin is assigned to do when setting it up. For example, if we want it to act as an ADC, we need to specify it is acting as an input, and then specify that it will act as an ADC (not a digital input). It will then configure the on-chip semiconductors (switches) to link the pin to the ADC module, rather than the digital module. We do this by setting up/writing to 'registers'. These are memory locations on the PIC which contain all settings for the PIC.

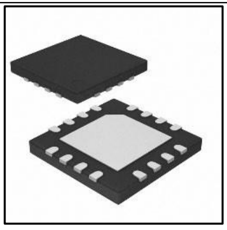
By default, the pins will 'float'; they are not tied to any voltage internally. However, when configured as a GPIO digital out, the PIC will pull the pin to either ground or V_{DD} depending on its required state. When configured as an input, it's voltage is under full control of the external circuitry and will act as a high impedance (resistance) load.

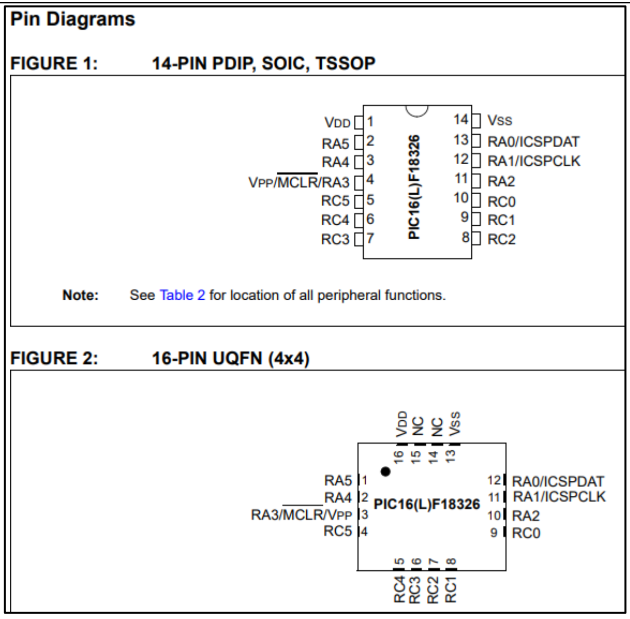
The I/Os are split into several different 'ports', these are groups of pins which are read/written to by the microcontroller at the same time when acting as digital inputs/outputs. Typically they are assigned a letter; for example, port A which contains pins PA0 up to PA7, port B which contains PB0 up to PB7, etc.

Package Types

Each PIC often comes in a variety of shapes and sizes. The standard 'package' is the 'Dual In Line' DIP (or PDIP, p for plastic), this is what we have for this project. It contains large protruding legs which can sit into through-hole chip holders, soldered onto the board. This is ideal for prototyping and projects where the risk of damaging the microcontroller during experiments is high, and may need to be swapped out. However, in industrial applications, a surface mount package is often used, which takes up less space and is cheaper to mount onto a PCB (through hole soldering is more expensive).







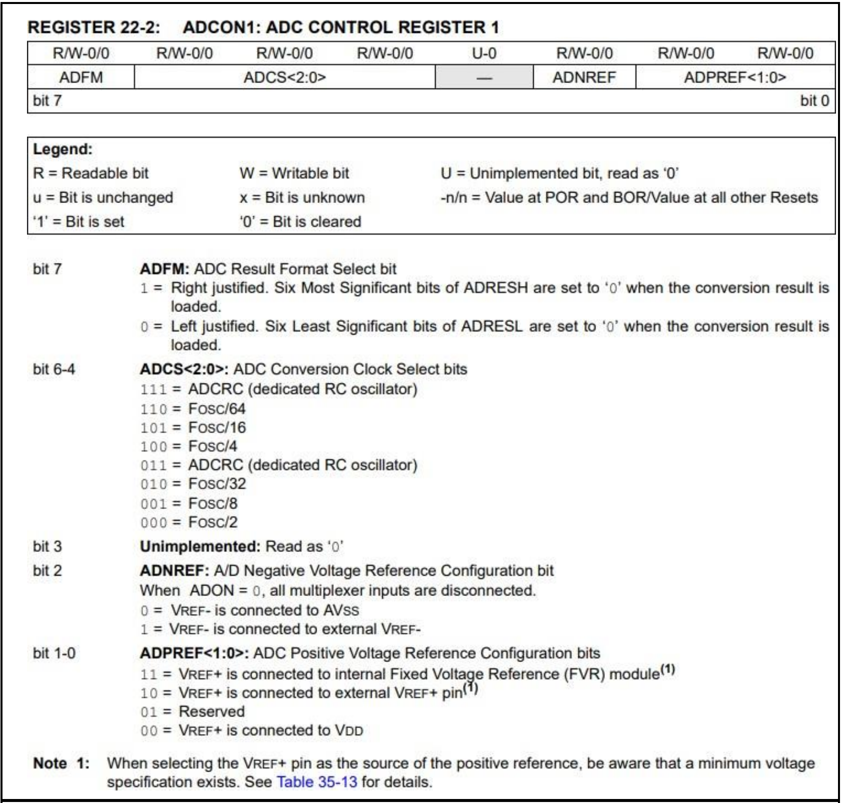
DIP Package PIC *UQFN Package* *Pinouts of each package*

Figure 1: Device Packages

Note that for the same PIC model, different packages may have more/less pins, therefore it is important to read the datasheet carefully. Our DIP package has 14 pins, whereas the UQFN equivalent for the same PIC16F18326 has 16 pins.

Datasheet and Application Notes

When programming any microcontroller, the first place to look is the datasheet. This contains information of all the pins, peripherals, capabilities, and registers which must be set to enable a certain function. The application notes go alongside the datasheet to provide a guide as to how to setup a peripheral, for example an ADC, they contain code examples and experiment setups.



REGISTER 22-2: ADCON1: ADC CONTROL REGISTER 1

R/W-0/0	R/W-0/0	R/W-0/0	R/W-0/0	U-0	R/W-0/0	R/W-0/0	R/W-0/0
ADFM	ADCS<2:0>			—	ADNREF	ADPREF<1:0>	
bit 7							bit 0

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
u = Bit is unchanged	x = Bit is unknown	-n/n = Value at POR and BOR/Value at all other Resets
'1' = Bit is set	'0' = Bit is cleared	

bit 7 **ADFM:** ADC Result Format Select bit
1 = Right justified. Six Most Significant bits of ADRESH are set to '0' when the conversion result is loaded.
0 = Left justified. Six Least Significant bits of ADRESL are set to '0' when the conversion result is loaded.

bit 6-4 **ADCS<2:0>:** ADC Conversion Clock Select bits
111 = ADCRC (dedicated RC oscillator)
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = ADCRC (dedicated RC oscillator)
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2

bit 3 **Unimplemented:** Read as '0'

bit 2 **ADNREF:** A/D Negative Voltage Reference Configuration bit
When ADON = 0, all multiplexer inputs are disconnected.
0 = VREF- is connected to AVSS
1 = VREF- is connected to external VREF-

bit 1-0 **ADPREF<1:0>:** ADC Positive Voltage Reference Configuration bits
11 = VREF+ is connected to internal Fixed Voltage Reference (FVR) module(1)
10 = VREF+ is connected to external VREF+ pin(1)
01 = Reserved
00 = VREF+ is connected to VDD

Note 1: When selecting the VREF+ pin as the source of the positive reference, be aware that a minimum voltage specification exists. See Table 35-13 for details.

Figure 2: PIC Register Example: ADCON1

PIC Registers

As mentioned, 'registers' are memory locations on the PIC which contain settings for things such as the Clock speed, the peripherals, and many other important settings. **In this lab we will use MPLab to deal with the low-level setup of these registers so we don't need to worry about understanding them fully**, but it is important to know vaguely what is going on. For this PIC, they are 8 bits long. To the right is an example register for the ADC module; called ADCON1. It may seem complicated at first, but breaking it down, it is simple. The bits are numbered from 0 to 7, therefore there are 8 in total. The rightmost bit 'bit 7' is called ADFM. This will control how the ADC data is formatted after being read. The next 3 bits, 6-4, are called the ADCS bits, and control the clock rate to the ADC module (how fast data is processed in the ADC module). Setting them to '110' causes a read rate of Fosc/64, aka the processor clock speed divided by 64. Bit 3 is unused. Bits 2, 1 and 0 control the voltage reference to the ADC module.

Whenever we set up a module, we can either select which individual bit we want to modify in the register, or modify all 8 bits in the register at once.

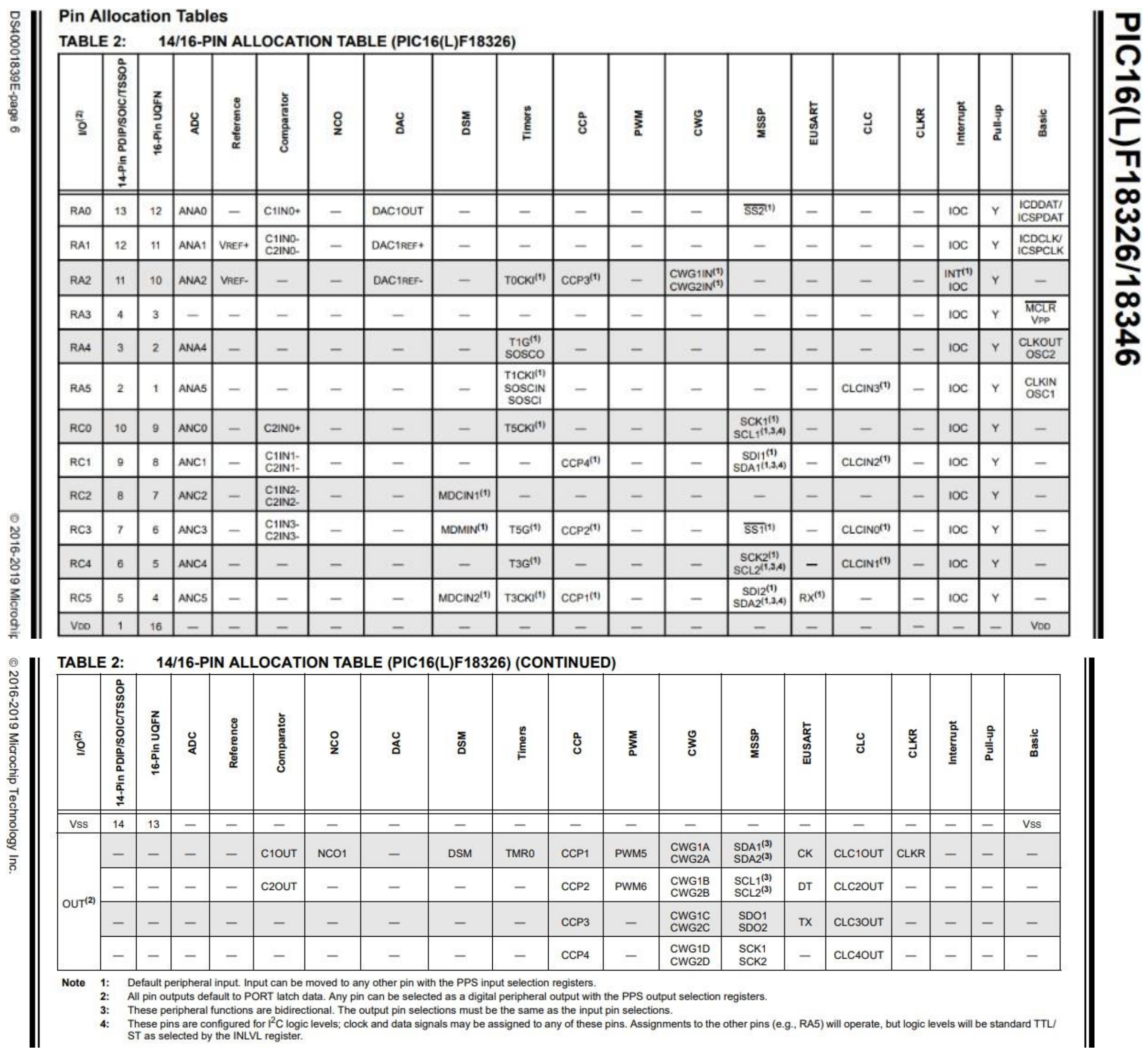
About our PIC

Our PIC is the PIC16F18326. The '16' refers to the family of the PIC, in this case it is mid range, and is related to the program instruction size; the larger, the more powerful the PIC. The F refers to the fact it's memory is flash based rather than EEPROM or older memory types.

The Table below from the datasheet shows the capability of each pin on the PIC. Our package is the PDIP, so we need to read the second column to find the pin number of each I/O. For example, if we identify that we want to use RA2, because it has the capability to have an ADC on this pin (see the ANA2 under the ADC column), we would setup the registers to use RA2 and ANA2, and then use this table to find that the pin number of this ADC input is 11, for our DIP.

The Datasheet for our PIC can be found at the below link:

https://ww1.microchip.com/downloads/aemDocuments/documents/MCU08/ProductDocuments/DataSheets/40001839E.pdf



DS40001839E-page 6

PIC16(L)F18326/18346

Pin Allocation Tables

TABLE 2: 14/16-PIN ALLOCATION TABLE (PIC16(L)F18326)

I/O$^{(2)}$	14-Pin PDIP/SOIC/TSSOP	16-Pin UQFN	ADC	Reference	Comparator	NCO	DAC	DSM	Timers	CCP	PWM	CWG	MSSP	EUSART	CLC	CLKR	Interrupt	Pull-up	Basic
RA0	13	12	ANA0	—	C1IN0+	—	DAC1OUT	—	—	—	—	—	$\overline{SS2}^{(1)}$	—	—	—	IOC	Y	ICDDAT/ ICSPDAT
RA1	12	11	ANA1	VREF+	C1IN0- C2IN0-	—	DAC1REF+	—	—	—	—	—	—	—	—	—	IOC	Y	ICDCLK/ ICSPCLK
RA2	11	10	ANA2	VREF-	—	—	DAC1REF-	—	T0CKI$^{(1)}$	CCP3$^{(1)}$	—	CWG1IN$^{(1)}$ CWG2IN$^{(1)}$	—	—	—	—	INT$^{(1)}$ IOC	Y	—
RA3	4	3	—	—	—	—	—	—	—	—	—	—	—	—	—	—	IOC	Y	$\overline{MCLR}$ VPP
RA4	3	2	ANA4	—	—	—	—	—	T1G$^{(1)}$ SOSCO	—	—	—	—	—	—	—	IOC	Y	CLKOUT OSC2
RA5	2	1	ANA5	—	—	—	—	—	T1CKI$^{(1)}$ SOSCIN SOSCI	—	—	—	—	—	CLCIN3$^{(1)}$	—	IOC	Y	CLKIN OSC1
RC0	10	9	ANC0	—	C2IN0+	—	—	—	T5CKI$^{(1)}$	—	—	—	SCK1$^{(1)}$ SCL1$^{(1,3,4)}$	—	—	—	IOC	Y	—
RC1	9	8	ANC1	—	C1IN1- C2IN1-	—	—	—	—	CCP4$^{(1)}$	—	—	SDI1$^{(1)}$ SDA1$^{(1,3,4)}$	—	CLCIN2$^{(1)}$	—	IOC	Y	—
RC2	8	7	ANC2	—	C1IN2- C2IN2-	—	—	MDCIN1$^{(1)}$	—	—	—	—	—	—	—	—	IOC	Y	—
RC3	7	6	ANC3	—	C1IN3- C2IN3-	—	—	MDMIN$^{(1)}$	T5G$^{(1)}$	CCP2$^{(1)}$	—	—	$\overline{SS1}^{(1)}$	—	CLCIN0$^{(1)}$	—	IOC	Y	—
RC4	6	5	ANC4	—	—	—	—	—	T3G$^{(1)}$	—	—	—	SCK2$^{(1)}$ SCL2$^{(1,3,4)}$	—	CLCIN1$^{(1)}$	—	IOC	Y	—
RC5	5	4	ANC5	—	—	—	—	MDCIN2$^{(1)}$	T3CKI$^{(1)}$	CCP1$^{(1)}$	—	—	SDI2$^{(1)}$ SDA2$^{(1,3,4)}$	RX$^{(1)}$	—	—	IOC	Y	—
VDD	1	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	VDD

TABLE 2: 14/16-PIN ALLOCATION TABLE (PIC16(L)F18326) (CONTINUED)

I/O$^{(2)}$	14-Pin PDIP/SOIC/TSSOP	16-Pin UQFN	ADC	Reference	Comparator	NCO	DAC	DSM	Timers	CCP	PWM	CWG	MSSP	EUSART	CLC	CLKR	Interrupt	Pull-up	Basic
VSS	14	13	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	VSS
OUT$^{(2)}$	—	—	—	—	C1OUT	NCO1	—	DSM	TMR0	CCP1	PWM5	CWG1A CWG2A	SDA1$^{(3)}$ SDA2$^{(3)}$	CK	CLC1OUT	CLKR	—	—	—
	—	—	—	—	C2OUT	—	—	—	—	CCP2	PWM6	CWG1B CWG2B	SCL1$^{(3)}$ SCL2$^{(3)}$	DT	CLC2OUT	—	—	—	—
	—	—	—	—	—	—	—	—	—	CCP3	—	CWG1C CWG2C	SDO1 SDO2	TX	CLC3OUT	—	—	—	—
	—	—	—	—	—	—	—	—	—	CCP4	—	CWG1D CWG2D	SCK1 SCK2	—	CLC4OUT	—	—	—	—

Note 1: Default peripheral input. Input can be moved to any other pin with the PPS input selection registers.
2: All pin outputs default to PORT latch data. Any pin can be selected as a digital peripheral output with the PPS output selection registers.
3: These peripheral functions are bidirectional. The output pin selections must be the same as the input pin selections.
4: These pins are configured for I^{2}C logic levels; clock and data signals may be assigned to any of these pins. Assignments to the other pins (e.g., RA5) will operate, but logic levels will be standard TTL/ST as selected by the INLVL register.

Figure 3: Pin Out Allocation of PIC16F18326

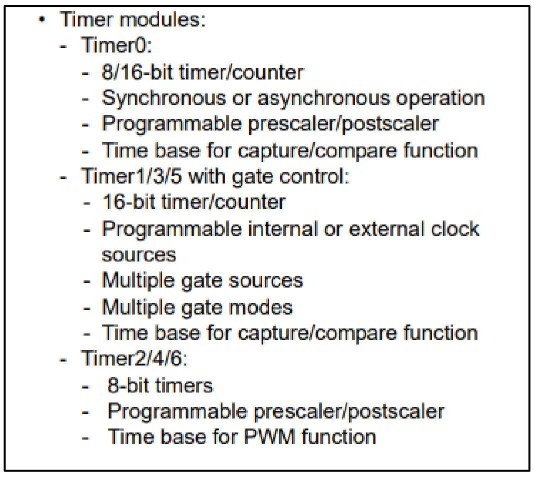
MPLab X

MPLab X is an IDE (integrated development environment). This means it can be used to write code, compile the code, download it onto the PIC and debug the code onboard the PIC. Almost everything we need to do PIC-wise can be done on this software.

The compiler we are using is the XC8 compiler. This contains libraries of functions specific to the 8 bit processor based PIC family, which contains our PIC16F18326.

Timer Modules

The PIC features multiple timer modules with varying capabilities. Timers effectively just count to a maximum possible number, incremented each time it receives a clock pulse. The main system clock can be divided down to a lower frequency for the timer, so they do not need to count at the same high frequency as the main processor clock. The timers with large bit sizes (such as the 16 bit timer) can count to a larger number than the 8 bit timer. This allows for more precision in counting. When reaching the maximum number, they reset to 0, however a register will flag that a reset has occurred. This can trigger an interrupt routine to execute some code, for a function which must be executed at a regular rate, such as a PID controller which requires constant 'dt'. This is the basis for 'Time Triggered Scheduling'.



- Timer modules:
  - Timer0:
    - 8/16-bit timer/counter
    - Synchronous or asynchronous operation
    - Programmable prescaler/postscaler
    - Time base for capture/compare function
  - Timer1/3/5 with gate control:
    - 16-bit timer/counter
    - Programmable internal or external clock sources
    - Multiple gate sources
    - Multiple gate modes
    - Time base for capture/compare function
  - Timer2/4/6:
    - 8-bit timers
    - Programmable prescaler/postscaler
    - Time base for PWM function

Figure 4: Timer Module Capabilities

Reading ADCs on a microcontroller

As we know, an ADC (analogue to digital converter) reads a voltage in the circuit and uses it to create a digital representation of this voltage. The PIC operates at 5V supply, so the ADC will only be able to measure a voltage between 0-5V. So if we're measuring a large voltage, we need to scale it down using a potential divider or op amp.

Once scaled down, the ADC converts this value to a number, between 0 and the maximum value of the number of bits the ADC works at. For the ADC module in this PIC, it works at 10 bits. Therefore, the max value is 2^10 = 1024. Essentially, this means that 0V on the ADC pin gives a digital value of 0, 2.5V gives 512, and 5V gives 1024.

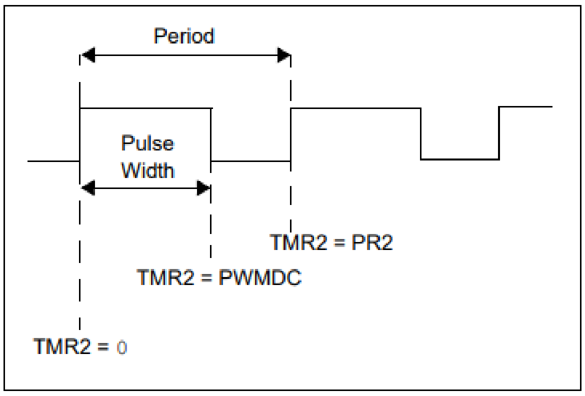
PWM Generation

PWMs (pulse width modulation; a varying ratio square wave) can work in various modes on a microcontroller. The first is using the dedicated PWM modules. On this PIC, there are two dedicated 10-bit PWM generators, PWM 5 and PWM 6. The output of these can be mapped onto any pin. The 10 bit represents the maximum value for the duty cycle control register. As they are 10 bits, more precision can be achieved with the duty cycle than a lower bit size PWM generator. The maximum value for the duty cycle register is 2^10 = 1024. The timer source, timer module 2, 4 or 6, is only 8 bits, meaning the maximum count value is 2^8 = 256.

The PWM module period is solely controlled via the associated timer; the 8 bit PRx value, the clock frequency, and the prescaler for the timer. The x in PRx corresponds with the number of the timer, so for timer 2 it is PR2. The larger the PRx value, the larger the period and thus lower the frequency. With PRx set to maximum at 256, we get the maximum possible period, for the current prescaler value. If we change the prescaler, we can get a larger period.

The duty cycle period is determined by the PWMxDC register, and also the PRx value, prescaler and clock frequency. The x being the PWM module, either 5 or 6. So PWM5DC for PWM module 5. As the Duty Cycle registers are 10 bits, the max count is 1024. This corresponds to 100% duty cycle when PRx is maximum at 256, and therefore 512 on PWMxDC would correspond to 50% duty cycle. When PRx is set to 128, 512 on PWMxDC would represent 100% duty, and 256 would be 50% duty. Therefore, the duty cycle register must be altered considering the value in the timer period register.

FIGURE 18-1: PWM OUTPUT



EQUATION 18-1: PWM PERIOD

$$PWM\ Period = [(PR2) + 1] \bullet 4 \bullet T_{OSC} \bullet (TMR2\ Prescale\ Value)$$

Note: TOSC = 1/FOSC

EQUATION 18-2: PULSE WIDTH

$$Pulse\ Width = (PWMxDC) \bullet T_{OSC} \bullet (TMR2\ Prescale\ Value)$$

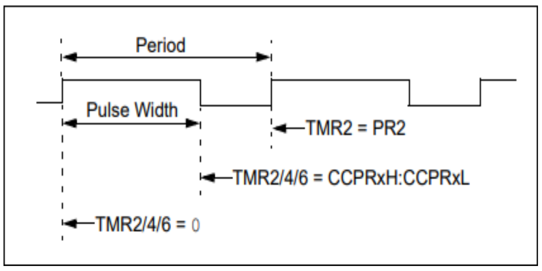
EQUATION 18-3: DUTY CYCLE RATIO

$$Duty\ Cycle\ Ratio = \frac{(PWMxDC)}{4(PR2 + 1)}$$

Figure 5: PWM Operation using PWM Module, from the datasheet

The other option is using the Capture Compare Modules. This PIC features 4x CCP modules. These are multipurpose modules which can be used for counting when in capture mode, using the timers. However, they can also be used to generate a PWM too, again with 10-bit resolution. The operation is very similar to the PWM module. Where PRx of the timer controls the period, the capture compare value CCPRx (x representing the capture compare module number) controls the duty cycle. As it is 10 bits, it must be stored on two 8 bit registers, with 6 bits on one not used. This is why the datasheet mentions CCPRx H and L; H for High, L for low. Where H is the upper 2 bits, L is the lower 8 bits, of the 10 bit value.

FIGURE 29-3: CCP PWM OUTPUT SIGNAL



EQUATION 29-1: PWM PERIOD

$$PWM\ Period = [(PR2x) + 1] \bullet 4 \bullet T_{OSC} \bullet (TMR2/4/6\ Prescale\ Value)$$

Note: TOSC = 1/FOSC

EQUATION 29-2: PULSE WIDTH

$$Pulse\ Width = (CCPRxH{:}CCPRxL\ register\ pair) \bullet T_{OSC} \bullet (TMR2\ Prescale\ Value)$$

EQUATION 29-3: DUTY CYCLE RATIO

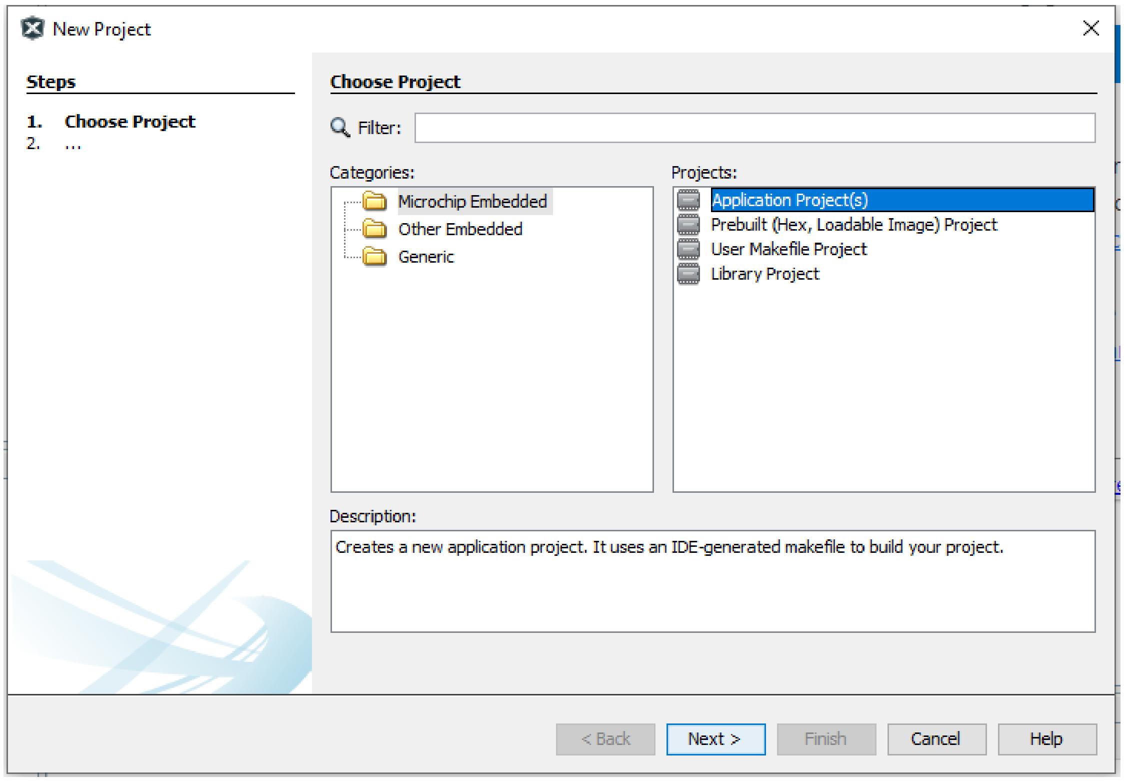
$$Duty\ Cycle\ Ratio = \frac{(CCPRxH{:}CCPRxL\ register\ pair)}{4(PR2 + 1)}$$

Figure 6: PWM Operation using the Capture Compare Module, from the Datasheet

Creating a Project On MPLab X

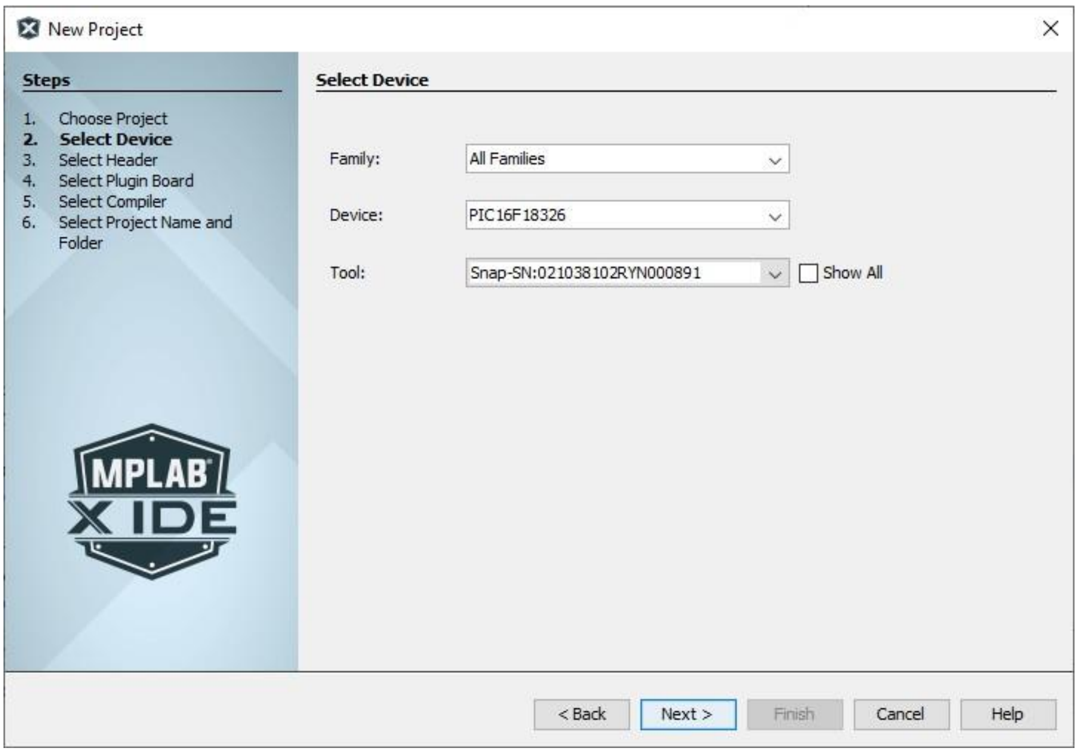
We will now look at creating a project. The first step as you would imagine, is to open MPLab X IDE. **Make sure it is the grey IDE and not the yellow IPE**.

1. Click File -> New Project
2. Click 'application project(s)', as in the image below, then press next.
3. Click on the 'Device' field; and begin typing in the name of our PIC; **PIC16F18326**. This should then appear in the drop-down list as an option. Click on this when it appears.
4. Make sure the programming device is connected to the computer. If using your grey programming device**, select the tool as 'Snap-SN'** which should be listed. If in the electronics labs and using the PICkit, select the tool as the 'PICkit 4-SN', then press next. If no device shows, recheck the device is connected, then ask a demonstrator.
5. Click on the **XC8** dropdown, then the option beneath it containing its location, and press next
6. Finally, choose a name for your project and type it in, **before changing the project location to your H: network drive**. This will allow you to access it on other computers (use File > Open Project and locate the folder from another computer).



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

Figure 7: New Project Step 2


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

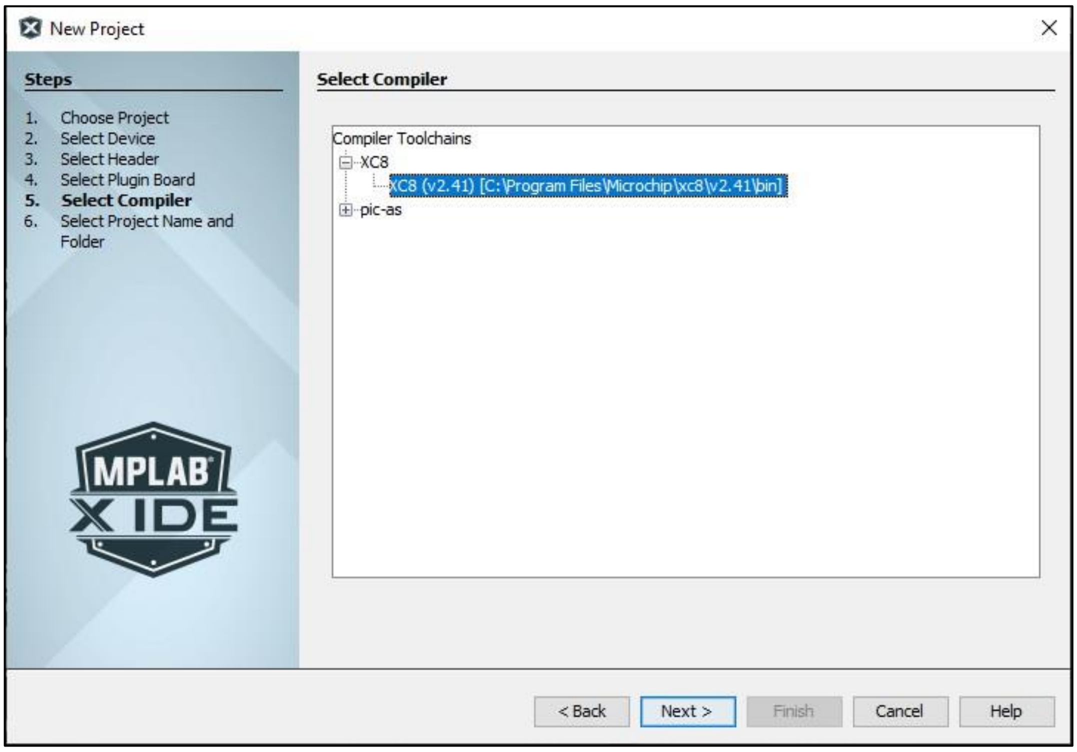
Select Device

Family: All Families

Device: PIC16F18326

Tool: Snap-SN:021038102RYN000891

Figure 8: New Project Step 3 and 4


New Project

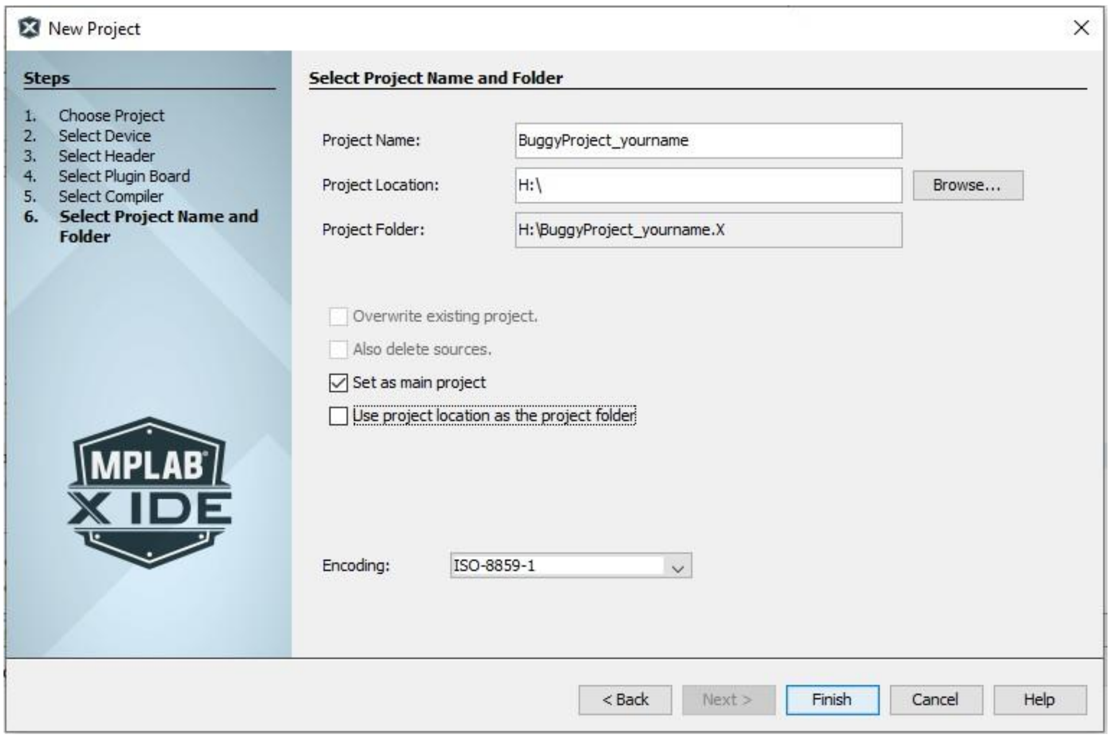
Steps

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
- XC8
  - XC8 (v2.41) [C:\Program Files\Microchip\xc8\v2.41\bin]
- pic-as

Figure 9: New Project Step 5


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: BuggyProject_yourname
Project Location: H:\
Project Folder: H:\BuggyProject_yourname.X

- [ ] Overwrite existing project.
- [ ] Also delete sources.
- [x] Set as main project
- [ ] Use project location as the project folder

Encoding: ISO-8859-1

Figure 10: New Project Step 6

You should now have a project created, with several folders within it created. The main folders we are concerned with is 'Header Files' and 'Source Files'. As you may imagine, the header files folder should contain any .h files you wish to create, and the Source files folder should contain all .c files, including main.c. The 'important files' folder contains the makefile which you may be familiar with from your C Programming University modules. This is generated and modified by MPLab, so don't touch it – it is complicated as it must deal with PIC specific settings.

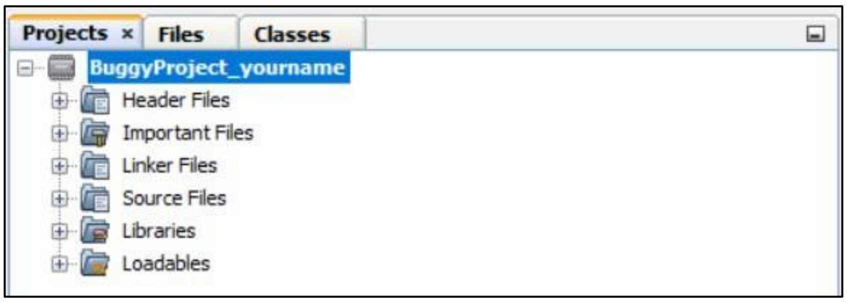


Figure 11: Project Folder

Using the MCC Content Manager Wizard

At this stage, we could begin writing all of our code from scratch. Typically, **(don't do this!)** we would create a main file by clicking file -> new file, then the C folder, and C Main file. MPLab will tell the compiler to look at this file for the 'main' function; the program entry point. We can create other C files and H files using the C Source File or C Header file option instead of C Main File.

However, to speed things up, we will use a Wizard.

1. Click on the Blue MCC icon on the top toolbar
2. After a loading period, a new MCC Content Manager window will appear. Click 'select MCC classic'. If a box opens which says MCC needs to install new packages, press 'finish' at the top. If MCC classic is selected you may have to refer to an older version of this document, on many of the PCs it will automatically start the new version without giving a choice.
3. We now have a complicated interface with lots of buttons, we will need to break this down

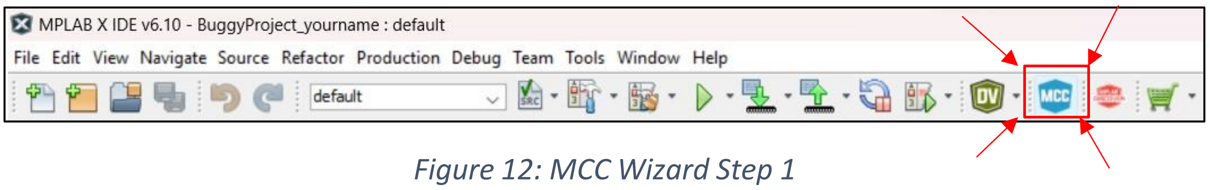
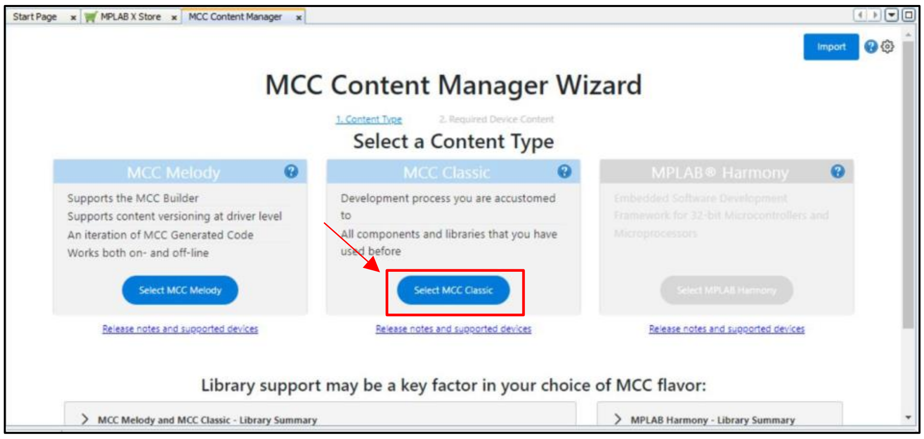

MPLAB X IDE v6.10 - BuggyProject_yourname : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default
MCC

Figure 12: MCC Wizard Step 1


Start Page | MPLAB X Store | MCC Content Manager

Import

MCC Content Manager Wizard

1. Content Type   2. Required Device Content

Select a Content Type

MCC Melody
Supports the MCC Builder
Supports content versioning at driver level
An iteration of MCC Generated Code
Works both on- and off-line
Select MCC Melody
Release notes and supported devices

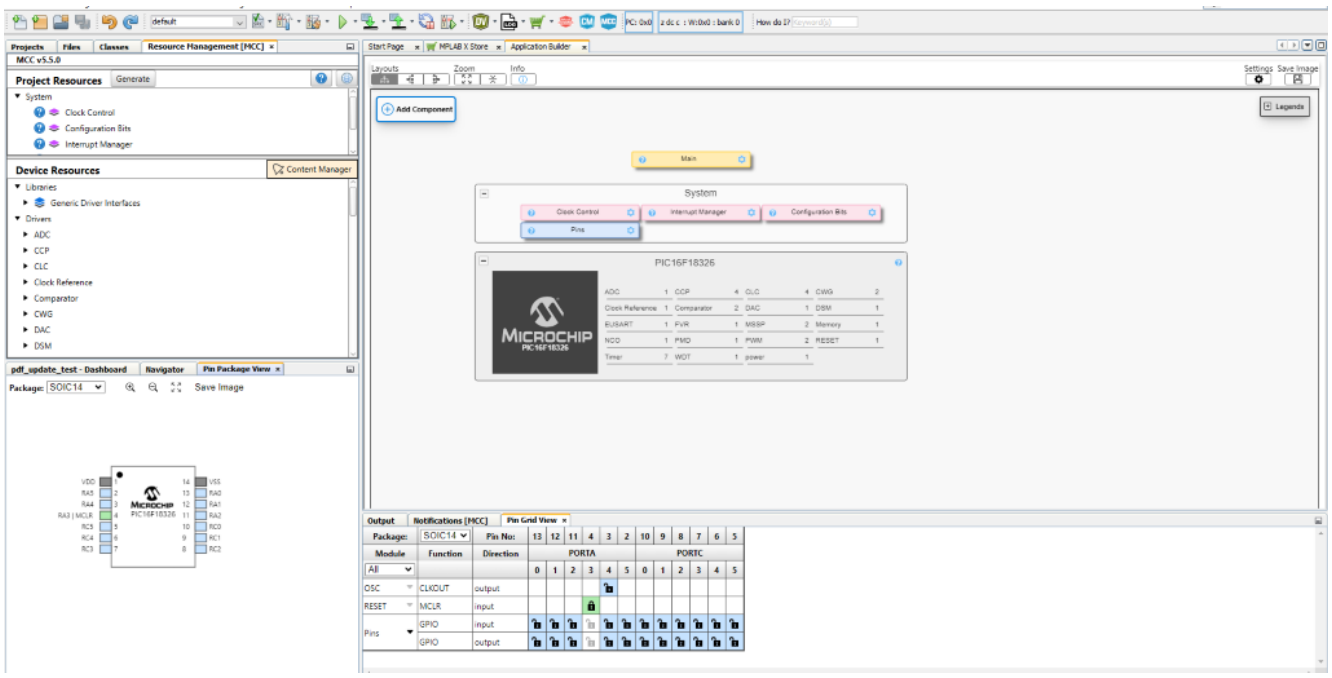
MCC Classic
Development process you are accustomed to
All components and libraries that you have used before
Select MCC Classic
Release notes and supported devices

MPLAB® Harmony
Embedded Software Development Framework for 32-bit Microcontrollers and Microprocessors
Select MPLAB Harmony
Release notes and supported devices

Library support may be a key factor in your choice of MCC flavor:

> MCC Melody and MCC Classic - Library Summary
> MPLAB Harmony - Library Summary

Figure 13: MCC Wizard Step 2



Package:	SOIC14	Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
Module	Function	Direction	PORTA						PORTC					
All			0	1	2	3	4	5	0	1	2	3	4	5
OSC	CLKOUT	output					🔓							
RESET	MCLR	input				🔒								
Pins	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓

Figure 14: MCC Wizard Interface

On the bottom left of the user interface we have the Pin Manager: Package View, with a diagram of our chip. Grey are the pre-defined pins – for the power supply. Blue are the free unassigned pins. Green are the pins which have been assigned. Currently, MCLR is the only one set. This is used by the programmer to tell the PIC to prepare for programming.

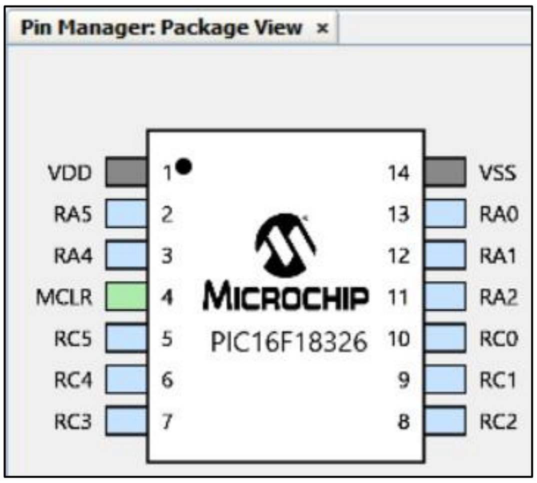
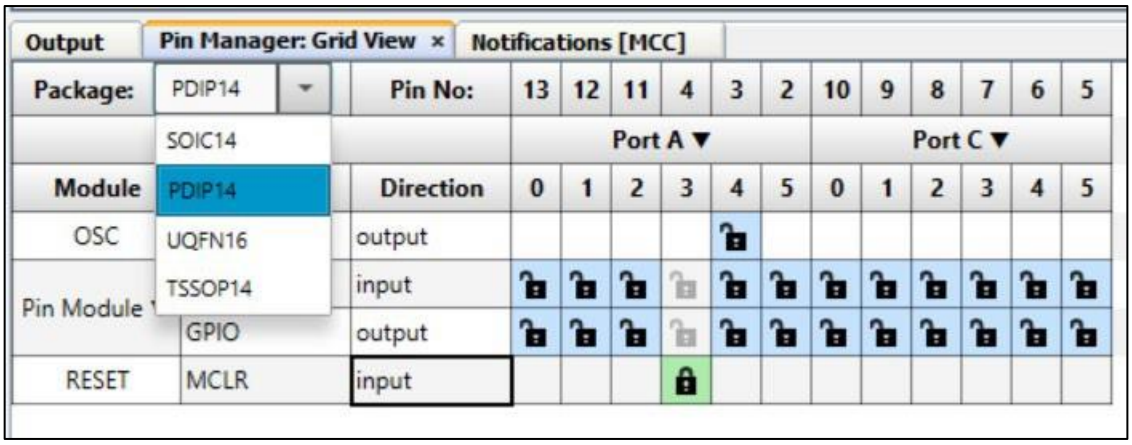


Figure 15: Pin Manager, Package View

At the bottom we have the Pin Manager: Grid View, to setup the pins.

1. Click on the dropdown next to ‘Package:’ to **change to our PDIP14 package**.



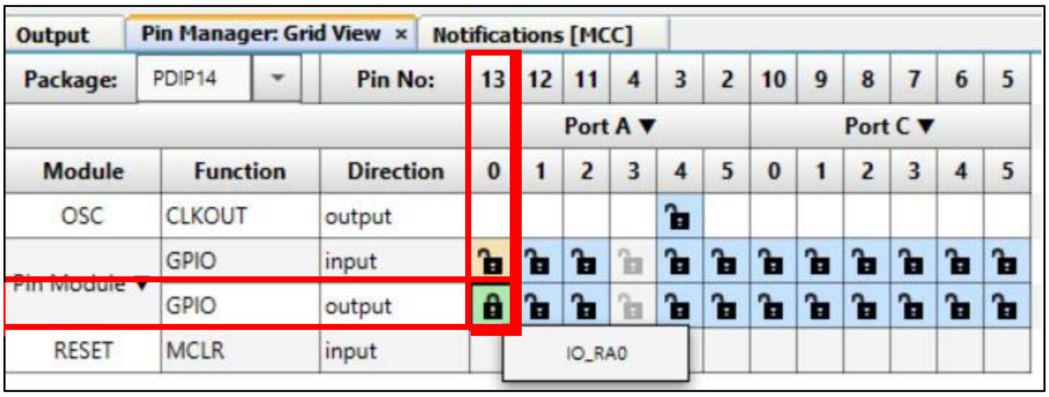
Package:	PDIP14	Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
			Port A ▼						Port C ▼					
Module		Direction	0	1	2	3	4	5	0	1	2	3	4	5
OSC		output					🔓							
Pin Module	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								

Figure 16: Pin Manager, Grid View

Setup a GPIO as a Digital Input/Output

Setting up a GPIO is rather simple. Let's say we want a digital output on RA0, which we can see is assigned to pin 13.

1. On the Pin Manager: Grid View, locate the pin 13/RA0 column
2. Locate the row, with the first 3 columns Pin Module (may be called pins in some versions) – GPIO – output.
3. Follow this row along until we reach the column under RA0. Now **Click on the Blue Unlocked Padlock icon**.
4. This symbol should turn green and locked, and the symbol above, corresponding to the same pin set to an input instead of an output, will turn orange. If we wanted this pin to be an input, these would be the opposite.
5. Finally, go to the Resource Management tab, then under Project Resources, then system, **we see the 'Pins' settings. Double click on this.**
6. Here, we can see the options for our RA0 digital out. 'Output' should already be ticked, since we selected the output lock earlier**. 'Analog' will need to be unticked, as we do not wish to use it as an Analog pin (ADC or DAC)**. Start high can be ticked if we wish to have the GPIO high on initialisation of the code. WPU is weak pull up, OD is open drain (sinks current only, does not source), IOC is interrupt on change. We do not need to worry about these final 3. **For now, leave 'Output' (and if given the option 'Slew Rate' and 'Input Level Control') as the only one ticked. Output may be forced in some versions, this is fine.**

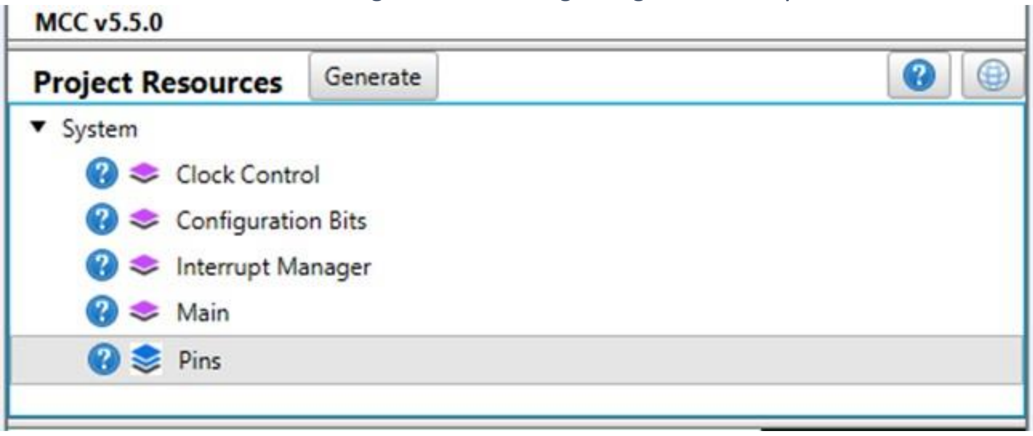


Output | Pin Manager: Grid View × | Notifications [MCC]

Package: PDIP14		Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
OSC	CLKOUT	output					🔓							
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input												

IO_RA0

Figure 17: Setting a Digital I/O Steps 1-4

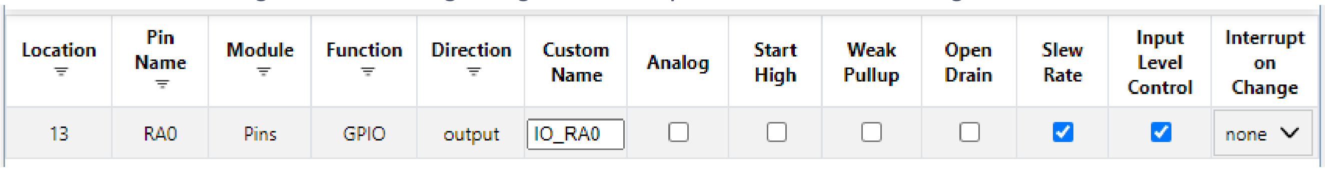


MCC v5.5.0

Project Resources | Generate

- System
  - Clock Control
  - Configuration Bits
  - Interrupt Manager
  - Main
  - Pins

Figure 18: Setting a Digital I/O Step 5; Pin Module settings



Location	Pin Name	Module	Function	Direction	Custom Name	Analog	Start High	Weak Pullup	Open Drain	Slew Rate	Input Level Control	Interrupt on Change
13	RA0	Pins	GPIO	output	IO_RA0	☐	☐	☐	☐	✓	✓	none

Figure 19: Setting A Digital I/O Step 6; Pin Module settings

Setup the System Module Settings

We will now look at the system level settings. These are critical settings which need to be considered for operation of the PIC.

1. Under Resource Management > Project Resources on the left, under system, double click on the system module option
2. In the new window, **change the oscillator to 'HFINTOSC', then HF internal clock to 12MHz, and the clock divider to 4**. This results in a clock frequency of 3MHz. It would be complicated to boost this further for our application– see if you can figure out why!

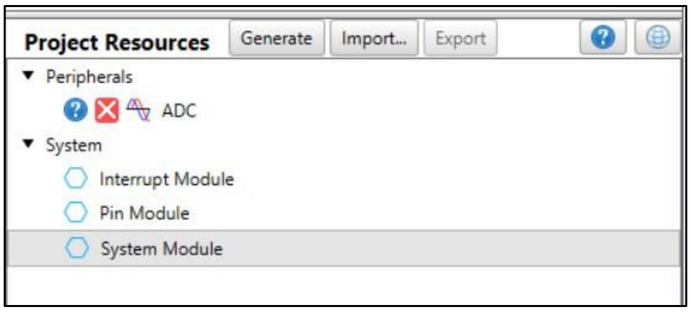
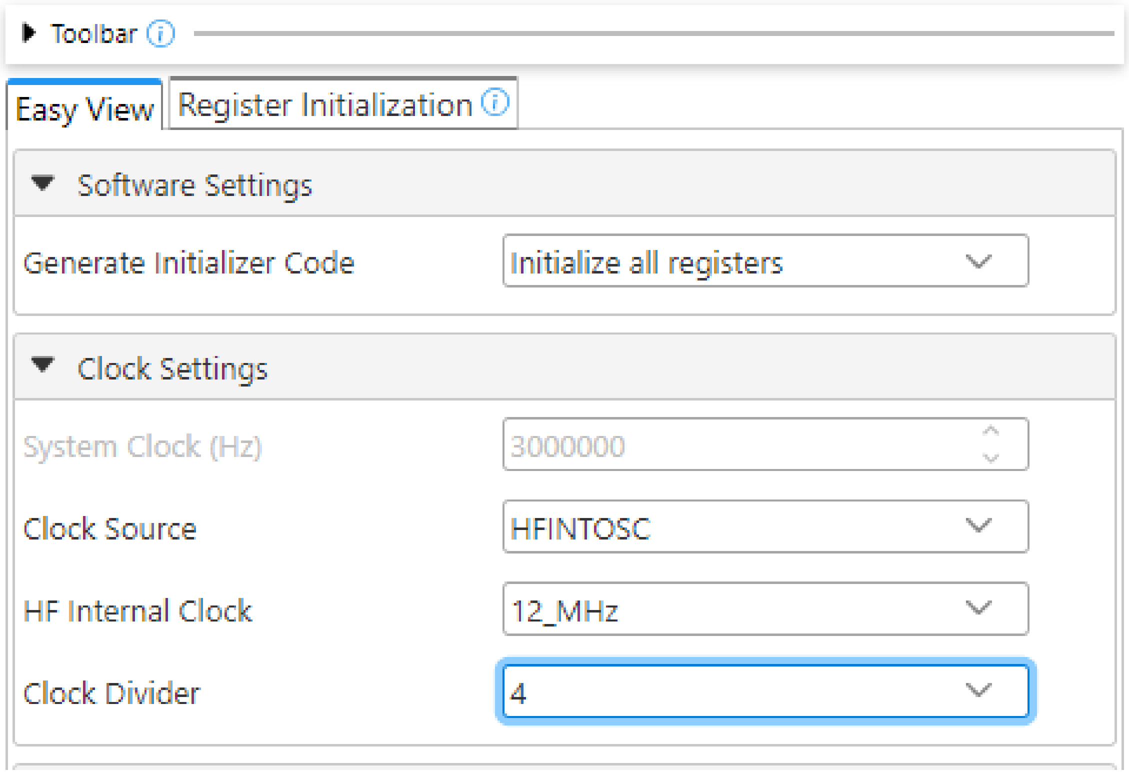


Figure 20: System Module Setup step 1



Generate Initializer Code: Initialize all registers
System Clock (Hz): 3000000
Clock Source: HFINTOSC
HF Internal Clock: 12_MHz
Clock Divider: 4

Figure 21: System Module Setup Step 2; oscillator settings

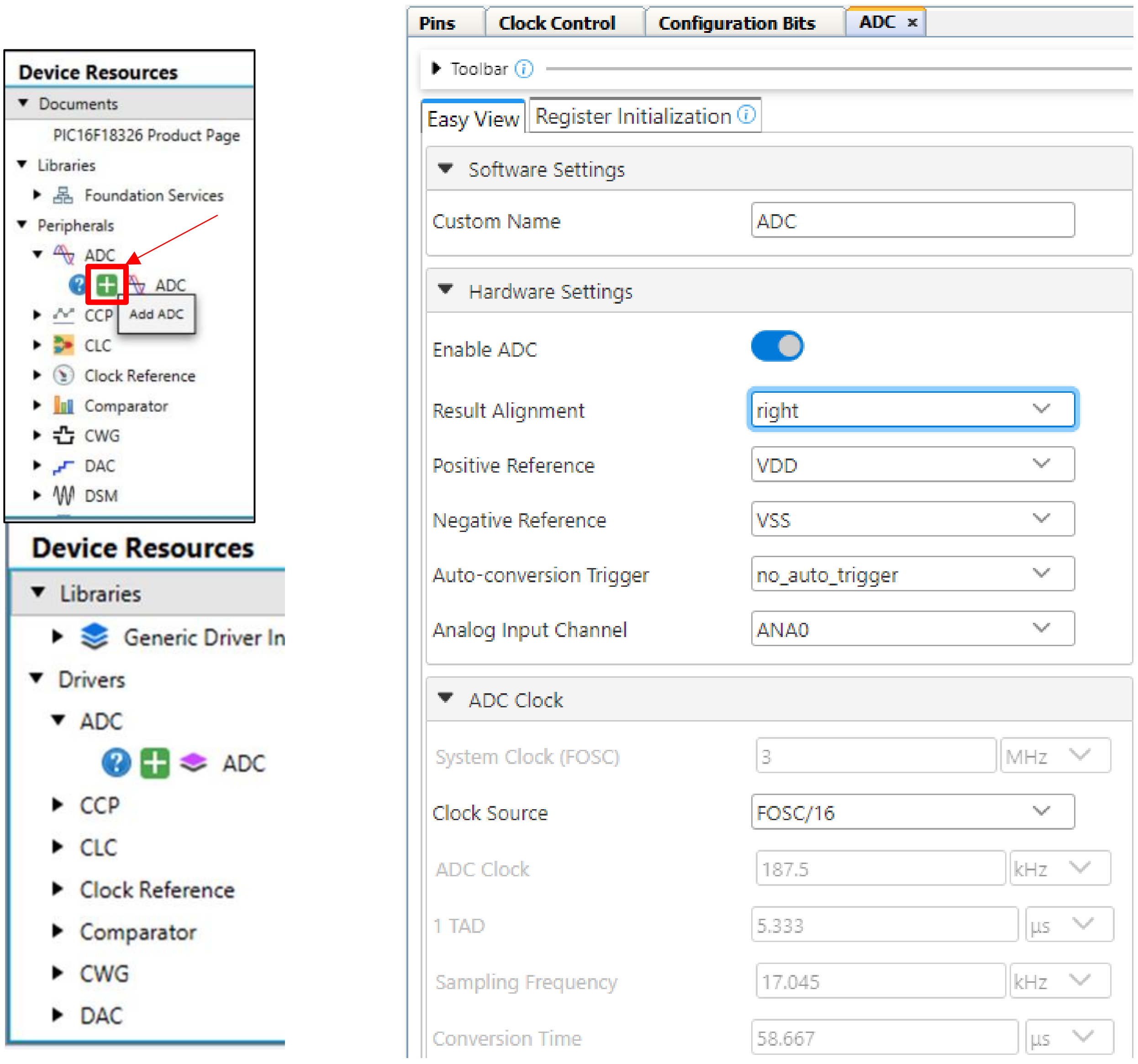
Setup an ADC Input

On the left, we have the 'Device Resources' window. From here, we can set up a peripheral/module of our choice. For the first example, we will setup an ADC.

1. Go to the Device Resources manager. Under drivers, find the ADC drop down. Click on the arrow to reveal a Green + symbol icon. Click on this.
2. For now, ensure the settings are as shown in the image below. **The clock source will need changing to Fosc/16** – a warning in the Notificationsbar will inform us if the ADC clock frequency is too high or low. **Change the result alignment to 'right'**. All of these settings correspond with the

ADCCON1 register shown earlier in the background section, just with a better visual interface.

3. By default, the wizard will set ANA0 or PA0 as an ADC input, but in a previous section (setup GPIO as digital Input/Output), we set this to a digital output, so a conflict occurs, as shown in the Notifications
4. Click back onto the Pin Manager: Grid View, and you will see a chain icon in the new ADC module row under pin 13, and in the GPIO row. **Resolve this conflict by clicking on the GPIO output chain**, this will release it to the ADC module, leaving a green padlock icon on the ADC module row.
5. Finally, go again to the Pin Module as in the Digital In/Out guide (Resource Management > Project Resources), and **make sure Analog is ticked, Output is unticked (output may be forced in some versions, this is fine).**
6. **Don't do this now,** but if we ever need to remove the ADC module, we can click the red X under resource management -> drivers. If we need to modify the settings, simply double click on the ADC row.



Software Settings

Custom Name	ADC

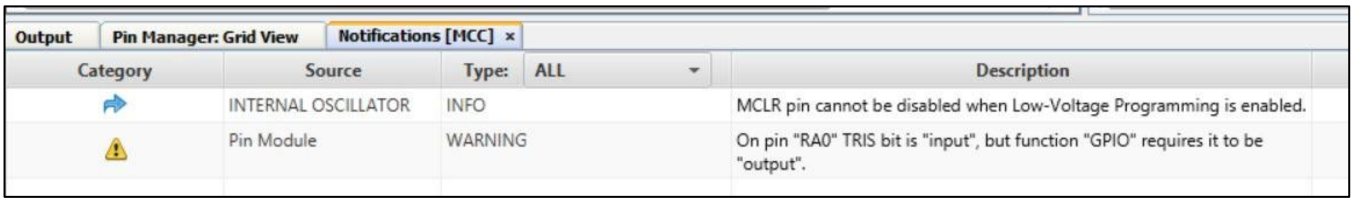
Hardware Settings

Enable ADC	✓
Result Alignment	right
Positive Reference	VDD
Negative Reference	VSS
Auto-conversion Trigger	no_auto_trigger
Analog Input Channel	ANA0

ADC Clock

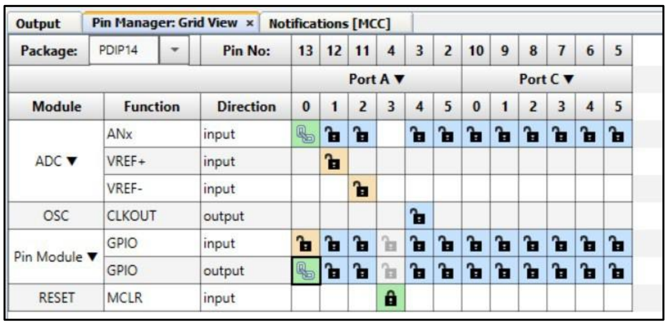
System Clock (FOSC)	3	MHz
Clock Source	FOSC/16	
ADC Clock	187.5	kHz
1 TAD	5.333	µs
Sampling Frequency	17.045	kHz
Conversion Time	58.667	µs

Figure 22: ADC Setup steps 1 and 2; adding ADC peripheral (for UI may be one way or the other due to version).

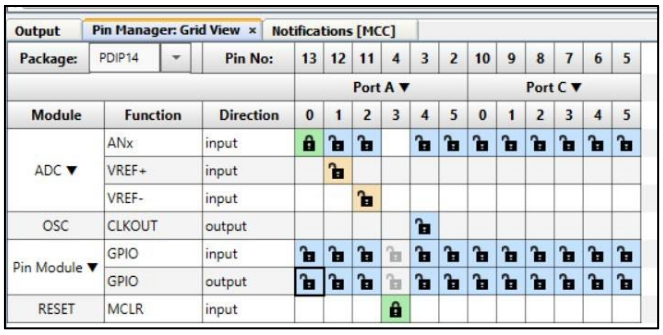


Category	Source	Type: ALL	Description
➡	INTERNAL OSCILLATOR	INFO	MCLR pin cannot be disabled when Low-Voltage Programming is enabled.
⚠	Pin Module	WARNING	On pin "RA0" TRIS bit is "input", but function "GPIO" requires it to be "output".

Figure 23: ADC Setup step 3; conflict warning



Package: PDIP14		Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
ADC ▼	ANx	input	🔗	🔓	🔓		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	VREF+	input		🔓										
	VREF-	input			🔓									
OSC	CLKOUT	output					🔓							
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔗	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								



Package: PDIP14		Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
ADC ▼	ANx	input	🔒	🔓	🔓		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	VREF+	input		🔓										
	VREF-	input			🔓									
OSC	CLKOUT	output					🔓							
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								

Figure 24: ADC Setup step 4; resolving conflict on pin allocation. VREFs under ADC may not be present depending on exact version, either way is fine

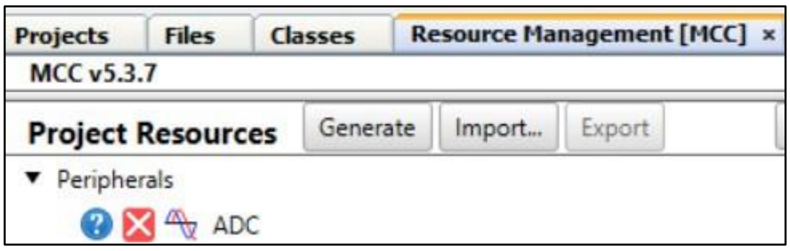


Figure 25: ADC Setup step 6; remove the peripheral or change settings.

Code Generation for Setup so far

1. Once we are happy with the setup, **click the ‘Generate’ button** under ‘Resource Management’ tab in the top left window
2. Now, click back onto the projects tab, then the source files folder. We can now view the generated code. If everything is done correctly in MCC then you shouldn’t need to touch these files and we would advise against editing them unless you know what you are doing.
3. Having a detailed look through these generated files may be of interest and help with general understanding of the system but should be unnecessary for this project

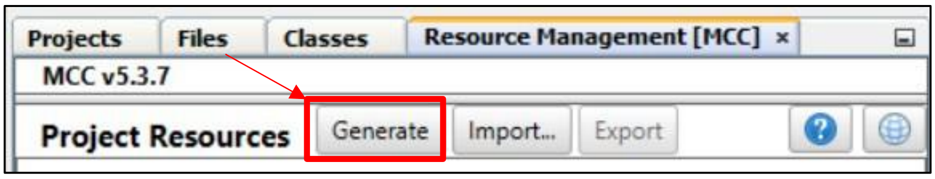
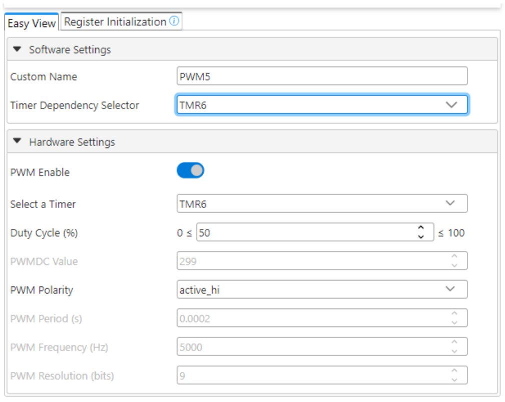


Figure 26: Code Generation Step 1; generate button

Setup a PWM using a PWM Module

As mentioned in the introduction, a PWM can be generated on any pin using a PWM module, or a Capture Compare Module (CCPx). First we'll set one up using the PWM module.

1. After looking at the generated code in the last section, if MCC has closed click again on the blue MCC icon to re-enter the wizard, if it is still open you should be able to change tab to the MCC UI. Re click on the 'resource management' on the right tab if necessary.
2. In device resources, under drivers, select the PWM drop-down, then the PWM5 option, and again press the green '+' button. On this example we'll use the PWM5, but there are two available.
3. **Change the timer source to timer 6**. Here we could also pre-set the duty cycle using the Duty Cycle (%) box, written as a percentage rather than the raw register value.
4. On the Grid View Pin Manager, we can see that the PWM module can be routed to any GPIO pin. For this example, we'll select PA4, **so click on the padlock corresponding to column PA4, and row PWM5.**
5. Go again to the Pin Module (Resource Management > Project Resources), and make sure **Analog is unticked, Output is ticked, for PWM5 (in some versions output may be forced on, this is fine)**.
6. **Setup the timer chosen for this PWM module, Timer 6.** See the 'Setup a Timer for the PWMs' section. If not setup, we receive a warning in the notifications box.

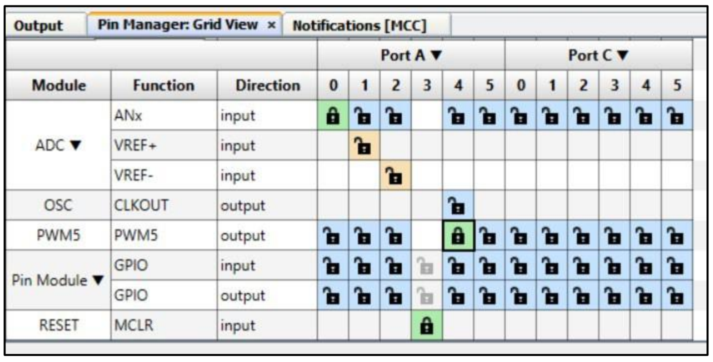


Software Settings

Custom Name	PWM5
Timer Dependency Selector	TMR6

Hardware Settings

PWM Enable	
Select a Timer	TMR6
Duty Cycle (%)	0 ≤ 50 ≤ 100
PWMDC Value	299
PWM Polarity	active_hi
PWM Period (s)	0.0002
PWM Frequency (Hz)	5000
PWM Resolution (bits)	9



			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
ADC ▼	ANx	input	🔒	🔓	🔓		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	VREF+	input		🔓										
	VREF-	input			🔓									
OSC	CLKOUT	output					🔓							
PWM5	PWM5	output	🔓	🔓	🔓		🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								

Figure 32: PWM Setup step 3 and 4; setting up PWM 5

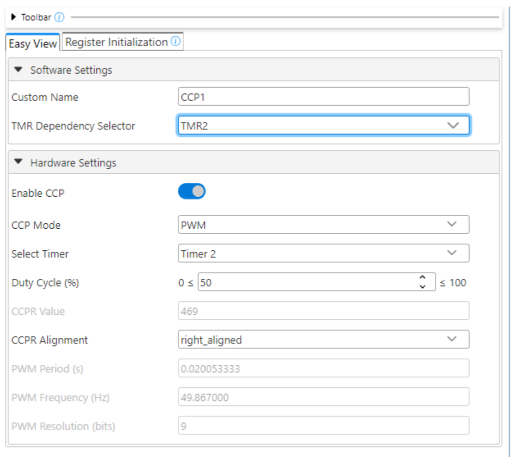
⚠ PWM5	WARNING	Configure Timer6 for PWM module.

Figure 33: Warning due to Timer 6 not being setup for the PWM5 Module

Setup a PWM using a CCP Module

A PWM can also be setup on a CCP module.

1. In device resources, under drivers, select the CCP drop down, then the CCP1 option, and press the green '+' button. On this example we'll use the CCP1, but there are four available.
2. **Change the CCP mode to 'PWM'. Select timer 2 as the source**. Here we could also pre-set the duty cycle by writing to the Duty Cycle % option, written as a percentage rather than a raw register value.
3. **Set the output onto the desired pin; in this case, PA2**, on the grid view.
4. Go again to the Pin Module (Resource Management > Project Resources), and **make sure Analog is unticked, Output is ticked (again output may be forced on, do not worry about this)**.
5. **Setup the timer chosen for this PWM module, timer 2.** See the 'Setup a Timer for the PWMs' section. We will receive a warning if not set up in the notifications box.



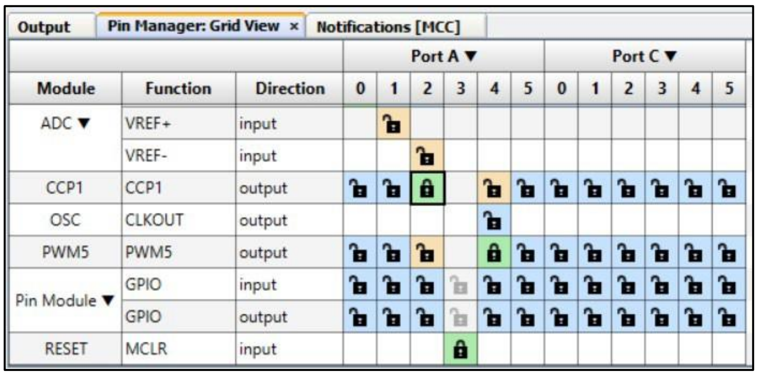
Software Settings

Custom Name	CCP1
TMR Dependency Selector	TMR2

Hardware Settings

Enable CCP	●
CCP Mode	PWM
Select Timer	Timer 2
Duty Cycle (%)	0 ≤ 50 ≤ 100
CCPR Value	469
CCPR Alignment	right_aligned
PWM Period (s)	0.020053333
PWM Frequency (Hz)	49.867000
PWM Resolution (bits)	9

Figure 34: Setup PWM using a CCP Module, step 2



			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
ADC ▼	VREF+	input		🔓										
	VREF-	input			🔓									
CCP1	CCP1	output	🔓	🔓	🔒		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
OSC	CLKOUT	output					🔓							
PWM5	PWM5	output	🔓	🔓	🔓		🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								

Figure 35: Setup PWM using a CCP Module, step 3

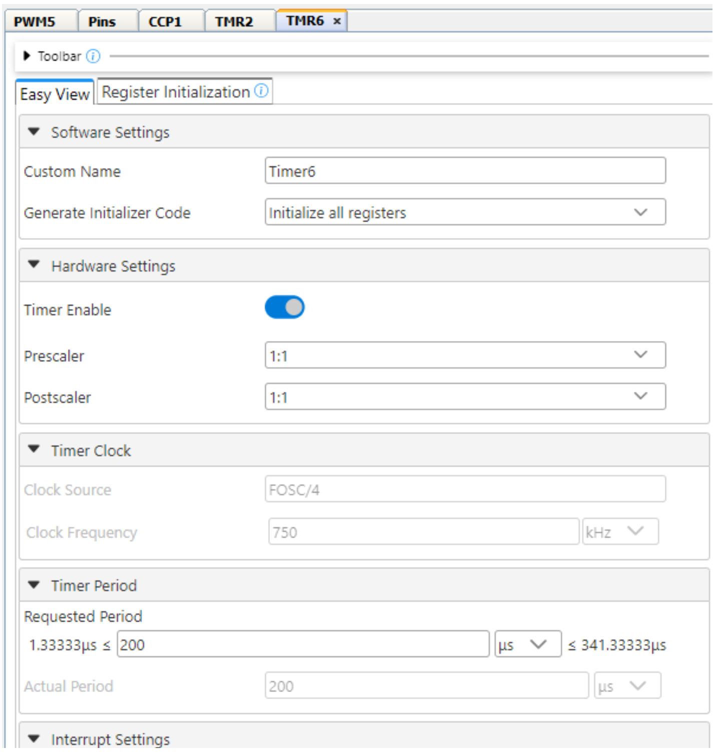
CCP1 WARNING Please configure the Timer 2 for PWM mode.

Figure 36: Warning due to Timer not being setup for CCP1 module

Setup a Timer for the PWMs

Setting up a timer using MCC is very easy due to the interface. A handy tool for deciding on prescalers and setting the PRx value is included.

1. In device resources, under drivers, select the Timer drop down, then the TMR6 option, and press the green '+' button. On this example only TMR6 is shown as we require it for the PWM5 we set up earlier, but there are six available. We also need to set up TMR2 for the CCP1 module.
2. The Timer period box shows the minimum and maximum period capable of being produced under the current prescaler and postscaler values. The postscalers allow more precision compared to the prescalers however the PWM modules receive the Timer output before postscalers, **so we can only use prescalers**. A PR6 value of 256 gives the maximum period shown on the right, and 1 would give the minimum on the left. **After deciding on the prescaler value, here we have selected 1:1; modify the number in the box between the min and max values, here we have selected 200us**. Based on the actual value of period we get from an integer value of PR6 (it cannot be a decimal!), the 'Actual Period' is displayed. As visible, the settings below result in a period of 200us, from a demanded value of 200us, so is the exact value.
3. We cannot see a change in the Pin Manager Grid view, but the timer is visible in the resource management window.



Software Settings

Custom Name: Timer6

Generate Initializer Code: Initialize all registers

Hardware Settings

Timer Enable: on

Prescaler: 1:1

Postscaler: 1:1

Timer Clock

Clock Source: FOSC/4

Clock Frequency: 750 kHz

Timer Period

Requested Period: 1.33333µs ≤ 200 µs ≤ 341.33333µs

Actual Period: 200 µs

Interrupt Settings

Figure 37: Timer Setup Step 2

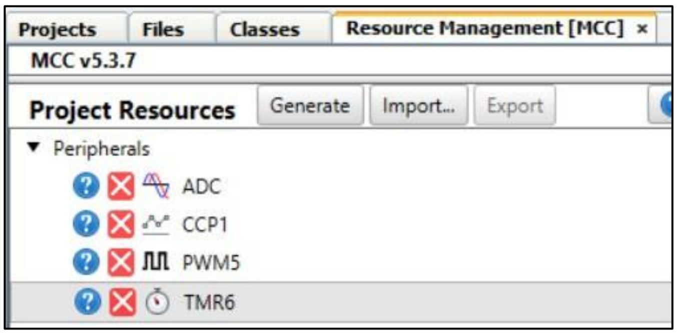
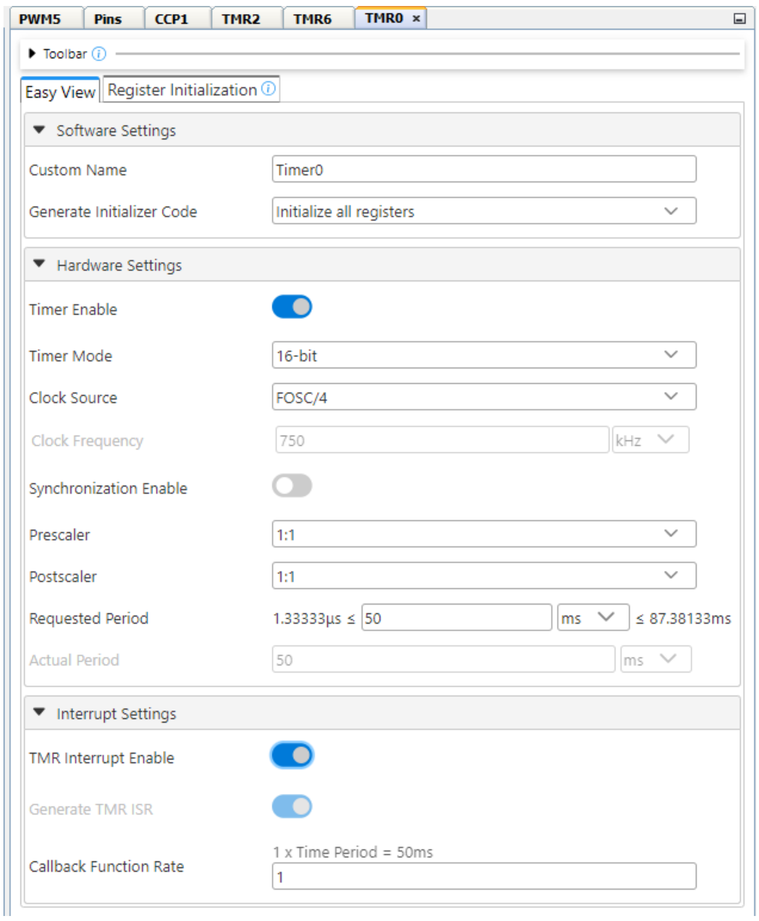


Figure 38: Timer Setup step 3; Timer visible in the Resource Management Window

Setup a Timer for Time Triggered Scheduling

We may also wish to use timers to perform time triggered scheduling, in other words execute tasks (reading ADCs, changing PWMs, performing PID control) at a regular rate. To do this, we use a timer interrupt, which interrupts the current algorithm the microcontroller is performing to perform the interrupt task when the timer count finishes. We will use Timer 0 for this in this example. We may want to cause this timer interrupt to occur at a high frequency for a faster controller, however the code it executes takes time, therefore the execution rate must be low enough to perform all tasks before triggering again. *For future reference, we can measure the time it takes to execute the controller by toggling a GPIO before and after the code*. For this example we will use 50ms as the interrupt rate.

1. In device resources, under drivers, select the Timer drop down, then the TMR0 option, and press the green '+' button.
2. Timer0 can be incremented (count) by an external signal using the T0CKI_PIN selection for the clock source. In this example we will use the internal clock, therefore **choose FOSC/4 for the clock source**.
3. **Change the prescaler, post scaler and timer mode unti**l the numbers either side of the timer period selection **allow 50ms in the 'requested period' box** Here, we have used a **16-bit timer mode**, so the timer duration is longer as it must count higher. **The clock prescaler is 1:1**. Then, **type 50ms** into the requested period box.
4. **Tick the Enable Timer Interrupt** box.
5. Go to the Interrupt Module (Resource Management > Project Resources). **Ensure Timer0 interrupt is ticked**.
6. The warning in the interrupt module tells use to enable Peripheral and Global interrupts in our code; these are registers which allow us to turn on or off all interrupts at once. We will turn these on later.



Software Settings

Custom Name	Timer0
Generate Initializer Code	Initialize all registers

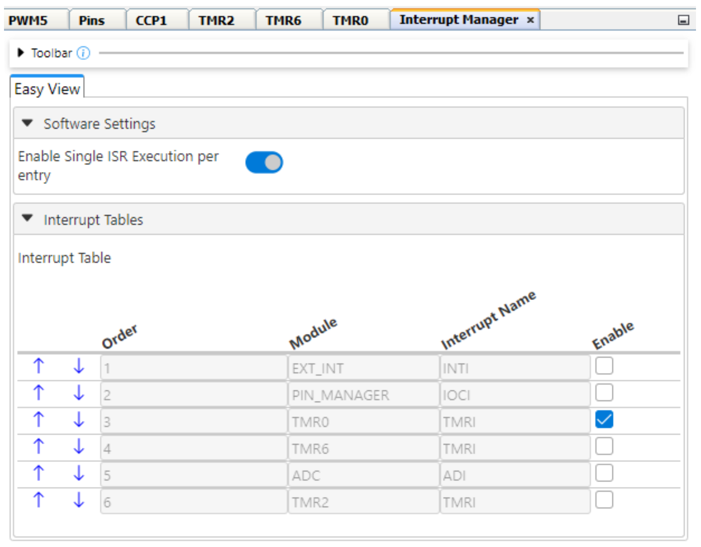
Hardware Settings

Timer Enable	On
Timer Mode	16-bit
Clock Source	FOSC/4
Clock Frequency	750 kHz
Synchronization Enable	Off
Prescaler	1:1
Postscaler	1:1
Requested Period	1.33333µs ≤ 50 ms ≤ 87.38133ms
Actual Period	50 ms

Interrupt Settings

TMR Interrupt Enable	On
Generate TMR ISR	On
Callback Function Rate	1 x Time Period = 50ms 1

Figure 39: Timer Interrupt for TTS step 2 to 4



Software Settings

Enable Single ISR Execution per entry: ✓

Interrupt Tables

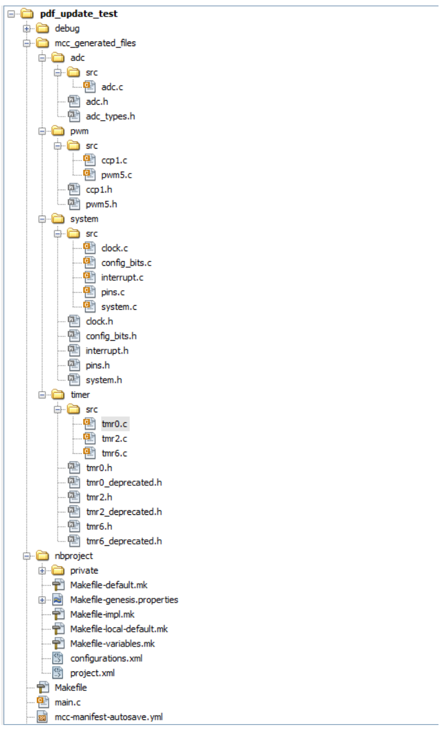
Interrupt Table

Order	Module	Interrupt Name	Enable
1	EXT_INT	INTI	☐
2	PIN_MANAGER	IOCI	☐
3	TMR0	TMRI	✓
4	TMR6	TMRI	☐
5	ADC	ADI	☐
6	TMR2	TMRI	☐

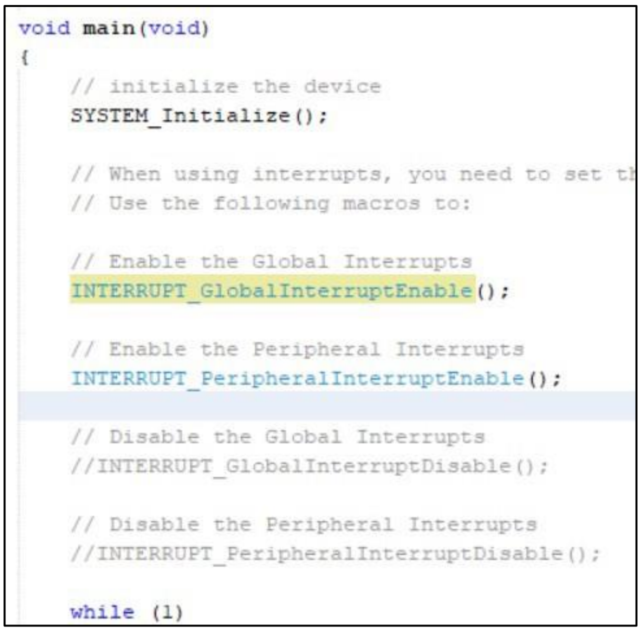
Figure 40: Timer Interrupt for TTS step 5; interrupt module

Code Generation for PWM and Timer Modules

1. **Set up timer 2 for CCP1, revisit the 'Setup Timer for the PWMs' but for a period of 20ms (50Hz) using a prescaler of 1:64** (not shown in guide). Check for any warnings in the notifications bar.
2. Once we are happy with the setup**, click the generate button** under 'Resource Management' tab in the top left window as before.
3. Now, click back onto the projects tab, then the source files folder. We can now view the generated code.
4. We have new libraries for Timer 0, 2 and 6, and PWM 1 and 5. PWM 1 corresponds to the capture compare module, CCP1, whereas PWM5 is the PWM module 5. We will not go through each library but they contain functions to setup the modules as specified, and additional functions we can use in our code to alter the modules in real time.
5. As we now use interrupts, we must **enable Global and Peripheral interrupts, by removing the // in front of INTERRUPT_PeripheralInterruptEnable(); and INTERRUPT_GlobalInterruptEnable();** in the main function as in the image below.
6. We also have a new library called interrupt_manager, since we enabled the timer 0 interrupt. Within the C file for this, we see the interrupt manager. It checks what caused the latest interrupt using if statements. If caused by our Timer 0 interrupt, it will run Timer0_ISR (ISR for interrupt service routine), which is in the TMR0.c file. This prepares the timer for the next interrupt, then runs the interrupt handler; we will create this later.

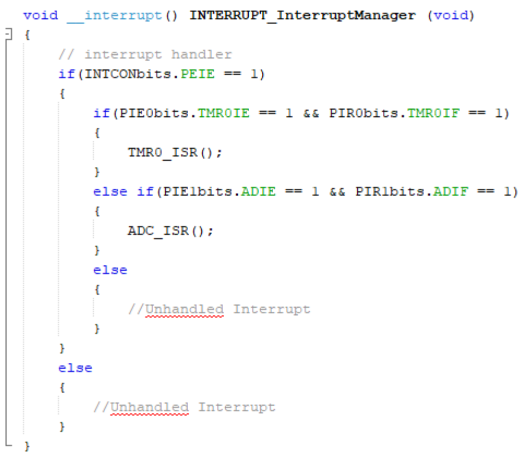


```
pdf_update_test
  debug
  mcc_generated_files
    adc
      src
        adc.c
      adc.h
      adc_types.h
    pwm
      src
        ccp1.c
        pwm5.c
      ccp1.h
      pwm5.h
    system
      src
        clock.c
        config_bits.c
        interrupt.c
        pins.c
        system.c
      clock.h
      config_bits.h
      interrupt.h
      pins.h
      system.h
    timer
      src
        tmr0.c
        tmr2.c
        tmr6.c
      tmr0.h
      tmr0_deprecated.h
      tmr2.h
      tmr2_deprecated.h
      tmr6.h
      tmr6_deprecated.h
  nbproject
    private
    Makefile-default.mk
    Makefile-genesis.properties
    Makefile-impl.mk
    Makefile-local-default.mk
    Makefile-variables.mk
    configurations.xml
    project.xml
  Makefile
  main.c
  mcc-manifest-autosave.yml
```

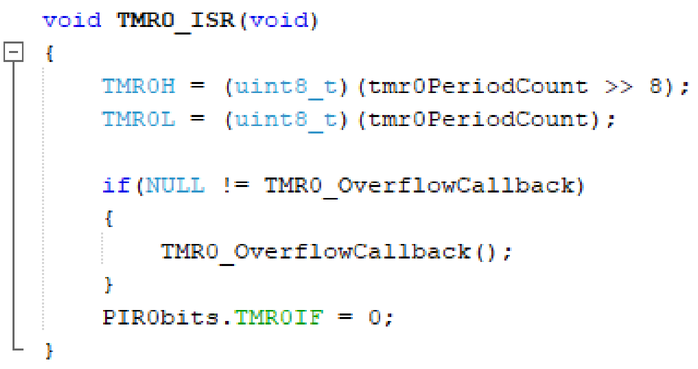


```
void main(void)
{
    // initialize the device
    SYSTEM_Initialize();

    // When using interrupts, you need to set tl
    // Use the following macros to:

    // Enable the Global Interrupts
    INTERRUPT_GlobalInterruptEnable();

    // Enable the Peripheral Interrupts
    INTERRUPT_PeripheralInterruptEnable();

    // Disable the Global Interrupts
    //INTERRUPT_GlobalInterruptDisable();

    // Disable the Peripheral Interrupts
    //INTERRUPT_PeripheralInterruptDisable();

    while (1)
```

Figure 41: Code Generation for PWM and Timer Modules; new libraries, and step 5; enable interrupts in the code



```
void __interrupt() INTERRUPT_InterruptManager (void)
{
    // interrupt handler
    if(INTCONbits.PEIE == 1)
    {
        if(PIE0bits.TMR0IE == 1 && PIR0bits.TMR0IF == 1)
        {
            TMR0_ISR();
        }
        else if(PIE1bits.ADIE == 1 && PIR1bits.ADIF == 1)
        {
            ADC_ISR();
        }
        else
        {
            //Unhandled Interrupt
        }
    }
    else
    {
        //Unhandled Interrupt
    }
}
```



```
void TMR0_ISR(void)
{
    TMR0H = (uint8_t)(tmr0PeriodCount >> 8);
    TMR0L = (uint8_t)(tmr0PeriodCount);

    if(NULL != TMR0_OverflowCallback)
    {
        TMR0_OverflowCallback();
    }
    PIR0bits.TMR0IF = 0;
}
```

Figure 42: Code Generation for PWM and Timer Modules step 6; interrupt manager and ISR in TMR0.c called by interrupt manager. Don't worry if the ADC part is missing on the interrupt manager section

Example Code Using Functions Generated by MCC

This section will provide an example of writing code using the functions generated by the MCC wizard. In this example**, we will adjust the duty cycle of the PWM according to the value of an ADC, which is varied by a potentiometer**. We want to make it so that across the full range of the potentiometer, the duty cycle varies from 0 to 100%, therefore some scaling is required. This operation will occur using the timer interrupt, every 50ms.

Scaling the ADC Value to the Duty Cycle Value - Explanation

As explained in the introduction section, **the duty cycle is at 100% when the integer value in the duty cycle register is 4 times that of the period register, (e.g PR6 for timer 6)**. In the Timer 6 initialisation code (tmr6.c in MCC Generated Files), we can see that PR6 is set to a decimal value of 149 (hex 0x95) for our 200us period (see figure below). 100% duty cycle corresponds to a PWM5DC register value of 149 * 4 = 596.

In this example, we will use the ‘uint16_t’ and ‘uint32_t’ variable classes, which are unsigned integers of limited size (16 bit and 32 bit), useful in embedded applications due to reducing the size of code and hence increasing computation speed. The variables we will use which store the ADC read value and the Duty Cycle value will be ‘uint16_t’, meaning their maximum value is 2^16 = 65,536.

The max ADC read value is 1024. **Therefore, to convert the ADC reading to Duty Cycle, we must scale the ADC reading down at a ratio of 1024 to 596** (multiply the ADC reading by 596/1024). To do this, first multiply the ADC read value by 596 (if we divided by 1024 first, the result would be a decimal, which is not supported by integer variables!). The result is a maximum value of 1024 * 596 = 610,304, which is well above the maximum value of our 16-bit variable (2^16 = 65,536). We must therefore increase the memory size of the result, by instructing the compiler to convert the variable we are operating on (ADCread) to 32 bit, using a (uint32_t) command before the variable. Also do this for the result of our calculation. We can then divide by 1024 to obtain a value which corresponds to 100% duty cycle when the ADC read is 1024. The max value of the result is 596, so can be stored in a 16 bit variable; DutyCycle. Be careful with the placement of brackets; we want the multiplication to occur first.

Setting up the Example Code

The following modules will need to be set up for this example:

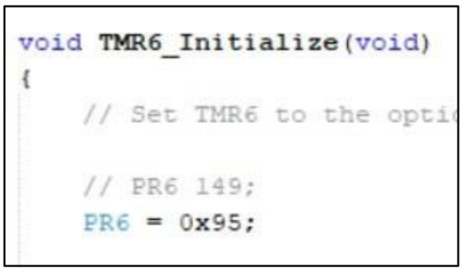
- Timer 0 set up to a 50ms interrupt as previously shown
- The ADC setup as previously shown
- PWM Module 5 setup onto pin PA4 and using Timer 6, as shown previously
- Timer 6 setup to 200us
- **ADC input on pin RC2 (ANC2)**
- **A GPIO Digital Output on pin RC1**

Most of these have been done in the guide so far, except for the last 2. **You will need to set these up now! (ADC input on RC2, GPIO digitals output on RC1).** Make sure the code is generated, then open the main.c file.s

1. Before the main function, **create a new void function called 'timer_Interrupt_Routine()'**, or similar name. Within this function, for now **set it to simply toggle the digital output on RC1**. To do this, we will use one of the functions we viewed earlier in the pin_manager.h file, but for RC1, named 'IO_RC1_Toggle()'. This should turn blue if recognised by the IDE. **If it doesn't turn blue, check that the MCC pin manager grid view, and the pin module settings for RC1 have been correctly configured.**
2. In the main function, before the while loop**, set the interrupt handler to the function we have just created** by passing the name of the function into the interrupt handler setup function; 'INT_SetInterruptHandler(timer_Interrupt_Routine)' (might be 'TMR0_SetInterruptHandler(timer_Interrupt_Routine);' in some versions). This tells the interrupt routine to run our function when we have a timer interrupt.
3. **Create a new variable named 'ADCresult'**. Use the 'uint16_t' variable type.
4. **Read the ADC using the ADC_ChannelSelectAndConvert (channel) function (might be called ADC_GetConversion(channel) in some versions)** we saw in ADC.c earlier, do this within the timer_Interrupt_Routine function. Pass the channel of the ADC we want to use, RC2. The list of channel names can be found in ADC.h. For RC2, we will use channel_ANC2. Write the result in the ADCresult variable.
5. **Perform the scaling and write the result in a new variable called DutyCycle,** again within the interrupt function**.** As described earlier, to do this, multiply the ADCresult by 596, informing the compiler the result will be

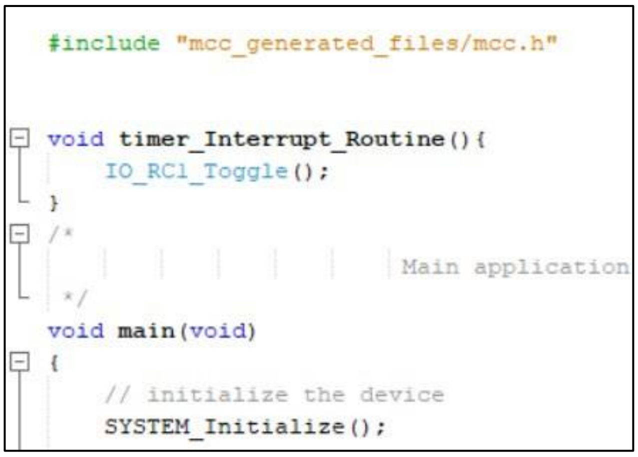
a larger, 32 bit unsigned value, then divide by 1024. **See '*Figure 46: MCC Code Example Steps 3, 4 and 5*' below for the full code**.

6. Finally, **use the PWM5_LoadDutyValue() function from the PWM5.c file to update the duty cycle**, by passing the DutyCycle variable.
7. **Compile the code** by clicking on the 'hammer' symbol.



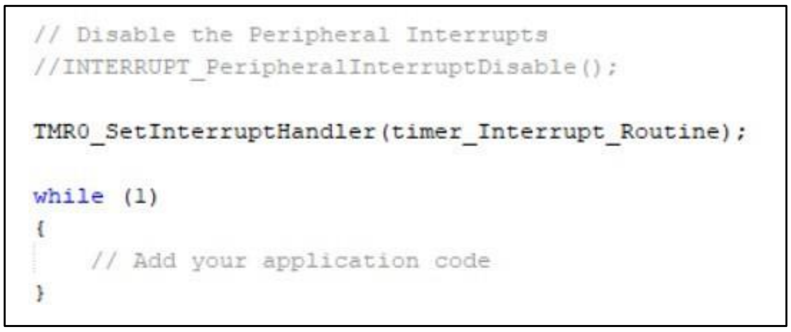
```
void TMR6_Initialize(void)
{
    // Set TMR6 to the opti

    // PR6 149;
    PR6 = 0x95;
```

Figure 43: MCC Code Example PR6 Value



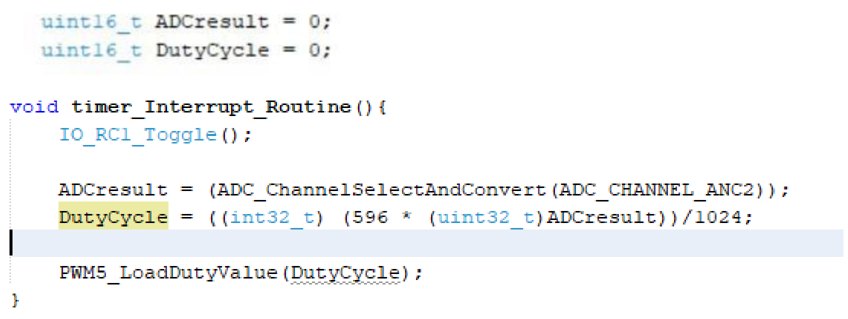
```
#include "mcc_generated_files/mcc.h"


void timer_Interrupt_Routine(){
    IO_RC1_Toggle();
}
/*
                        Main application
 */
void main(void)
{
    // initialize the device
    SYSTEM_Initialize();
```

Figure 44: MCC Code Example Step 1



```
// Disable the Peripheral Interrupts
//INTERRUPT_PeripheralInterruptDisable();

TMR0_SetInterruptHandler(timer_Interrupt_Routine);

while (1)
{
    // Add your application code
}
```

Figure 45: MCC Code Example Step 2



```
    uint16_t ADCresult = 0;
    uint16_t DutyCycle = 0;

void timer_Interrupt_Routine(){
    IO_RC1_Toggle();

    ADCresult = (ADC_ChannelSelectAndConvert(ADC_CHANNEL_ANC2));
    DutyCycle = ((int32_t) (596 * (uint32_t)ADCresult))/1024;

    PWM5_LoadDutyValue(DutyCycle);
}
```

Figure 46: MCC Code Example Steps 3, 4 and 5

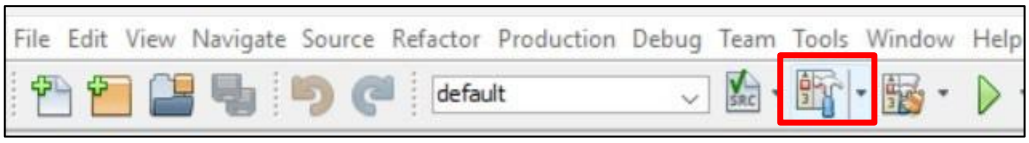


Figure 47: MCC Code Example Step 6; Build Button

Programming the PIC and Debugging the Code

Next, we will debug our code to make sure it operates as we want it to. Firstly, either using breadboard or the controller interface board, connect the PIC as in the diagram below, with a 5V supply and decoupling capacitor, a potentiometer with the wiper on RC2, and programming pins connected to the programming header with a pullup resistor on MCLR. On the controller board, these are all present already **except the potentiometer**, and the programming header is J15. Connect oscilloscope probes to RC1 (Digital Out) and RA4 (PWM).

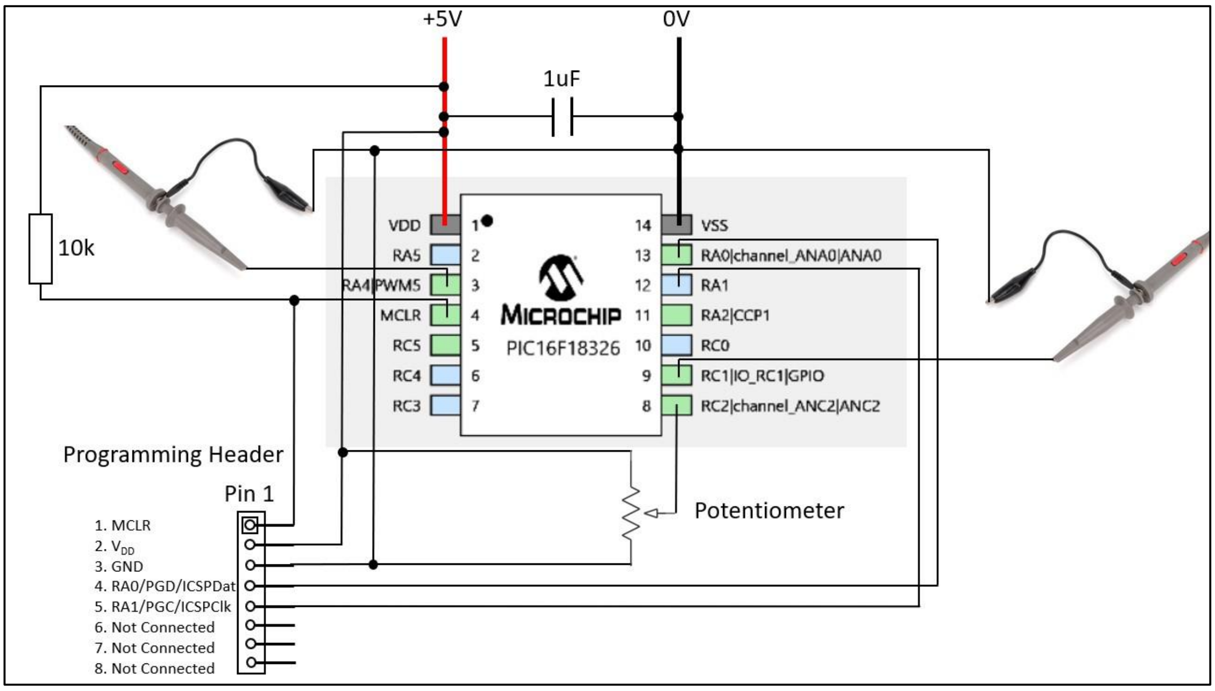
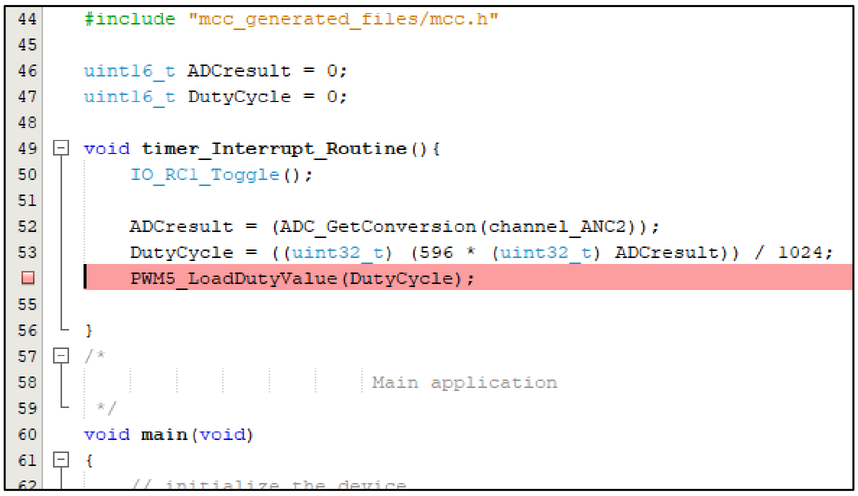


Figure 48: Code Example Debugging and Experiment Setup

1. Once the PIC test set up has been prepared, **connect the Snap-SN or the PICKit to the debug header**, making sure to line up the arrow on the side of the connector with pin 1.
2. First, **place a breakpoint** on the line of code which loads the new duty cycle value, do this by double clicking the line number to the left of the function. A red square will appear and the line will turn red.
3. Next, **load the debug program onto the PIC** by clicking the debug button shown in the figure below, with a breakpoint and a play button on. The program should compile successfully, before giving a warning that the correct PIC must be connected. Check the device code is correct in this message, before pressing OK.
4. If successful, the debugger will run through the code before stopping at the breakpoint made earlier. **If this does not happen, check with a**

demonstrator and/or retrace earlier steps. The breakpoint line will turn green to show it has paused here. Click on the 'breakpoints' tab on the window on the bottom of the screen to view a list of breakpoints, we can tick or untick breakpoints made earlier to check different parts of the code.

5. **Click the green play button** located at the top, again the debugger should stop at this breakpoint on the next interrupt.
6. **Add a watch on the two variables of interest,** this will allow us to see the variable values when the debugger is paused; right click on the 'ADCresult' variable, and click 'new watch', then press OK on the pop up. Do the same for the 'DutyCycle' variable. To view the watches, click on the 'windows' tool bar, then 'debugging > watches' as shown below. This should add a new tab on the botom window showing the watches.
7. The watches may be shown as hexadecimal values. To show them in decimal, right click on one of the column titles (e.g 'Value') and left click o the decimal option.
8. **Rotate the potentiometer fully clockwise, then press play.** The debugger should again stop on the breakpoint, and the watches should be updated with the current variable values. In one direction of the potentiometer, te 'ADCread' value should be close to 1024, and the 'DutyCycle' value close to 596 (or the value you used in the code corresponding to max duty cycle). In the maximum other direction, they should be both close to 0.
9. **Test the ADC through the full range** by rotating the potentiometer, then pressing play and observing the new values.
10. Once happy with the code, click the red square stop button to stop debugging, then click the button on the top toolbar with the downwards green arrow to program the device permanently.
11. Once programmed and running, one oscilloscope channel should display the timer interrupt executions, with the output on RC1 toggling every 50ms The other channel should show the higher frequency PWM5 output, with a period of 200us. Varying the potentiometer should vary the duty cycle of this PWM.



```
#include "mcc_generated_files/mcc.h"

uint16_t ADCresult = 0;
uint16_t DutyCycle = 0;

void timer_Interrupt_Routine(){
    IO_RC1_Toggle();

    ADCresult = (ADC_GetConversion(channel_ANC2));
    DutyCycle = ((uint32_t) (596 * (uint32_t) ADCresult)) / 1024;
    PWM5_LoadDutyValue(DutyCycle);

}
/*
                            Main application
 */
void main(void)
{
    // initialize the device
```

Figure 49: Debugging the PIC step 2; adding a breakpoint

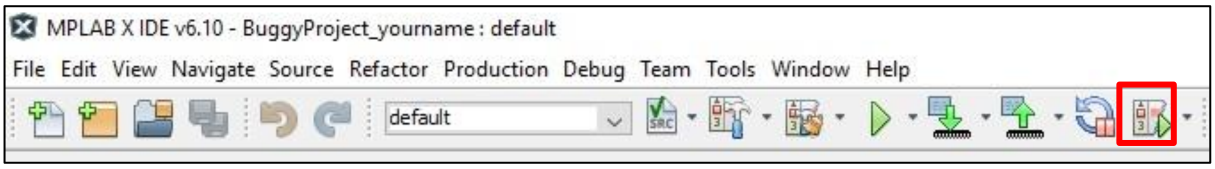
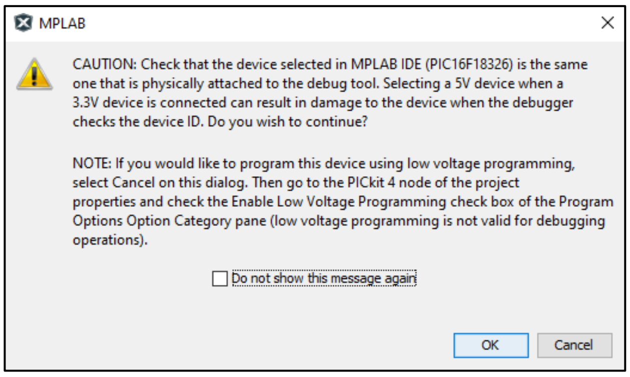

MPLAB X IDE v6.10 - BuggyProject_yourname : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default

Figure 50: Debugging the PIC step 3; the debug button



MPLAB

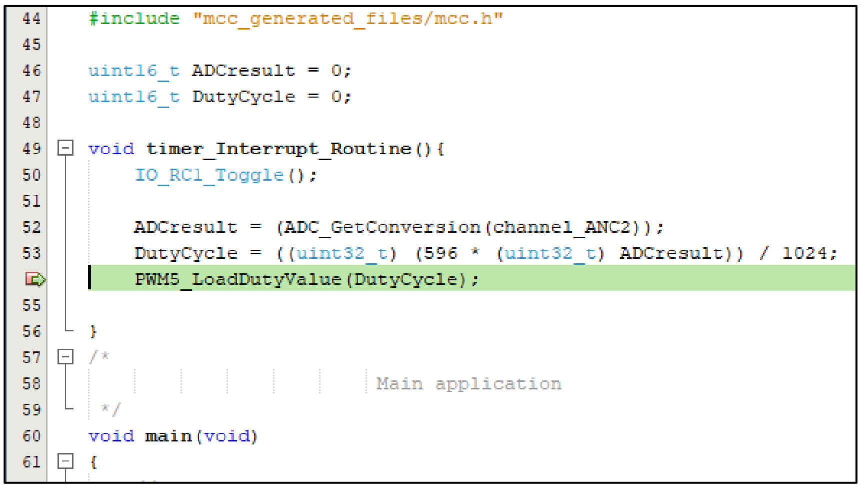
CAUTION: Check that the device selected in MPLAB IDE (PIC16F18326) is the same one that is physically attached to the debug tool. Selecting a 5V device when a 3.3V device is connected can result in damage to the device when the debugger checks the device ID. Do you wish to continue?

NOTE: If you would like to program this device using low voltage programming, select Cancel on this dialog. Then go to the PICkit 4 node of the project properties and check the Enable Low Voltage Programming check box of the Program Options Option Category pane (low voltage programming is not valid for debugging operations).

☐ Do not show this message again

OK Cancel

Figure 51: Debugging the PIC step 3; warning message



```
#include "mcc_generated_files/mcc.h"

uint16_t ADCresult = 0;
uint16_t DutyCycle = 0;

void timer_Interrupt_Routine(){
    IO_RC1_Toggle();

    ADCresult = (ADC_GetConversion(channel_ANC2));
    DutyCycle = ((uint32_t) (596 * (uint32_t) ADCresult)) / 1024;
    PWM5_LoadDutyValue(DutyCycle);

}
/*
                            Main application
 */
void main(void)
{
```

Figure 52: Debugging the PIC step 4; debug pausing at breakpoint

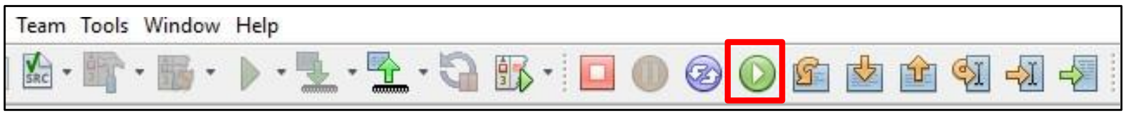
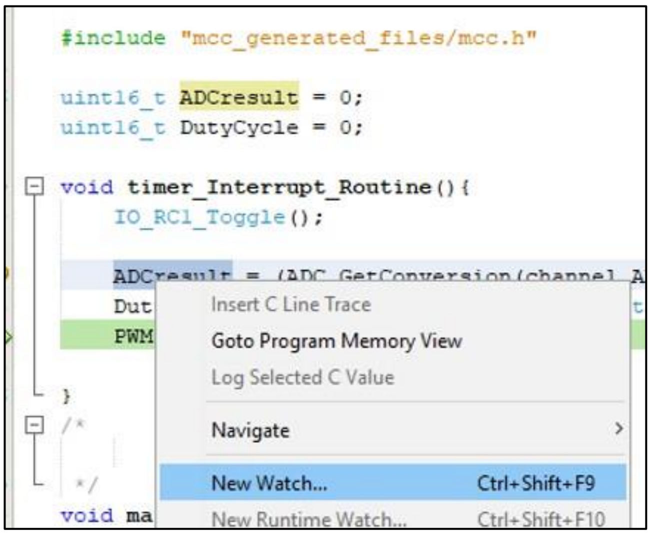


Figure 53: Debugging the PIC step 5; the play button



```
#include "mcc_generated_files/mcc.h"

uint16_t ADCresult = 0;
uint16_t DutyCycle = 0;

void timer_Interrupt_Routine(){
    IO_RC1_Toggle();

    ADCresult = (ADC_GetConversion(channel_A
    Dut
    PWM
}
/*

 */
void ma
```

Figure 54: Debugging the PIC step 6; adding a new 'watch'

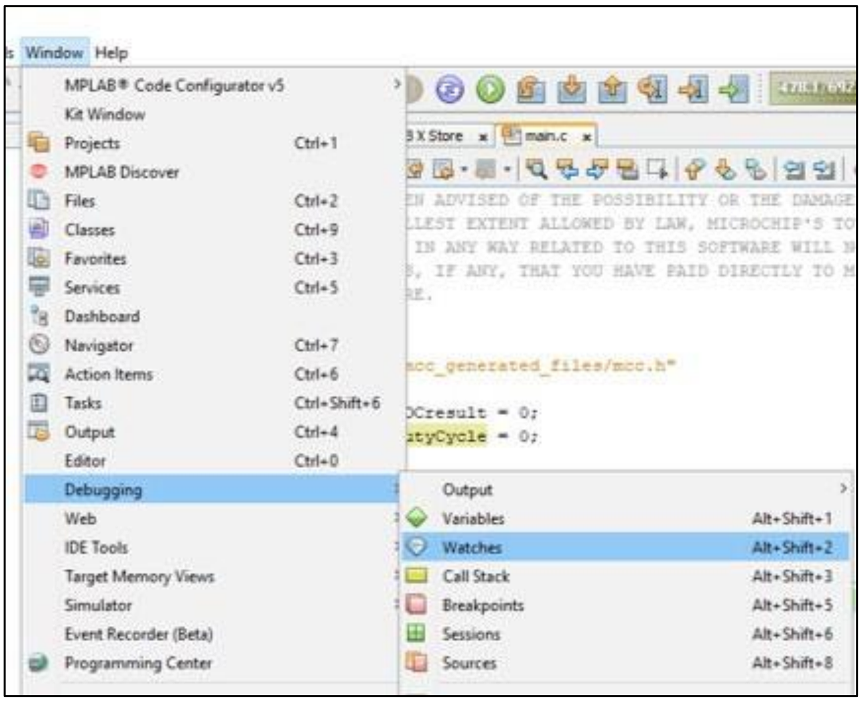


Figure 55: Debugging the PIC step 6; adding the 'watches' tab

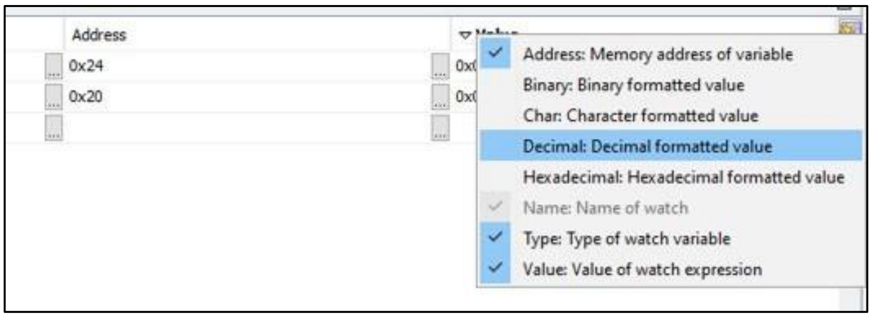
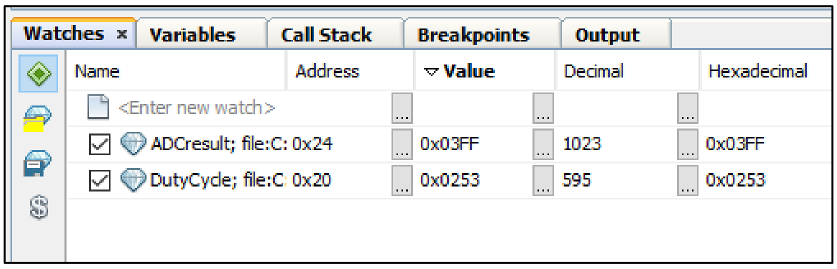

Address
0x24
0x20
Address: Memory address of variable
Binary: Binary formatted value
Char: Character formatted value
Decimal: Decimal formatted value
Hexadecimal: Hexadecimal formatted value
Name: Name of watch
Type: Type of watch variable
Value: Value of watch expression

Figure 56: Debugging the PIC step 7; showing decimal values


Watches × | Variables | Call Stack | Breakpoints | Output

Name	Address	Value	Decimal	Hexadecimal
<Enter new watch>				
ADCresult; file:C: 0x24		0x03FF	1023	0x03FF
DutyCycle; file:C: 0x20		0x0253	595	0x0253

Figure 57: Debugging the PIC step 8; maximum ADC equating to maximum duty

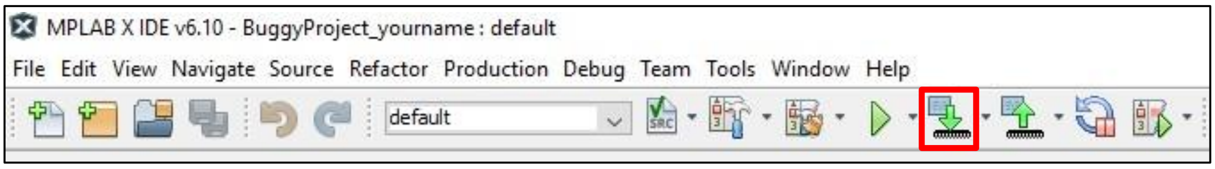

MPLAB X IDE v6.10 - BuggyProject_yourname : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default

Figure 58: Programming the PIC

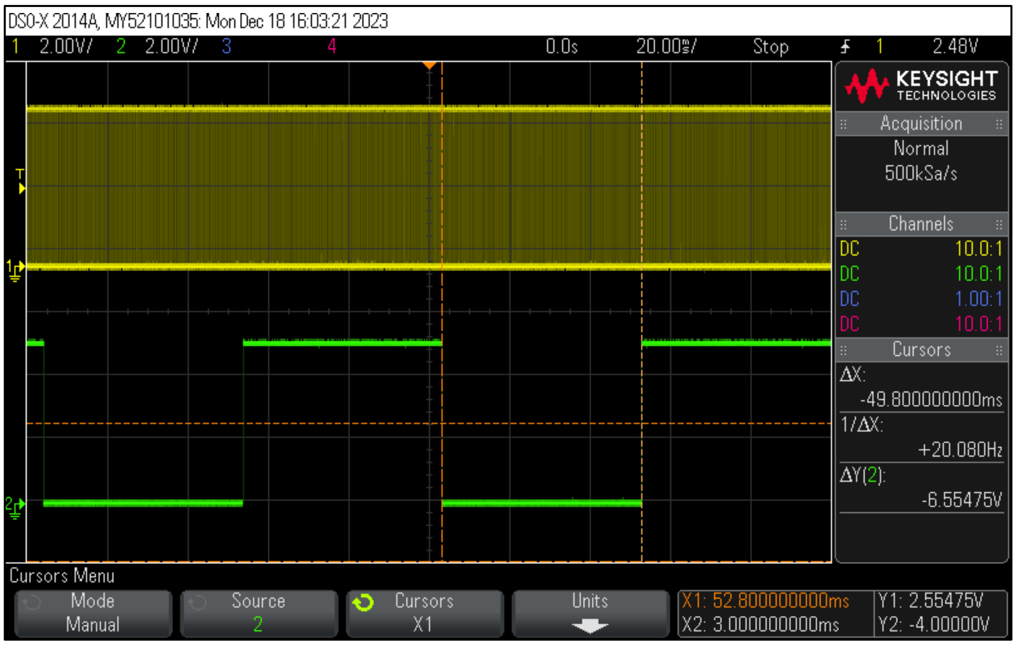
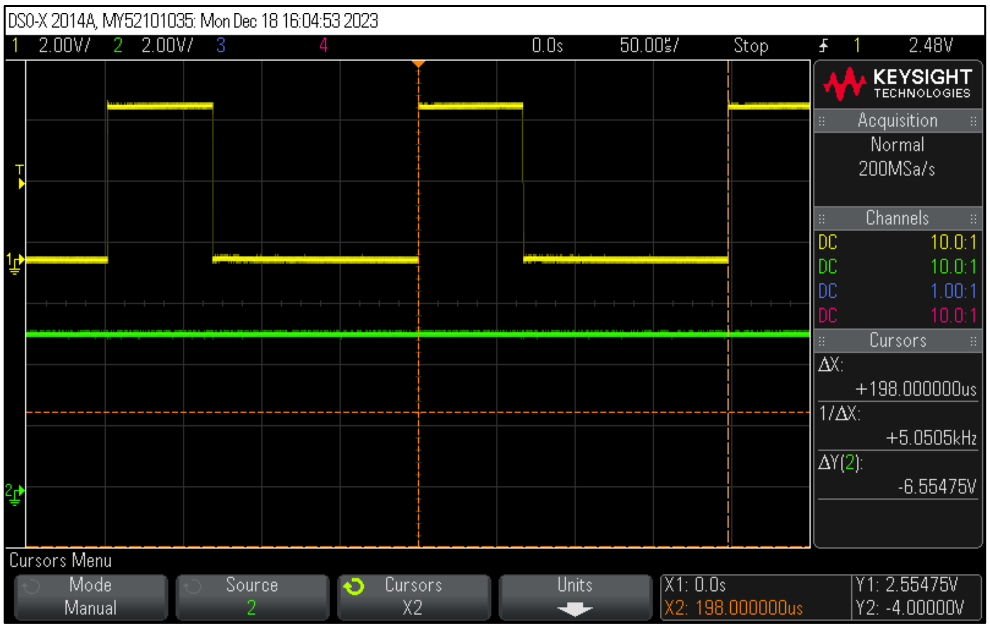


Figure 59: Timer interrupt GPIO toggles



ΔX: +198.000000us
1/ΔX: +5.0505kHz
ΔY(2): -6.55475V
X1: 0.0s
X2: 198.000000us
Y1: 2.55475V
Y2: -4.00000V

Figure 60: PWM5 output

Alternate Scaling Method Using Floats

The previous method is the fastest and most memory efficient way of performing the ADC scaling. However, you may also wish to use floats in your code, especially for the PID controller, as it will be much simpler. You can test this method out and run another debug to observe how it works. Note the memory consumption will be much higher, as seen in the figures below.

1. **Add a new variable**, with the **float** type, named **ADCScaling**.
2. **Divide the ADC reading by 1024 and write to the ADCScaling variable,** this result is not an integer and contains a fractional value between the value of 0 and 1, where 1 represents the ADC at maximum value (1024 or 5V). We have to convert the ADCresult variable to a float first, therefore inform the compiler to do this, using (float).
3. **Multiply the ADCScaling variable by the maximum Duty Cycle value from earlier, 596.** This does the same task as earlier but is much simpler; a ADCScaling value of 0 gives 0% duty, 0.5 gives 50% duty, 1 gives 100% duty. **See '*Figure 63: Method of using Floats to perform ADC Scaling*' for the full code.**
4. **Run the debug again and observe the operation.**

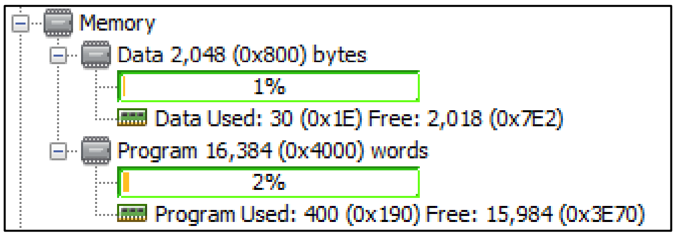

```
Memory
  Data 2,048 (0x800) bytes
    1%
    Data Used: 30 (0x1E) Free: 2,018 (0x7E2)
  Program 16,384 (0x4000) words
    2%
    Program Used: 400 (0x190) Free: 15,984 (0x3E70)
```

Figure 61: Memory Consumption Without Floats

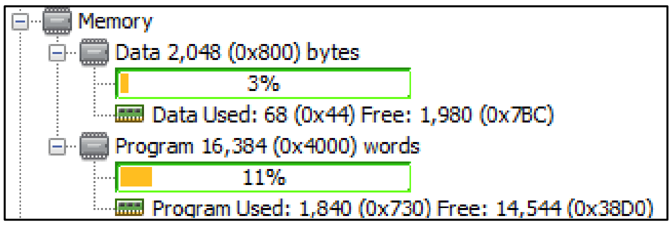
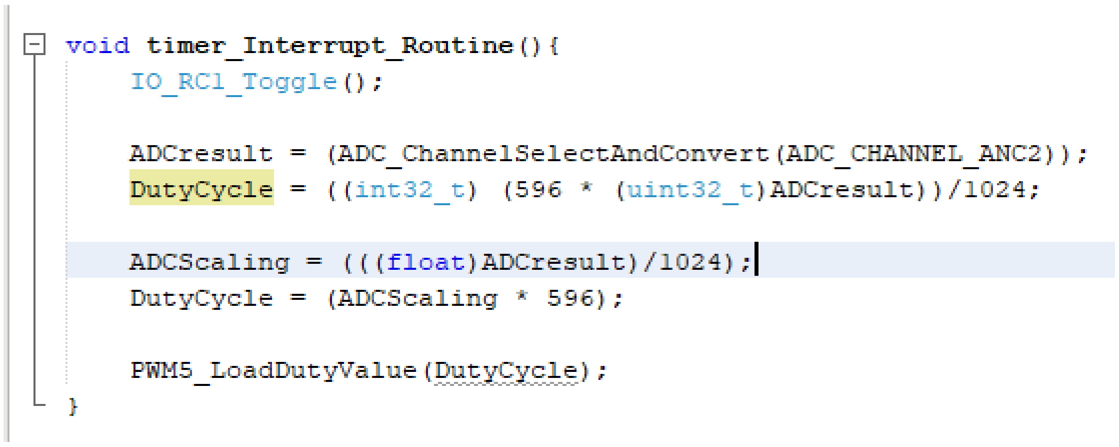

```
Memory
  Data 2,048 (0x800) bytes
    3%
    Data Used: 68 (0x44) Free: 1,980 (0x7BC)
  Program 16,384 (0x4000) words
    11%
    Program Used: 1,840 (0x730) Free: 14,544 (0x38D0)
```

Figure 62: Memory Consumption with Floats

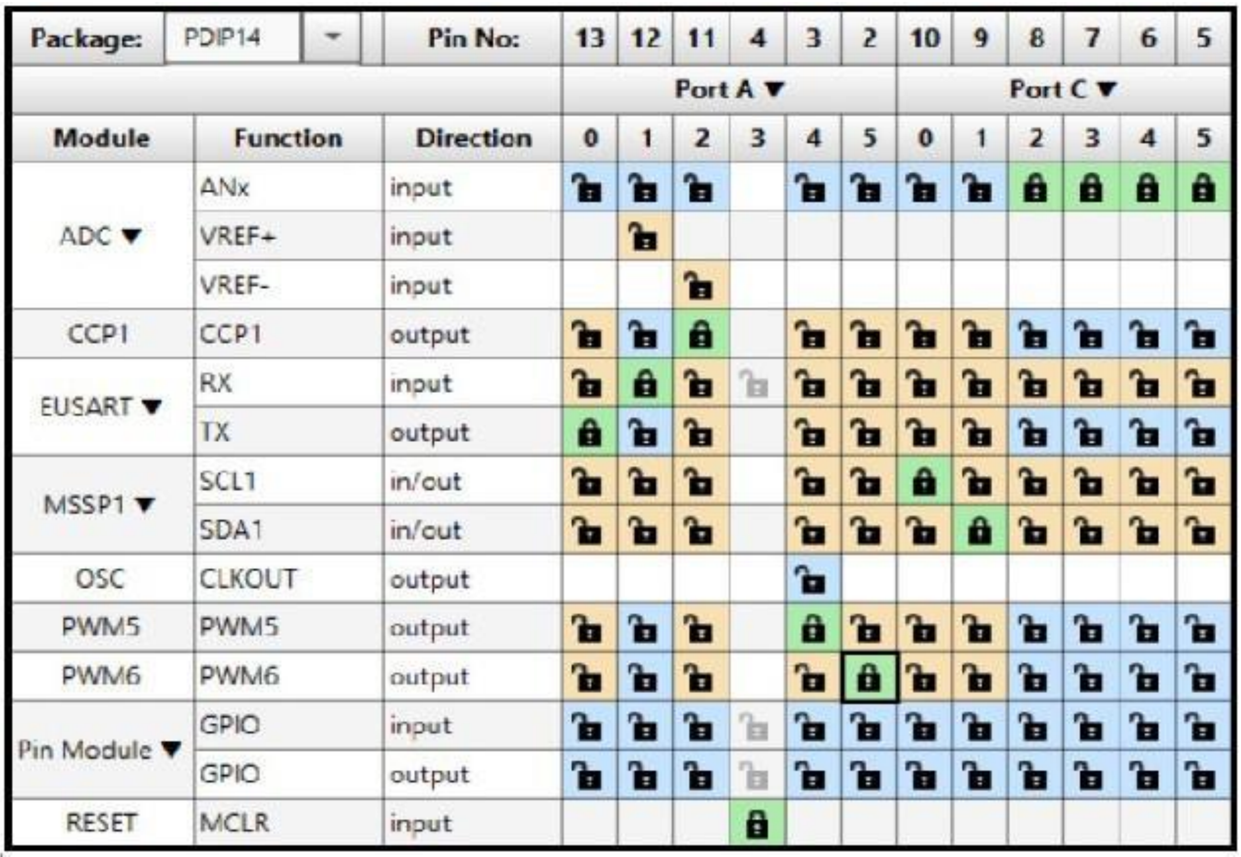


```
void timer_Interrupt_Routine(){
    IO_RC1_Toggle();

    ADCresult = (ADC_ChannelSelectAndConvert(ADC_CHANNEL_ANC2));
    DutyCycle = ((int32_t) (596 * (uint32_t)ADCresult))/1024;

    ADCScaling = (((float)ADCresult)/1024);
    DutyCycle = (ADCScaling * 596);

    PWM5_LoadDutyValue(DutyCycle);
}
```

Figure 63: Method of using Floats to perform ADC Scaling

Pinout for the Buggy Project

As part of the buggy project, we have a custom-built controller interface board, to connect the PIC up to the various subsystems and sensors as required. The PIC pinout allocation is therefore pre-determined. The figure below shows the pinout from the Pin Manager: Grid View. The second figure then shows the schematic of the interface board. You will need to setup the PIC using the MCC wizard according to the information provided below. Some of the setup examples correspond with the following diagrams: i.e a 200us PWM on RA4 was shown earlier. Header J15 is the programming header.



Package:	PDIP14	Pin No:	13	12	11	4	3	2	10	9	8	7	6	5
			Port A ▼						Port C ▼					
Module	Function	Direction	0	1	2	3	4	5	0	1	2	3	4	5
ADC ▼	ANx	input	🔓	🔓	🔓		🔓	🔓	🔓	🔓	🔒	🔒	🔒	🔒
	VREF+	input		🔓										
	VREF-	input			🔓									
CCP1	CCP1	output	🔓	🔓	🔒		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
EUSART ▼	RX	input	🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	TX	output	🔒	🔓	🔓		🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
MSSP1 ▼	SCL1	in/out	🔓	🔓	🔓		🔓	🔓	🔒	🔓	🔓	🔓	🔓	🔓
	SDA1	in/out	🔓	🔓	🔓		🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓
OSC	CLKOUT	output					🔓							
PWM5	PWM5	output	🔓	🔓	🔓		🔒	🔓	🔓	🔓	🔓	🔓	🔓	🔓
PWM6	PWM6	output	🔓	🔓	🔓		🔓	🔒	🔓	🔓	🔓	🔓	🔓	🔓
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
RESET	MCLR	input				🔒								

Figure 64: Pinout Allocation of PIC Controller Board

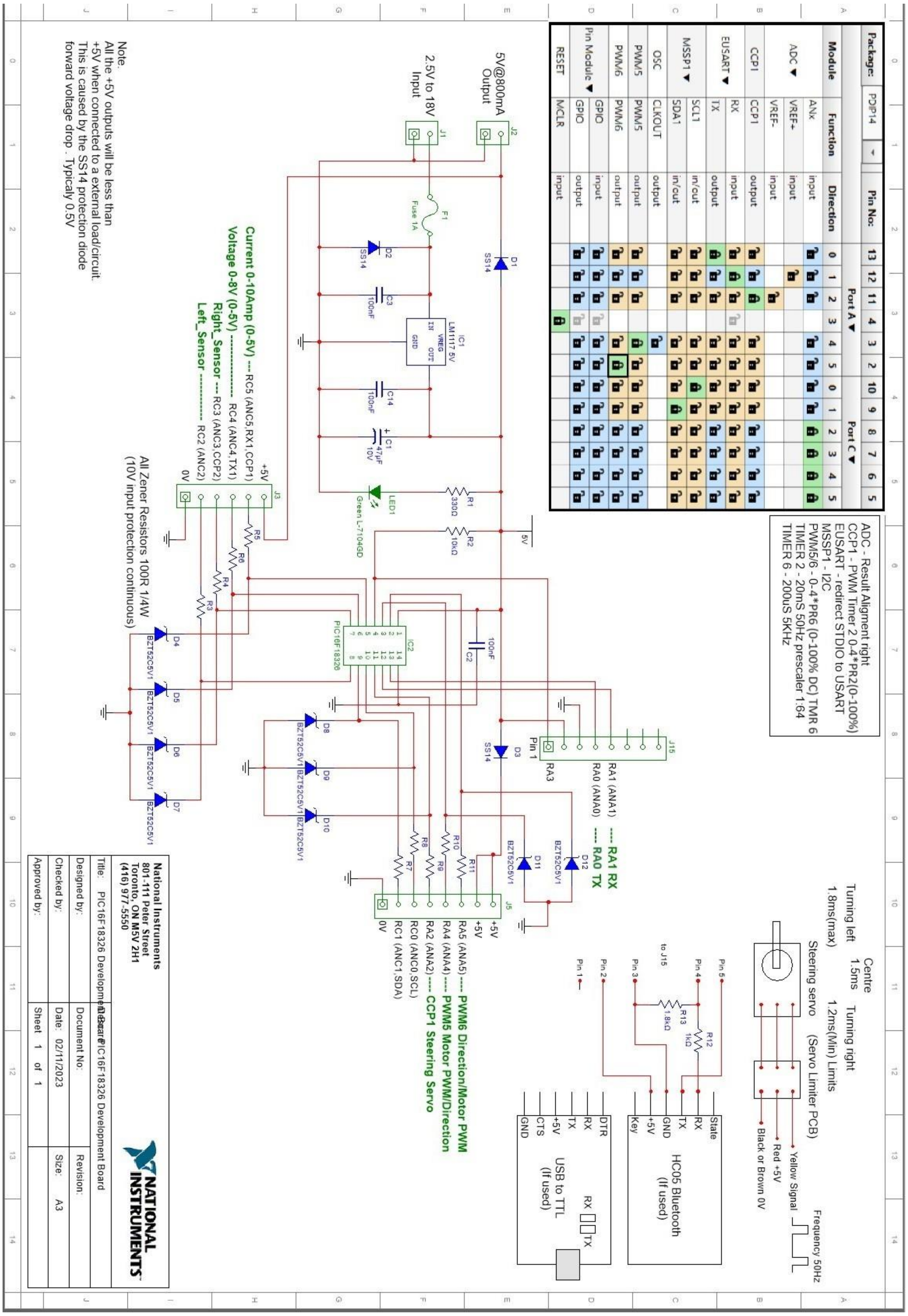

Package: PDIP14
Module / Function / Direction / Pin No: / Port A / Port C
ADC ANx input; VREF+ input; VREF- input
CCP1 CCP1 output
EUSART RX input; TX output
MSSP1 SCL1 in/out; SDA1 in/out
OSC CLKOUT output
PWM5 PWM5 output
PWM6 PWM6 output
Pin Module GPIO input; GPIO output
RESET MCLR input
13 12 11 4 3 2 10 9 8 7 6 5
0 1 2 3 4 5 0 1 2 3 4 5
ADC - Result Aligment right
CCP1 - PWM Timer 2 0-4*PR2(0-100%)
EUSART - redirect STDIO to USART
MSSP1 - I2C
PWM5/6 - 0-4*PR6 (0-100% DC) TMR 6
TIMER 2 - 20mS 50Hz prescaler 1:64
TIMER 6 - 200uS 5kHz
5V@800mA Output
2.5V to 18V Input
J1
J2
F1 Fuse 1A
D1 SS14
D2 SS14
C3 100nF
IC1 LM1117 5V VREG IN OUT GND
C14 100nF
C1 47µF 10V
R1 330Ω
LED1 Green L-7104GD
R2 10kΩ
5V
Current 0-10Amp (0-5V) --- RC5 (ANC5,RX1,CCP1)
Voltage 0-8V (0-5V) ---------------- RC4 (ANC4,TX1)
Right_Sensor --- RC3 (ANC3,CCP2)
Left_Sensor ----------- RC2 (ANC2)
+5V
0V
J3
R3 R4 R5 R6
All Zener Resistors 100R 1/4W
(10V input protection continuous)
IC2 PIC16F18326
C2 100nF
D4 D5 D6 D7 BZT52C5V1
D8 D9 D10 BZT52C5V1
D3 SS14
J15
Pin 1
RA3
RA1 (ANA1) ---- RA1 RX
RA0 (ANA0) ---- RA0 TX
D11 BZT52C5V1
D12 BZT52C5V1
R7 R8 R9 R10 R11
J5
+5V
RA5 (ANA5) ---- PWM6 Direction/Motor PWM
RA4 (ANA4) ---- PWM5 Motor PWM/Direction
RA2 (ANA2) ---- CCP1 Steering Servo
RC0 (ANC0,SCL)
RC1 (ANC1,SDA)
0V
Note.
All the +5V outputs will be less than +5V when connected to a external load/circuit.
This is caused by the SS14 protection diode forward voltage drop. Typicaly 0.5V
National Instruments
801-111 Peter Street
Toronto, ON M5V 2H1
(416) 977-5550
NATIONAL INSTRUMENTS
Title: PIC16F18326 Development Board
Designed by:
Document No:
Revision:
Checked by:
Date: 02/11/2023
Size: A3
Approved by:
Sheet 1 of 1
Centre 1.5ms
Turning left 1.8ms(max)
Turning right 1.2ms(Min) Limits
Steering servo (Servo Limiter PCB)
Yellow Signal
Red +5V
Black or Brown 0V
Frequency 50Hz
Pin 5
Pin 4
Pin 3
Pin 2
Pin 1
to J15
R12 1kΩ
R13 1.8kΩ
State RX TX GND +5V Key
HC05 Bluetooth (If used)
DTR RX TX +5V CTS GND
USB to TTL (If used)
RX TX

Figure 65: Schematic of the PIC Controller Board