48441 Introductory Digital Systems – PIC Lab 1

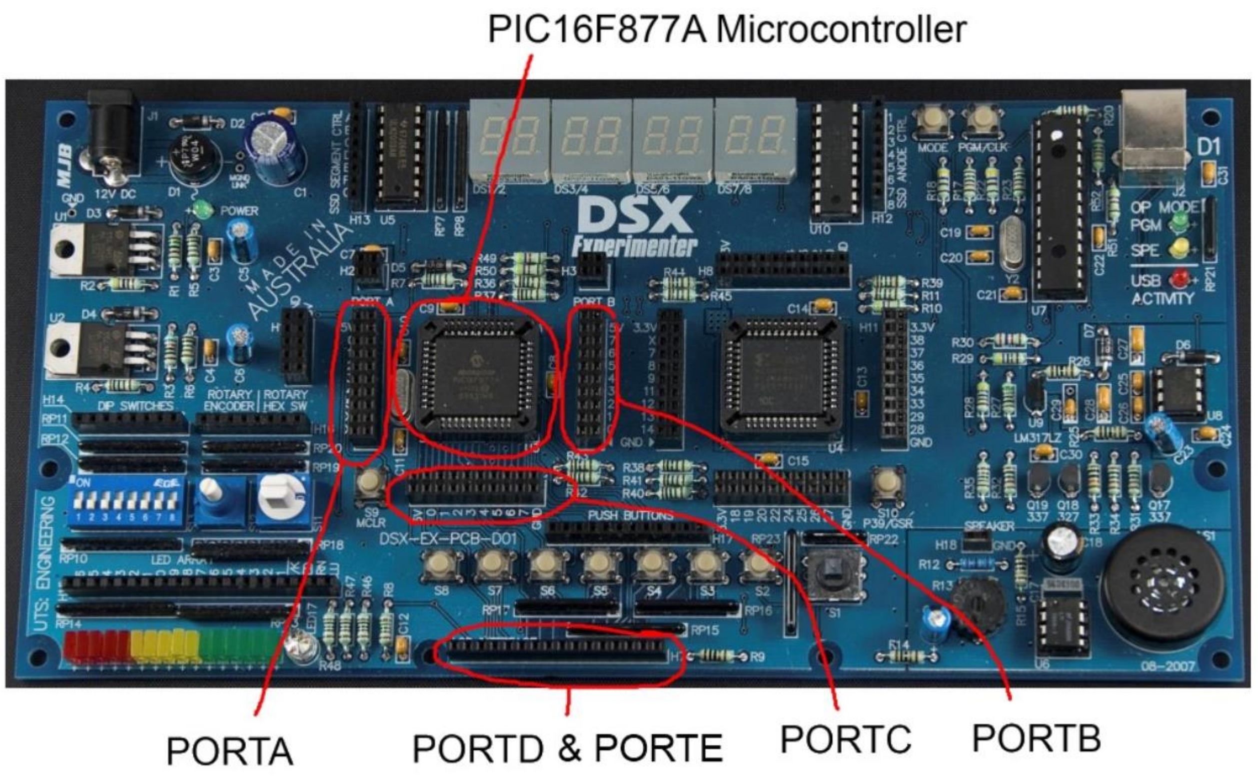
The PIC Microcontroller

- In the second part of the course, we are going to be using the PIC microcontroller on the DSX Kit.
- Where the CPLD uses schematics, the PIC uses MPLAB Assembly.
- The procedure is complex, so follow carefully and ask for help if you have issues.

Introduction to the PIC Microcontroller

The PIC16F877A is located on the left side of the DSX Kit.

Surrounding it are three headers labelled PORT A, PORT B and PORT C. Using the PIC we are able to set or read a '1' or '0' from each of the pins on each port. Some pins may also have extra functions.



To best understand the PIC16F877A

READ THE DATA SHEET

The ACTUAL Lab 1 Task

The first part of this lab is reference material for creating and debugging a project.

Our first lab is making a very basic calculator function: adding.

Our goal: Add two numbers, check for carry, and output the result to PORTB

We've broken this goal into 4 steps:

1) Take two 8 bit literal values (put them in “FIRST” & “SECOND”)
2) Add those values (in “ADDITION”)
3) Check if there is a carry (store in “CARRY”)
4) Output ADDITION to PORTB (ignoring the carry)

- If you haven't already, download the base code from UTSOnline and create a project as per guide above.
- The rest of this we'll demo in class, so make changes as we work through it together.

The *PIC16F877A Data Sheet*:

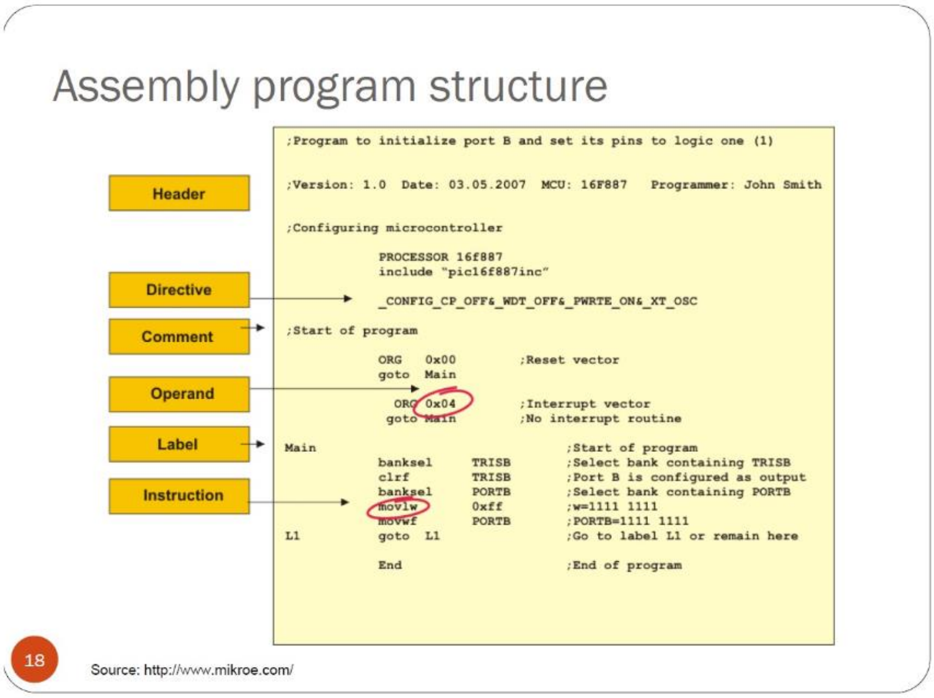
- Learn how to read it. It’s essential and you won’t be able to do well without it.
- Print out page 17, “PIC16F876A/877A REGISTER FILE MAP”. These are the memory locations we use when programming the PIC.
- Print out page 160, “PIC16F87XA INSTRUCTION SET”. These are the instructions we will use to program the PIC.
- I suggest printing them double sided and then laminating the page – you can refer to this when writing code.

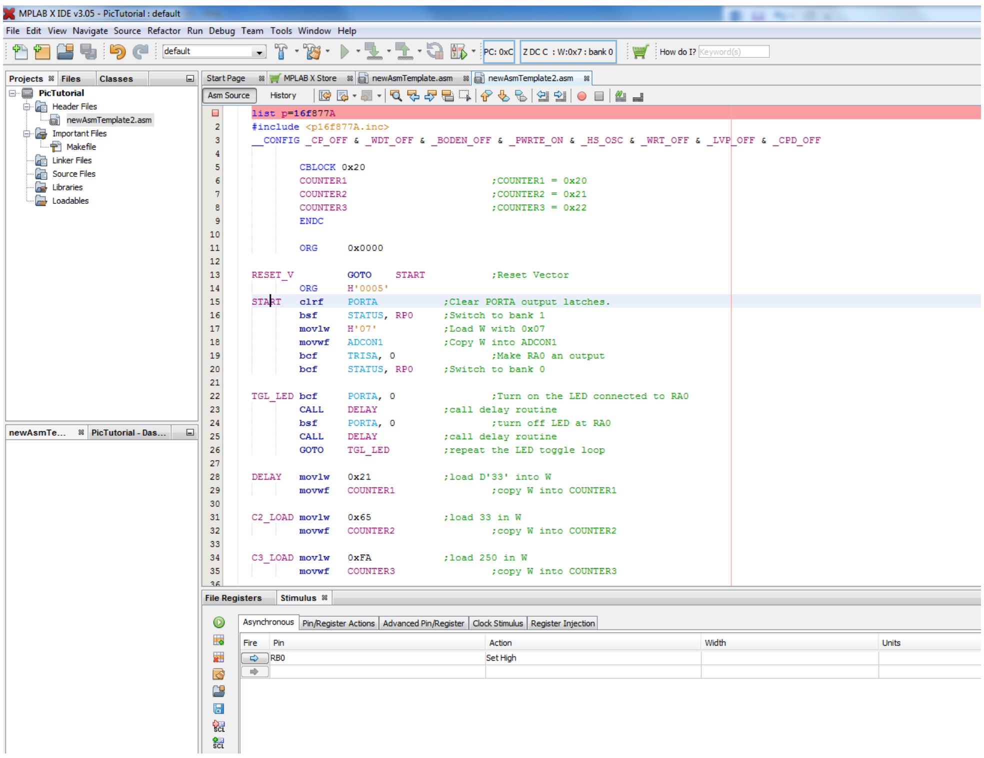
THE REST OF THE MATERIAL BELOW IS INCLUDED FOR YOUR REFERENCE

PIC Assembly

When we worked on the CPLD we made logic circuit schematics which we would eventually download to the CPLD on our DSX kit.

When working with the PIC we use a form of programming called Assembly, specifically the MPLAB syntax. We write the Assembly line by line, each line is given a number and performs a single command. When the PIC is powered on, it runs the command at line 0 (0x0000). Each command runs on a clock cycle to the microcontroller, except some commands which take two.





An Assembly line is made up of three parts, for example on **Line 13** we have; the Label (RESET_V), the Directive (GOTO) and the Operand (START).

The Label is a name we can assign a line in program memory, here RESET_V is a name we give Program Memory 0x0000. Any word at the start of a line is considered a Label.

The Directive is one of the many operations the Microcontroller can do. In this case GOTO changes the next command to be run to one specified. Other examples of directives are CLRF, which clears an 8-bit memory location, and MOVLW, which places a literal value in the working register.

The Operands are a comma separated list of values to specify what the directive should do. Here the Operand is “START”, which means the label START on line 10. GOTO in this case will make line 10, Program Memory location 0x05 the next line to execute, instead of 0x01.

But what about the other stuff?

Anything after a semicolon in an Assembly file is considered a “comment”. Comments and blank lines are ignored.

The first three lines should always be present in a PIC16F877A Assembly file, they tell the compiler which controller to use and what commands are available.

ORG and **END** are compiler commands and won't take up Program Memory. **ORG** tells the compiler that the next line has a specific memory location. **END** informs the compiler that the file has ended, and nothing after it should be considered.

PIC16F877A Memory

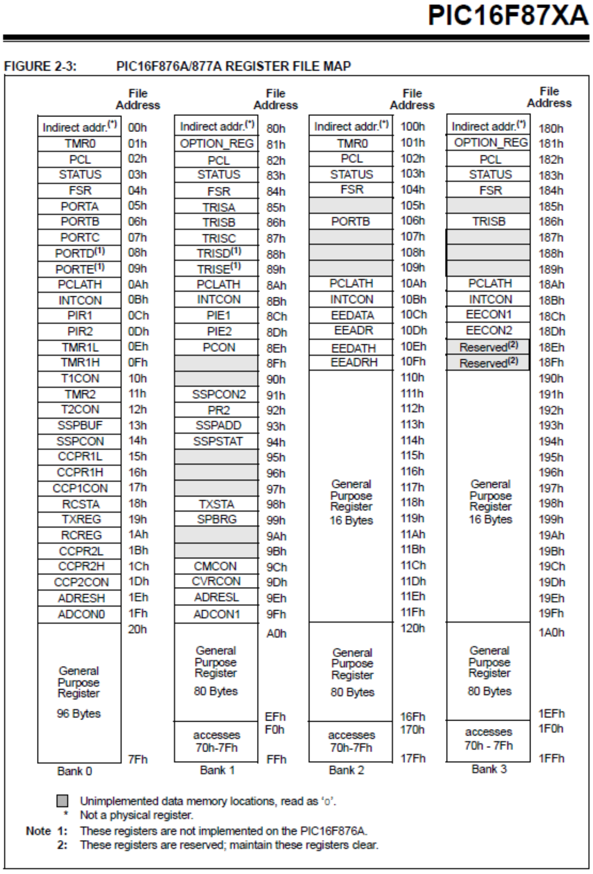
The PIC unit has 512 memory locations, each one holding 8-bits of data.

512 memory locations means we need a 9-bit number to hold the address (29 = 512).

But we are limited to using 7-bit numbers as operands in assembly lines! To overcome this, the PIC flash memory is divided up into 4 “banks” which can be selected by setting two bits in the STATUS register (RP0 and RP1). 2 STATUS bits + 7 Operand bits = 9 address bits.

For example, if I wanted to write to the TRISB memory location, I would need to write to location 0x86 (**01**00 0110 in binary). This location is in bank 1, so I before can write to it I set RP0 = 1 and RP1 = 0 (Bank 01). Now I can write to memory location 0x6, which in bank 1 is TRISB. <u>The STATUS register is available in all banks.</u>

Registers up to memory location 0x20 (0x10 in banks 2 and 3), have reserved purposes, like the STATUS register. Reading and writing data to these registers is how we interact with other components on the board, and use functions on the PIC. These special registers can all be referred to by name, so don't worry about remembering their exact memory location.



PIC16F87XA

FIGURE 2-3: PIC16F876A/877A REGISTER FILE MAP

	File Address		File Address		File Address		File Address
Indirect addr.(*)	00h	Indirect addr.(*)	80h	Indirect addr.(*)	100h	Indirect addr.(*)	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h		107h		187h
PORTD(1)	08h	TRISD(1)	88h		108h		188h
PORTE(1)	09h	TRISE(1)	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch	EEDATA	10Ch	EECON1	18Ch
PIR2	0Dh	PIE2	8Dh	EEADR	10Dh	EECON2	18Dh
TMR1L	0Eh	PCON	8Eh	EEDATH	10Eh	Reserved(2)	18Eh
TMR1H	0Fh		8Fh	EEADRH	10Fh	Reserved(2)	18Fh
T1CON	10h		90h	General Purpose Register 16 Bytes	110h	General Purpose Register 16 Bytes	190h
TMR2	11h	SSPCON2	91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch	CMCON	9Ch		11Ch		19Ch
CCP2CON	1Dh	CVRCON	9Dh		11Dh		19Dh
ADRESH	1Eh	ADRESL	9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Register 96 Bytes	20h	General Purpose Register 80 Bytes	A0h	General Purpose Register 80 Bytes	120h	General Purpose Register 80 Bytes	1A0h
			EFh		16Fh		1EFh
		accesses 70h-7Fh	F0h	accesses 70h-7Fh	170h	accesses 70h - 7Fh	1F0h
	7Fh		FFh		17Fh		1FFh
Bank 0		Bank 1		Bank 2		Bank 3	

Unimplemented data memory locations, read as ‘0’.
* Not a physical register.
Note 1: These registers are not implemented on the PIC16F876A.
2: These registers are reserved; maintain these registers clear.

FIGURE 1: PAGE 17, PIC MANUAL

The registers past this point can be used to store data in the regular course of your program.

The W Register

There is one register we can use that isn't present in the banks. This is the 'W' or working register. In the example above we used the **MOVLW** command to put the number 7 into the working register. Most of the more complicated commands, such as addition and subtraction, can only be used on the value stored in the working register.

The standard workflow for most operations is to:

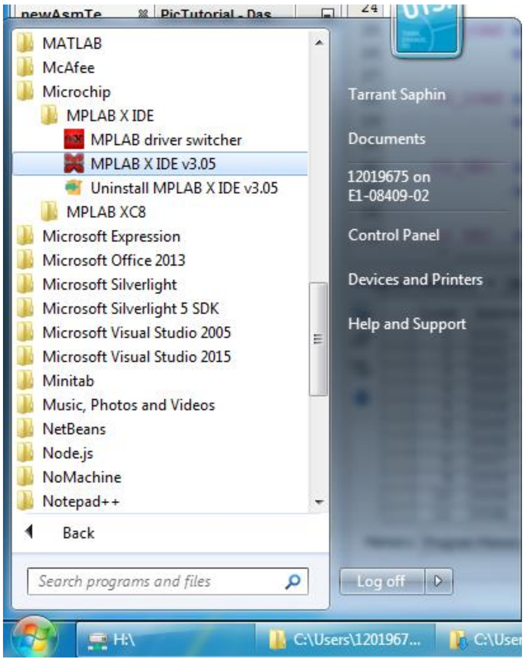
1. Copy a value into the working register
2. Do an operation on the working register
3. Copy the value out of the working register into a flash register.

Format of Literals [Hex, Decimal, Binary, Char]

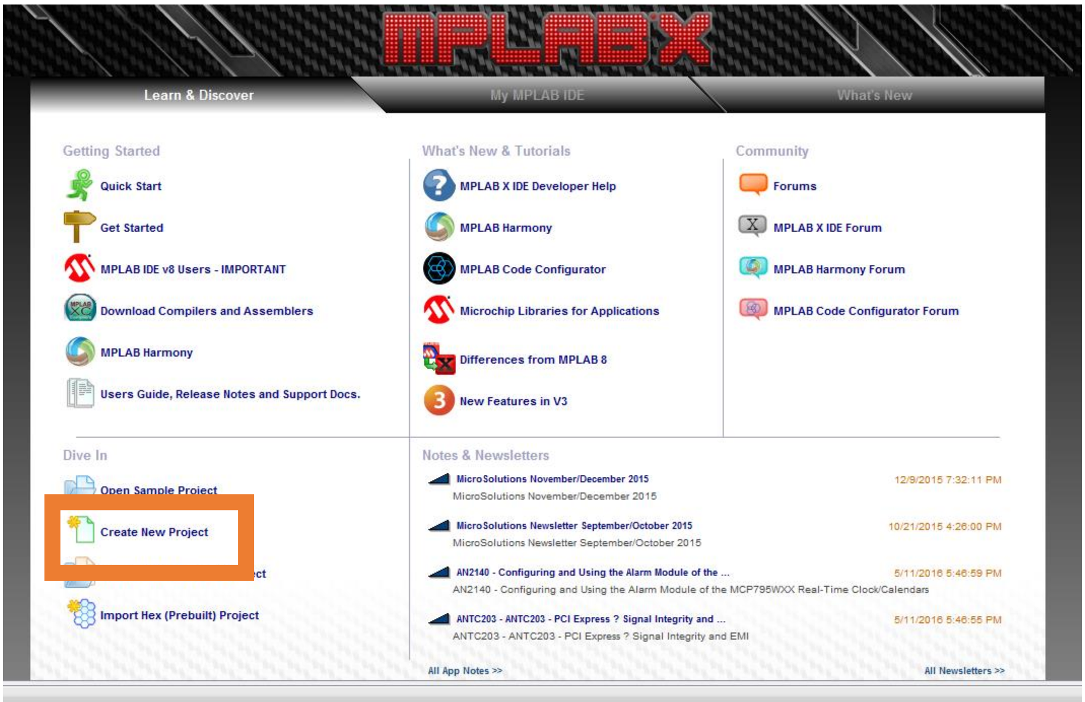
- Hex: 0x42 H’42’ h’42’ 42h - if a number begins with a letter use 0x, otherwise you can use either format
- Binary: b’10010110’
- Decimal: d’42’ .42
- Char: ‘H’ ‘e’ ‘L’ ‘l’ ‘O’

Creating an MPLAB X IDE Project

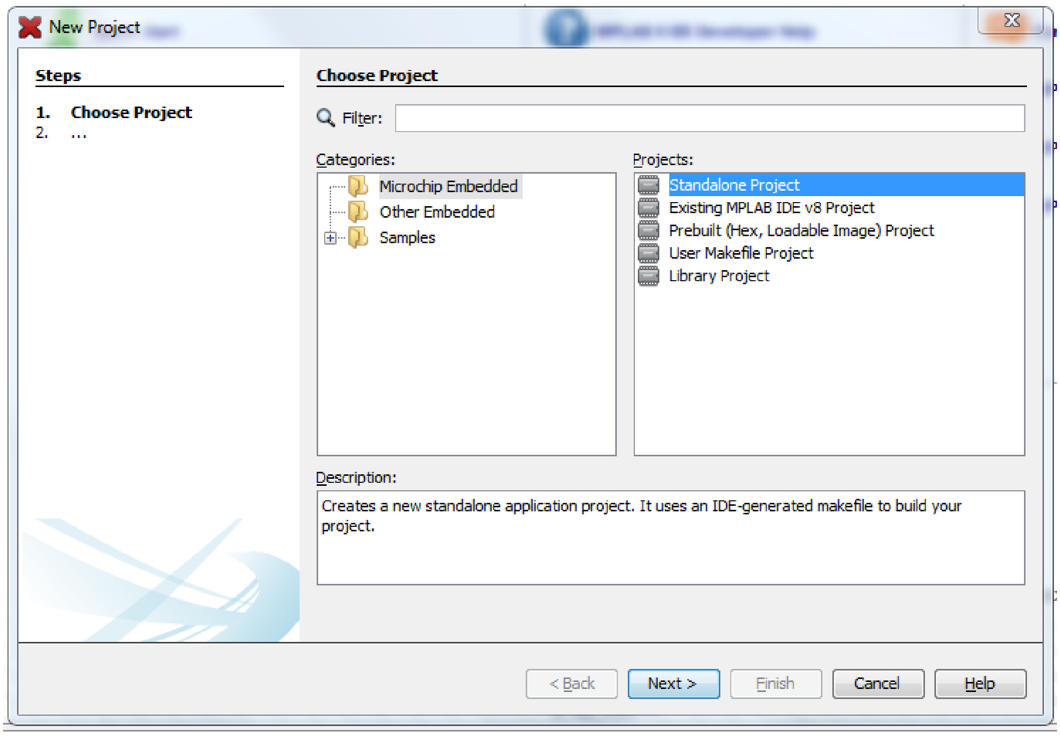
1. Open MPLAB X IDE from *Start* → *Microchip* → *MPLAB X IDE* (or other version)



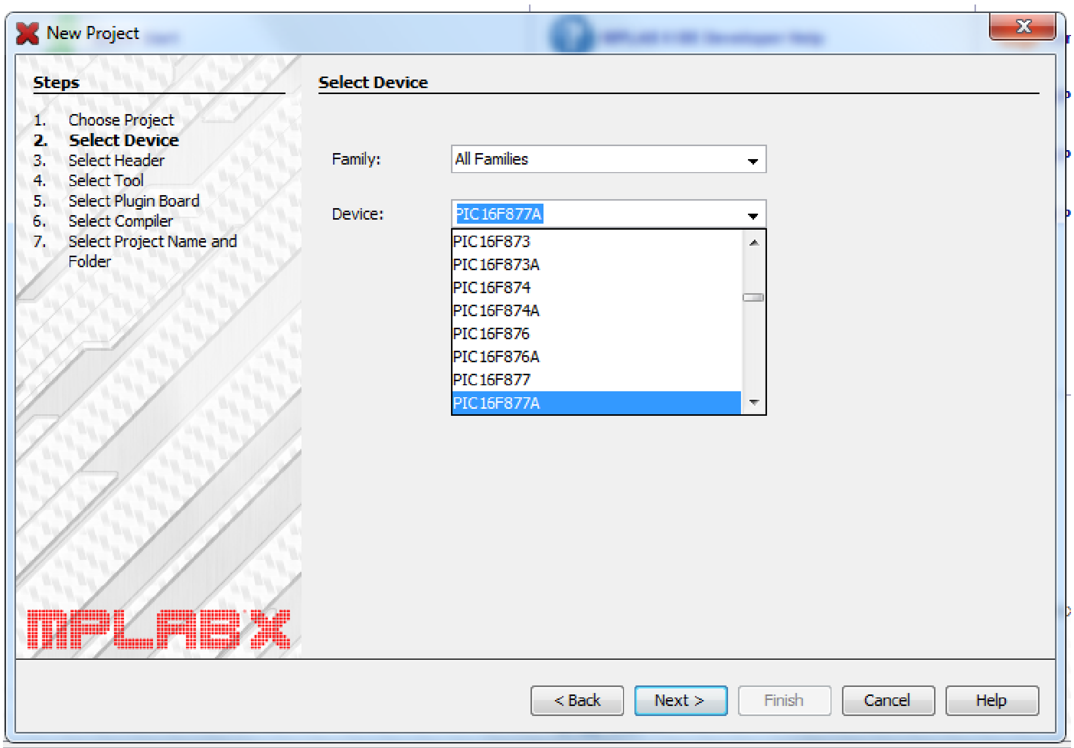
2. Using the Welcome Screen, select "Create New Project"



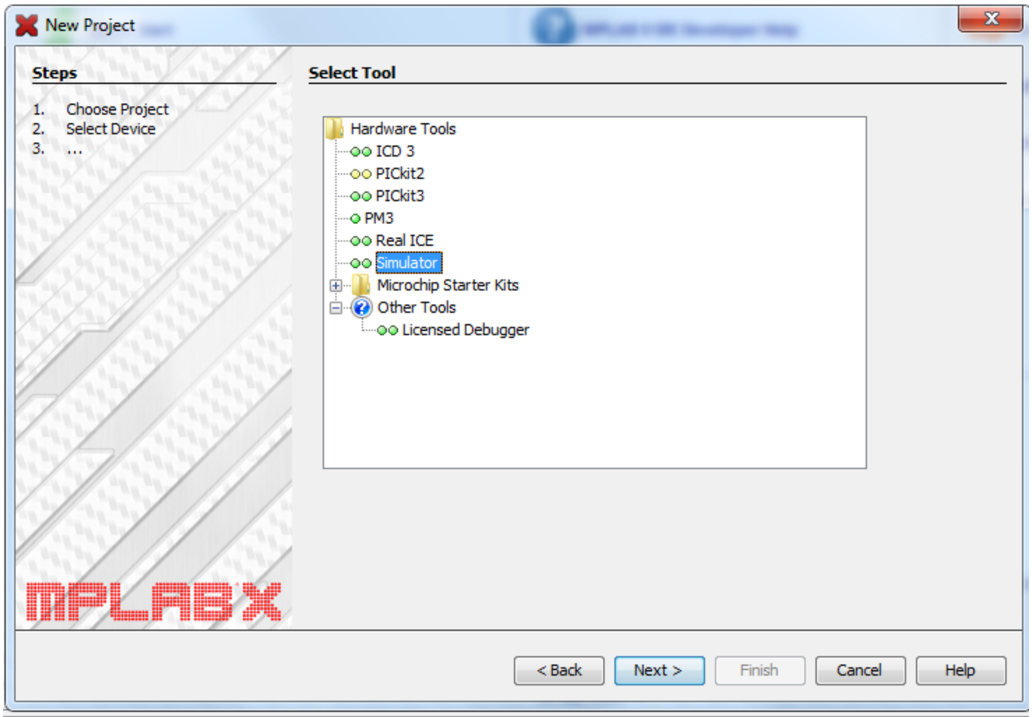
3. Under Choose Project, select Microchip Embedded -> Standalone Project, then click Next >



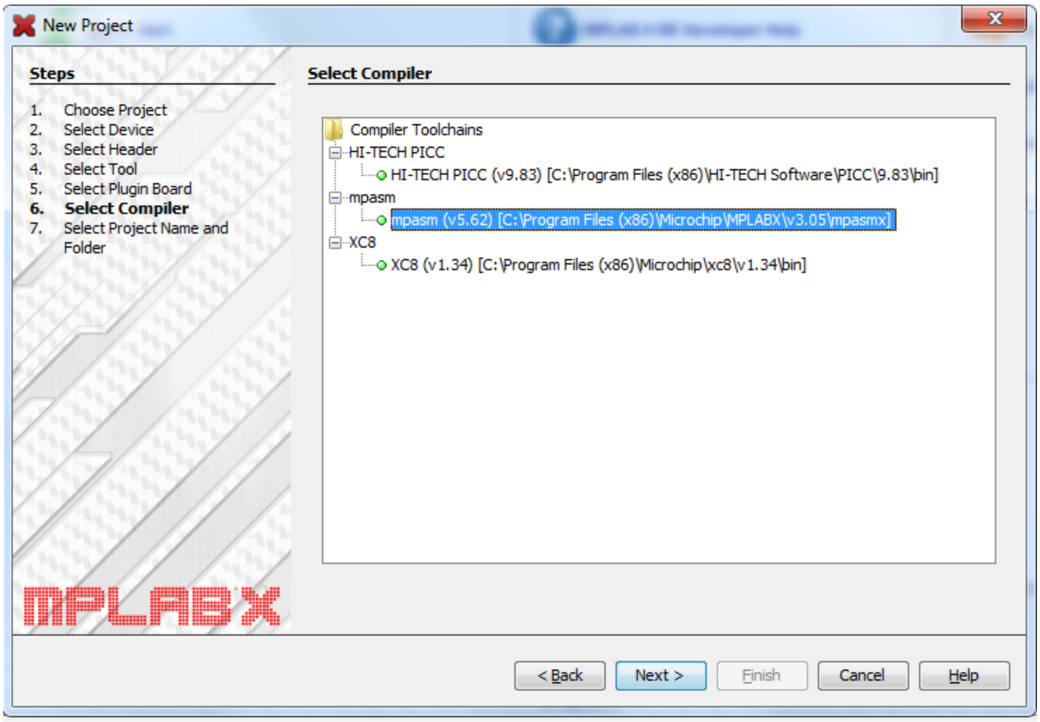
4. From the device list, select *PIC16F877A*, then click *Next >*



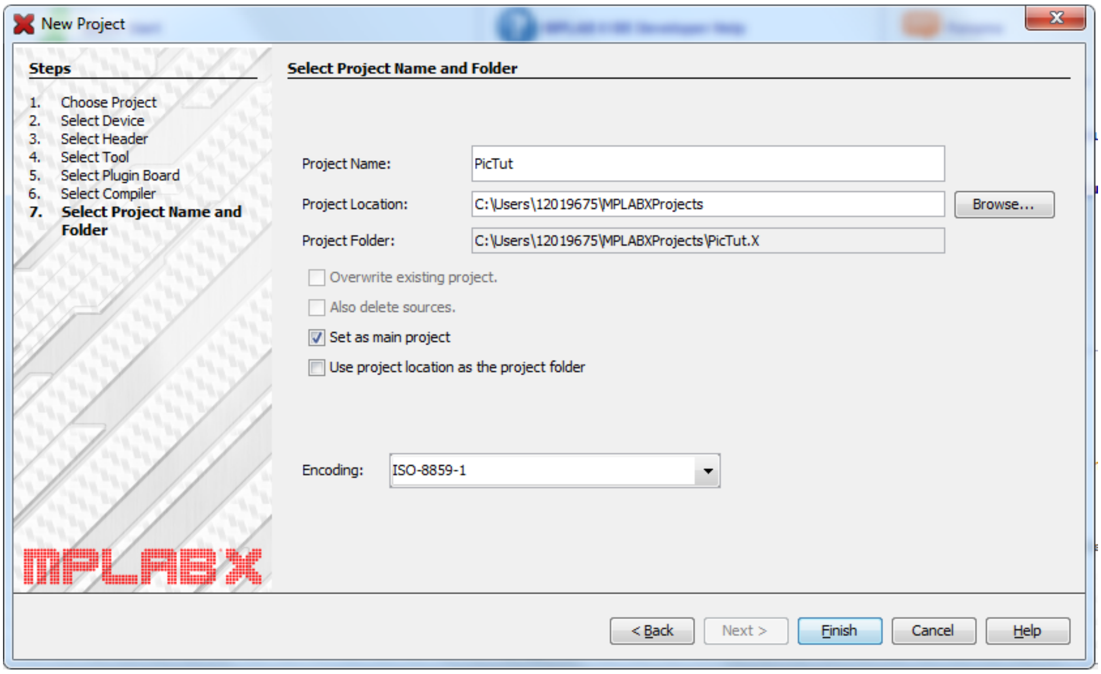
5. On the Select Tool Page, Select Hardware Tools -> Simulator, then click *Next >*



6. On the Select Compiler Page, select mpasm (v5.62), then click Next >

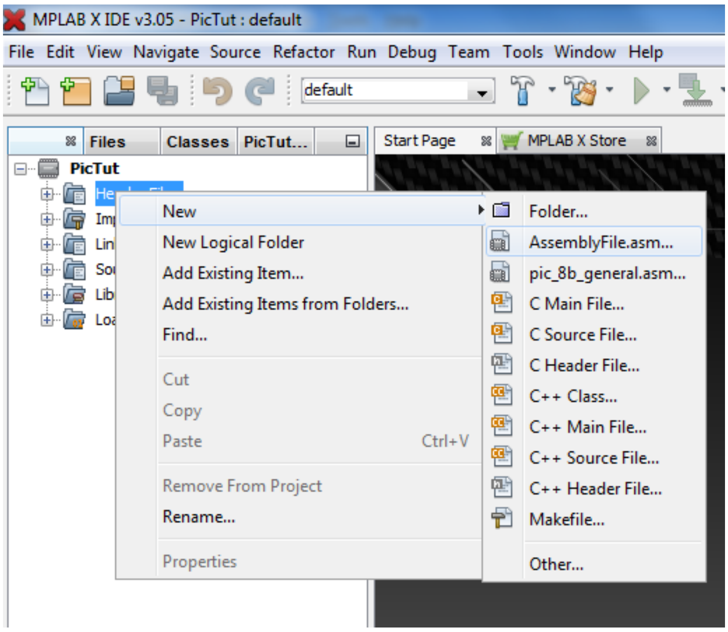


7. On this page, select a project location to save your project. You may choose to save to your USB drive or C:Drive [local computer]. And give your project a name. When you are done, click Finish.

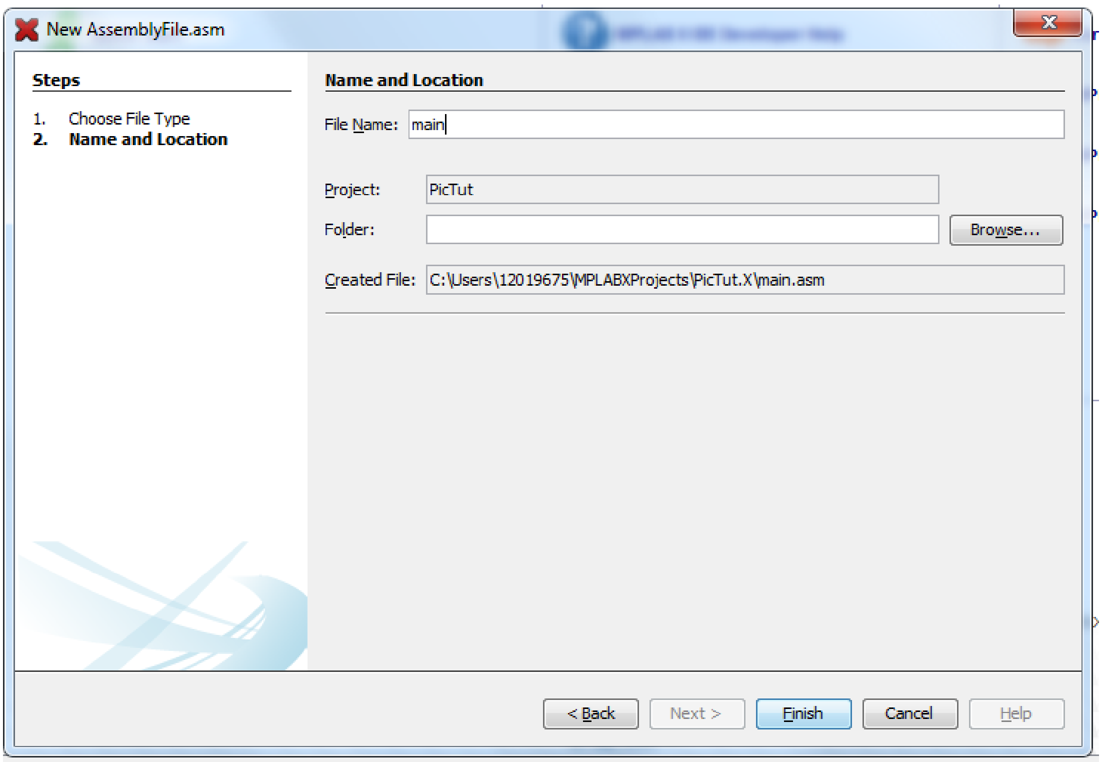


Adding files to our project

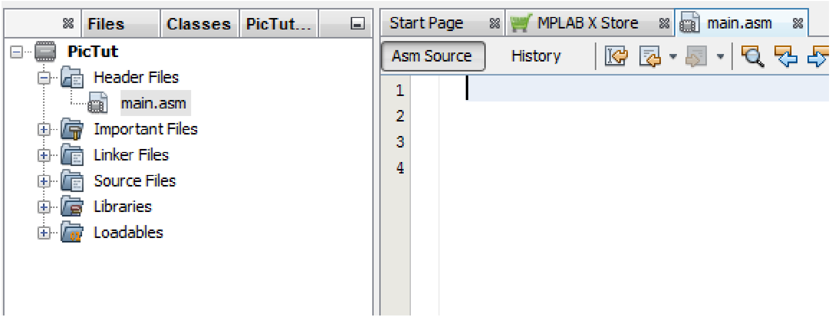
1. To add a new assembly source file to our project, On the left hand side, <u>**Right Click on the Header Files**</u> under your project, and click <u>*New->AssemblyFile.asm*</u>



2. Give your file a name under "File Name", then click Finish. In my example, I've typed in main.



3. You should see your asm file on the left hand side under Header Files.



4. You should now have a blank assembly file in your project

Some tips and a simple example

- The *PIC16F877A Data Sheet*:
  - Learn how to read it. It's essential and you won't be able to do well without it.
  - Print out page 17, "PIC16F876A/877A REGISTER FILE MAP". These are the memory locations we use when programming the PIC.
  - Print out page 160, "PIC16F87XA INSTRUCTION SET". These are the instructions we will use to program the PIC.
  - I suggest printing them double sided and then laminating the page – you can refer to this when writing code.
- Writing code in MPASM:
  - In general, most instructions aren't case sensitive. That said, always capitalise your instructions for consistency – port names are always capitalised.
  - Commenting is key – use a single semicolon to make the rest of the line a comment, so you can explain what you're doing.
  - Use the TAB key to separate your columns of code, not spaces.
  - CTRL+SHIFT+C will comment/uncomment lines
- **If you need it, the first three lines for a program are below:**

```
list p=16f877A
#include <p16f877A.inc>
__CONFIG _CP_OFF & _WDT_OFF & _BODEN_OFF & _PWRTE_ON & _HS_OSC & _WRT_OFF &
_LVP_OFF & _CPD_OFF
```

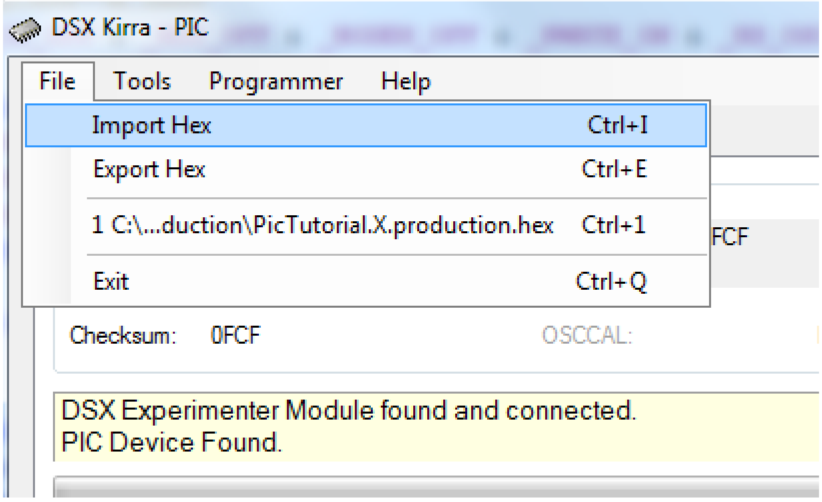
Programming your DSX Kit PIC using DSX Kirra

Using MPLAB, every time you build your program successfully, MPLAB will create a .hex file that you can upload to your DSX Kit using KSX Kirra

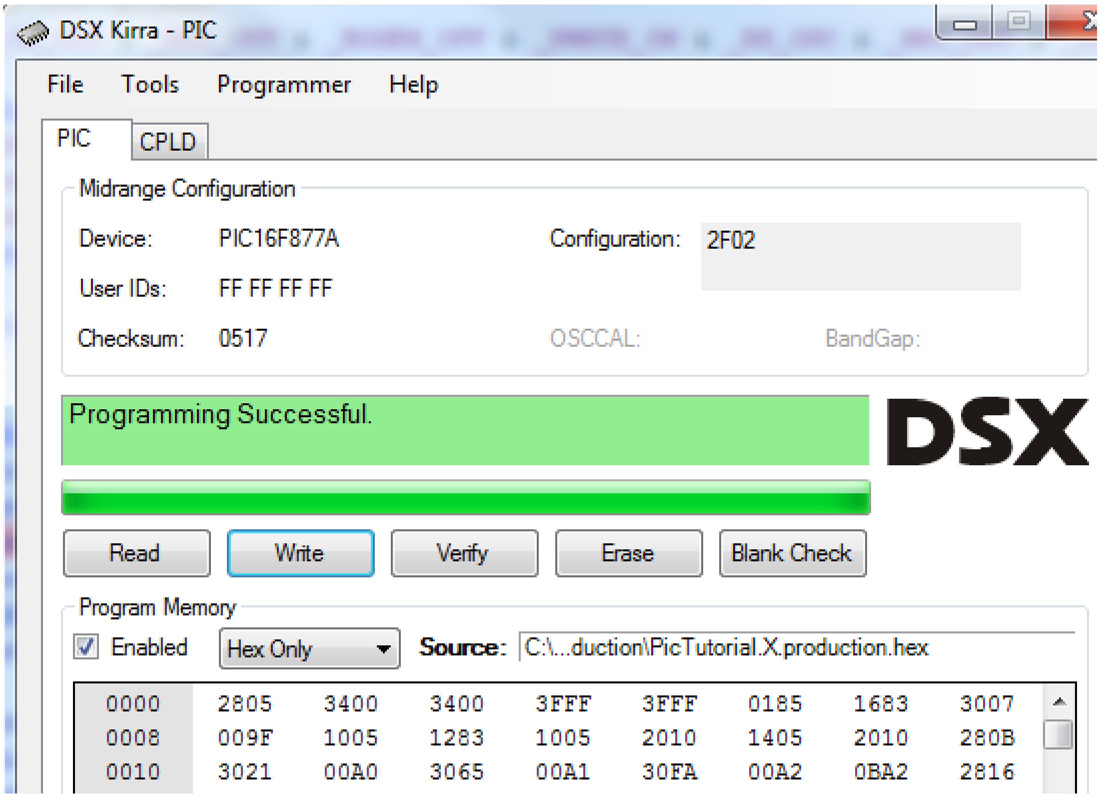
On a successful build you will see the following message

```
BUILD SUCCESSFUL (total time: 102ms)
Loading code from C:/Users/12019675/MPLABXProjects/PicTutorial.X/dist/default/production/PicTutorial.X.production.hex...
Loading completed
```

If you goto that directory on your computer and upload that hex file into DSX Kirra



Then Click Write to upload your hex file to your DSX Kit

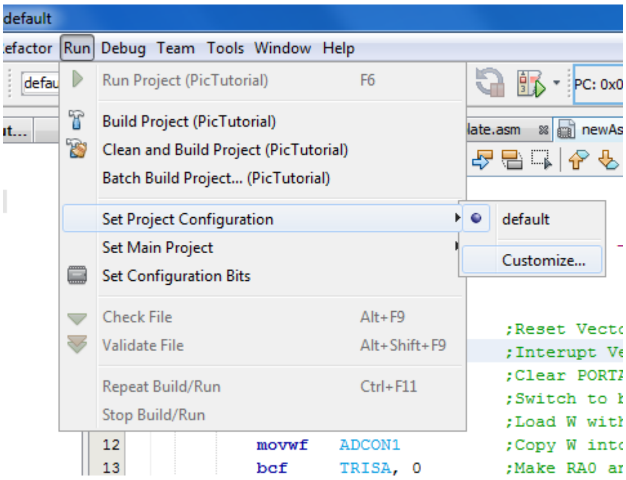


MPLAB X IDE Simulation

MPLAB X IDE comes with a powerful simulator that lets you run your program line by line to follow the logic, watch changes in the file registers and stimulate inputs to the ports.

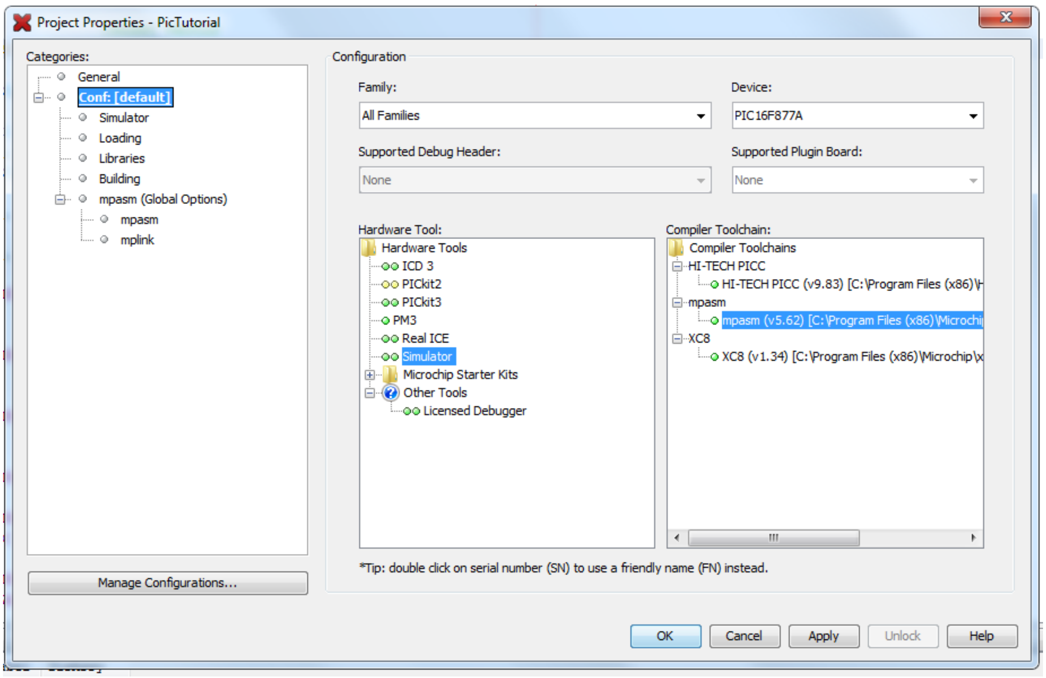
Before simulating make sure that you have a project open and your source file is shown in the project window, else the simulator won't know which file to run.

First, make sure that you have select the correct debugging tool, by going to Run->Set Project Configuration->Customize



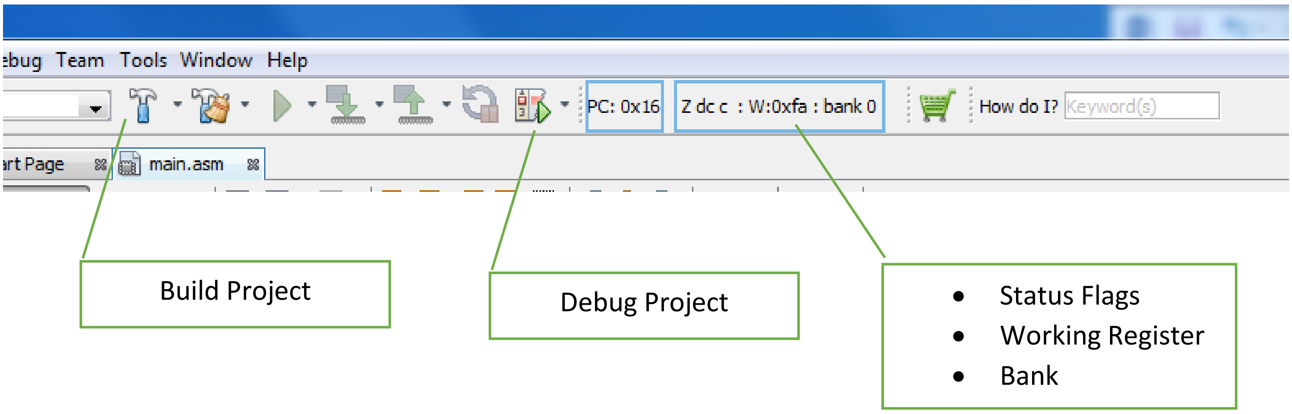
On the Configuration Screen, make sure you have the following settings

- Device: PIC16F877A
- Hardware Tool: Simulator
- Compiler Toolchain: mpasm



Program Counter

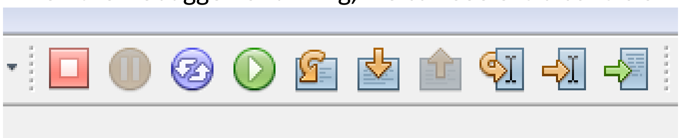
We now have some controls at the top of the screen.



These controls are:

Build Project	Build the project and create a .hex file – *Upload to your DSX Kit*
Debug Project	Debug your project using MPLAB – Step through your project
Program Counter	Used during debug – Show the current value of the Program Counter
Status Flags	Used during debug – Shows the current status flags
Working Register	Used during debug – Shows the current value of the Working Register
Bank	Used during debug – Shows the current bank

When the Debugger is running, we can see extra controls



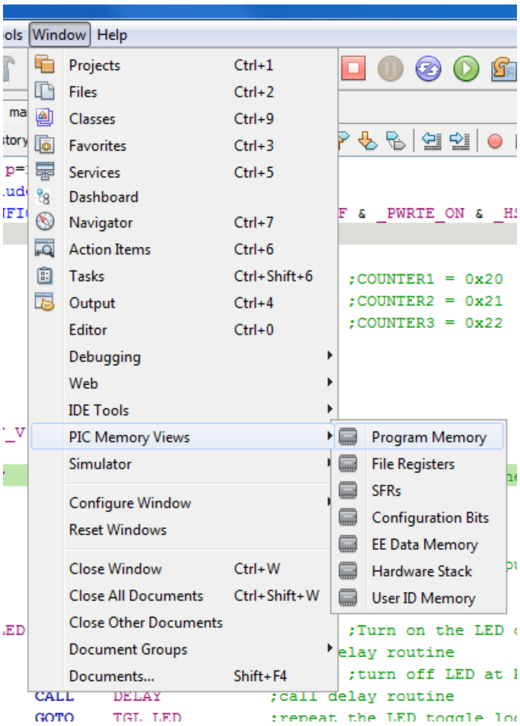
There controls are from left to right

Stop	End the simulation
Pause	Pause the simulation
Reset	Reset the simulation
Continue	Continue the simulation from a Pause
Step Over	While Paused, Step over a function – Not used
Step Into	While Paused, continue to the next line
Step Out	While Paused, Step out of a function – Not used
Run to Cursor	Run till the program reaches where your cursor is and pause
Set PC at Cursor	Sets the Program Counter to where the cursor is
Focus Cursor at PC	Moves your cursor to the current location of the Program Counter

Viewing Memory

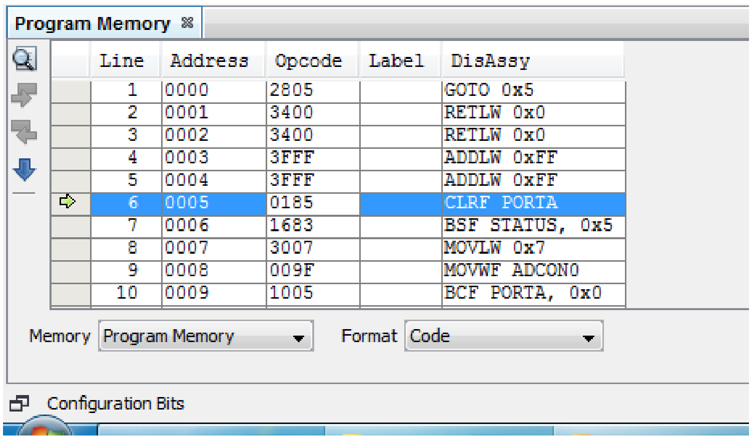
We can now use the debugger to view the memory blocks on the PIC in real-time. To view them, go the Window Menu->PIC Memory Views. We will be using the:

- Program Memory
- SFRs - Special Function Registers
- File Registers



Program Memory

Once the project is compiled we can see the how our assembly is written into the program memory on the DSX Kit by selecting Program Memory from the View menu:



Special Function Registers

When we are simulating we can watch the special function registers as they update:

The green arrow next to a line indicates it is the next line to be execute

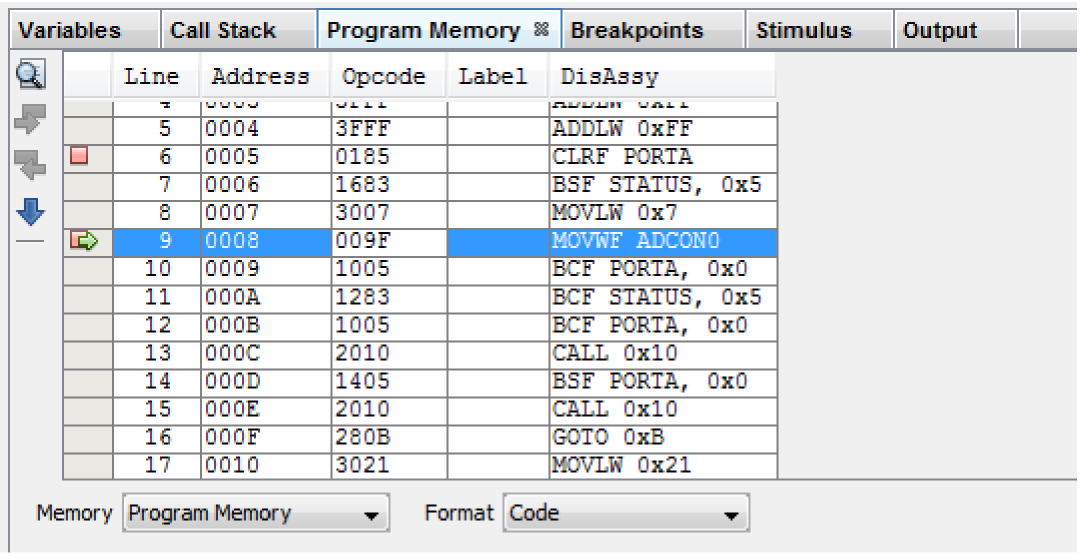
The Red Square indicates a breakpoint, can be toggled by double clicking

Green line indicates the next line to be executed

Red Line Indicate the breakpoint

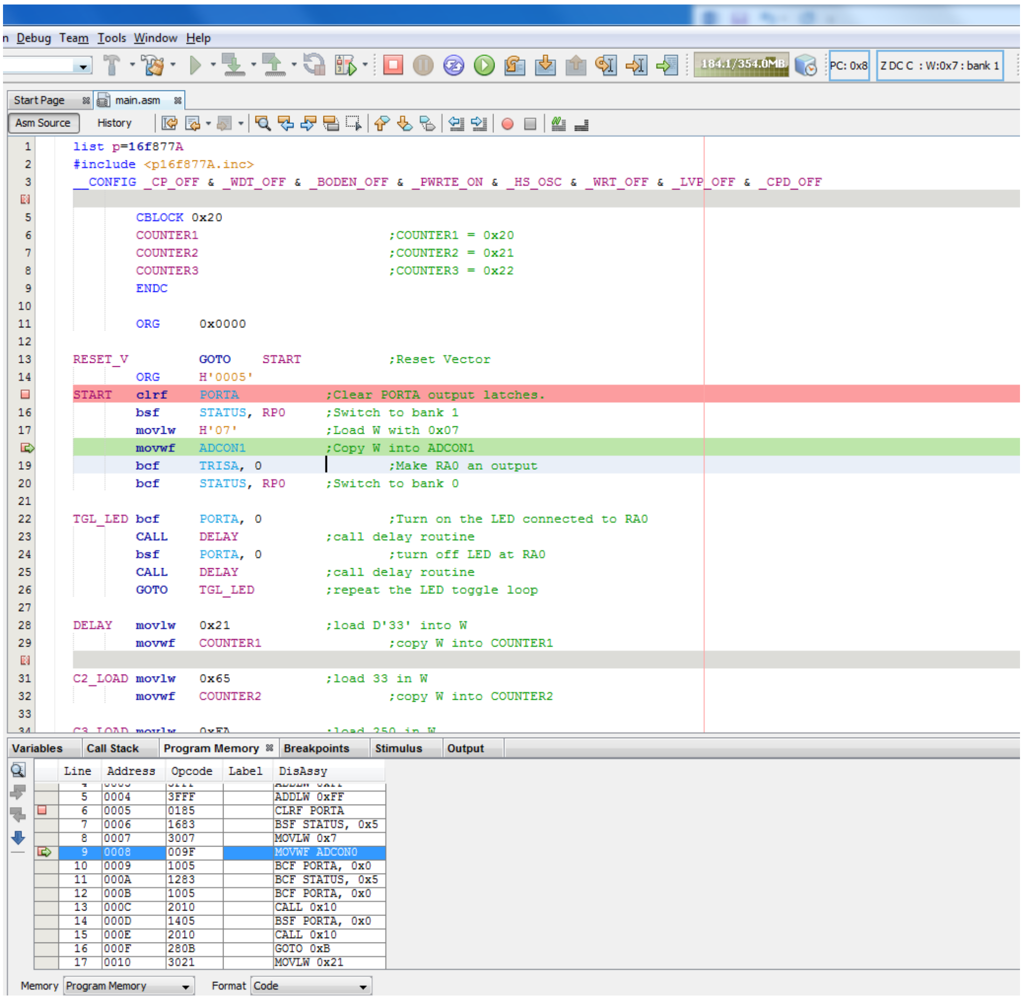
The Special Function register window is opened by clicking 'Special Function Registers' from the View menu.

This shows all the named registers, their address, and their value in hexadecimal. We can see the value in decimal and binary by right clicking on the column headings and ticking the appropriate option.



Line	Address	Opcode	Label	DisAssy
5	0004	3FFF		ADDLW 0xFF
6	0005	0185		CLRF PORTA
7	0006	1683		BSF STATUS, 0x5
8	0007	3007		MOVLW 0x7
9	0008	009F		MOVWF ADCON0
10	0009	1005		BCF PORTA, 0x0
11	000A	1283		BCF STATUS, 0x5
12	000B	1005		BCF PORTA, 0x0
13	000C	2010		CALL 0x10
14	000D	1405		BSF PORTA, 0x0
15	000E	2010		CALL 0x10
16	000F	280B		GOTO 0xB
17	0010	3021		MOVLW 0x21

In this example, execution is at line 18, and the W register currently holds a value of 0x07. When we click "step into", this line is executed: MOVWF loads the W register into ADCON1 (in this case, 0x07).

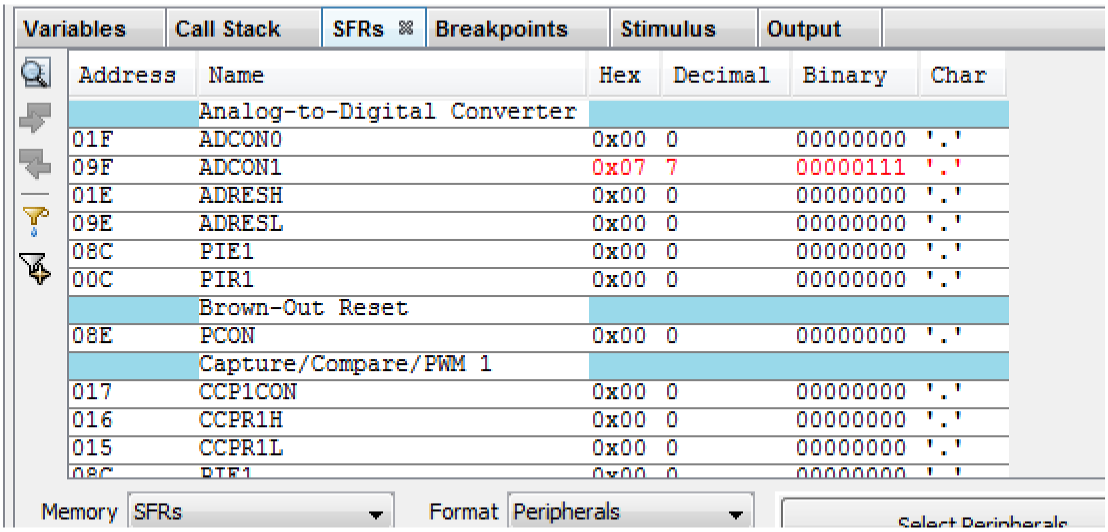


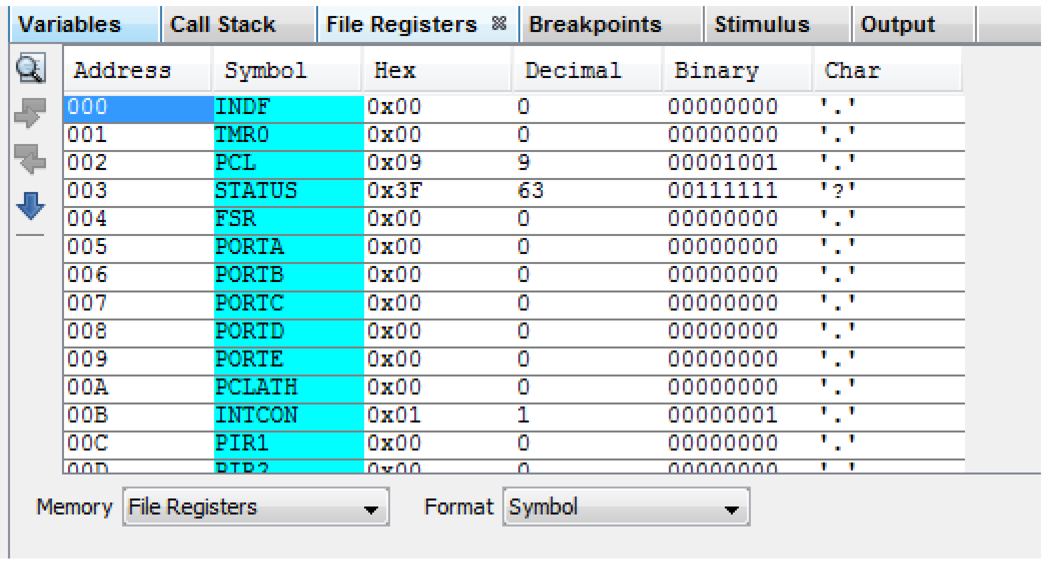
Execution is now at line 19, and ADCON1 now has a value of 0x07.

All File Registers

The Special Function Registers show us all the named registers, but if we want to view the general purpose registers we need to use the File Registers window.

When it is opened it will show a hexadecimal table. Each two digits will be a register, the columns and rows the memory address. We can see a more familiar and easier to read version by clicking “symbolic” in the lower left corner.





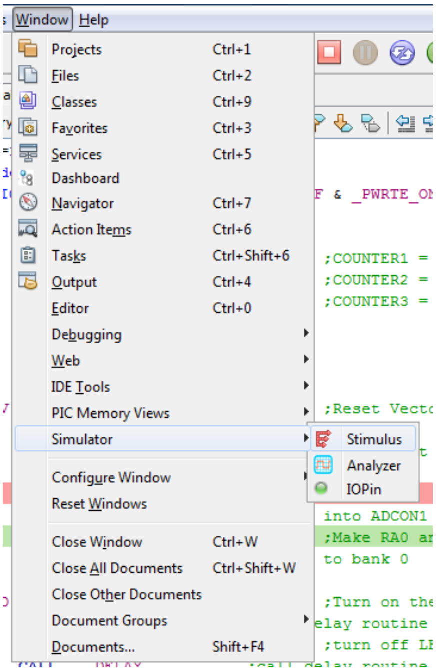
Here we can see the Address, Value and Symbol Name in a neat single column.

The General Purpose registers are available from 0x20 onwards in bank 0, 0xA0 onwards in bank 1, 0x110 in bank 2 and 0x190 in bank 3.

0x70 to 0x7H in bank 0 are copied across the last 16 registers in every bank.

Stimulation

Not only can we follow changes in the file registers, but we can make changes to them using the stimulus menu. To start testing inputs in your simulations, you will need to open the Stimulus window by going to Window->Simulator->Stimulus



In the Stimulus Workbook you're able to signal individual pins on demand from the Asynchronous tab. Select the Pin you wish to signal, the signal you wish to use, and the length of the signal. During simulation, click the fire button to send the signal.

“Set High/Low” sets the pin to a value until you change it, “Pulse High/Low” sets the pin to a value for a specified period. “Toggle” changes the value of the pin to it's opposite.

Here PORTB has been set-up as input, and a loop copies PORTB directly to the W register. We have two signals in our workbook, RB0 Set High, and RB1 Pulse High for 3 cycles. We have clicked fire on both signals.