EE216 **31 March 2010**

Tutorial on C Compiler

C Programming for PIC Microcontroller

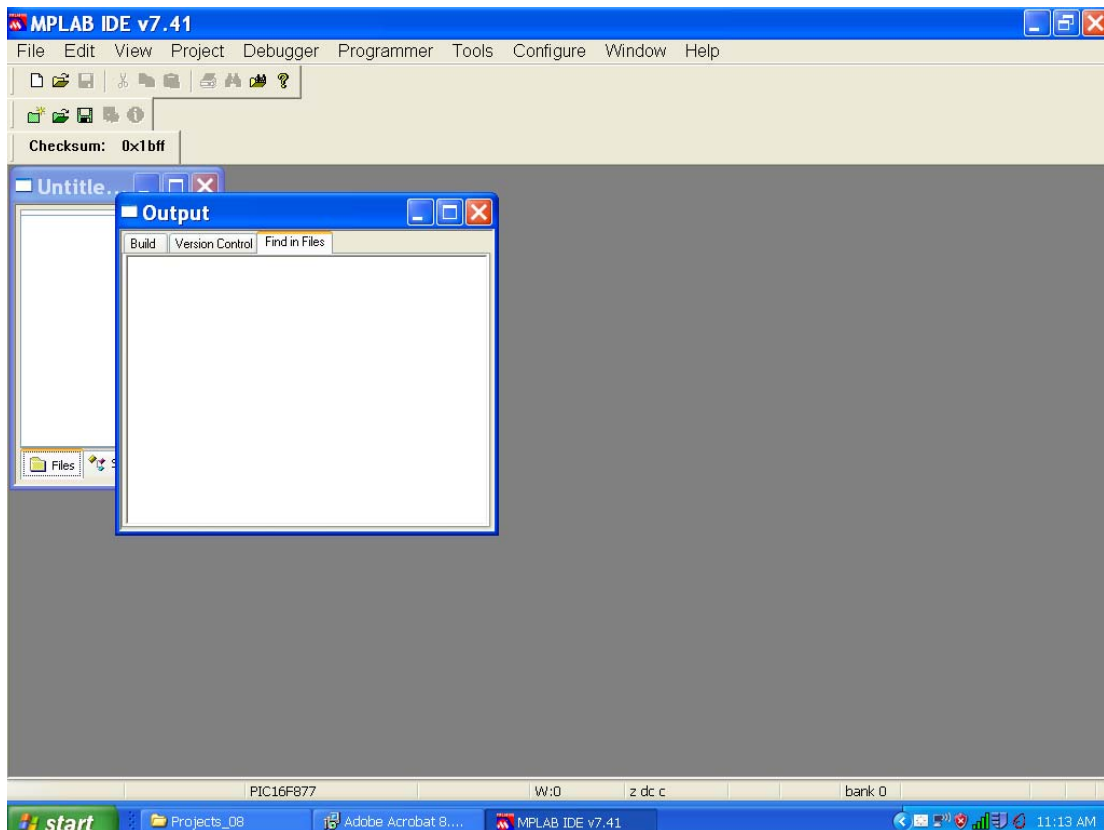
Introduction:

This tutorial will introduce you to using the C programming language for microcontroller applications. We will be using a freeware C compiler, called PICC Lite, supplied by a company called Hi-Tech. However, our code will still be written in MPLAB environment.

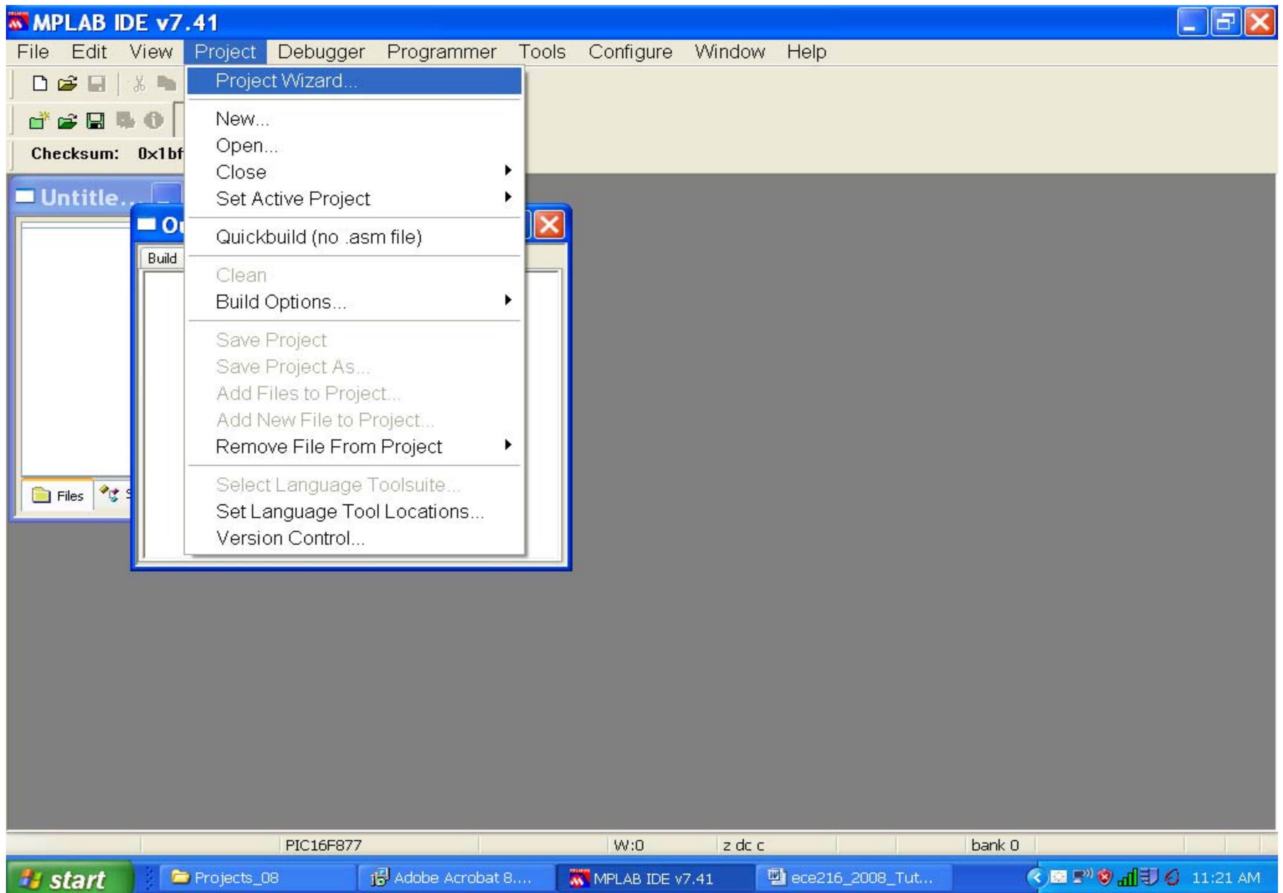
Part I: Using PICC Lite

1. PICC Lite is a free version of Hi-Tech's PICC compiler. It supports several but not all Microchip devices including the 16f84, 16F84A, 16F877, 16F877A, and the 12f675. There are some limitations, but nothing that will affect our coding. (Take a look at www.htsoft.com for more info.)

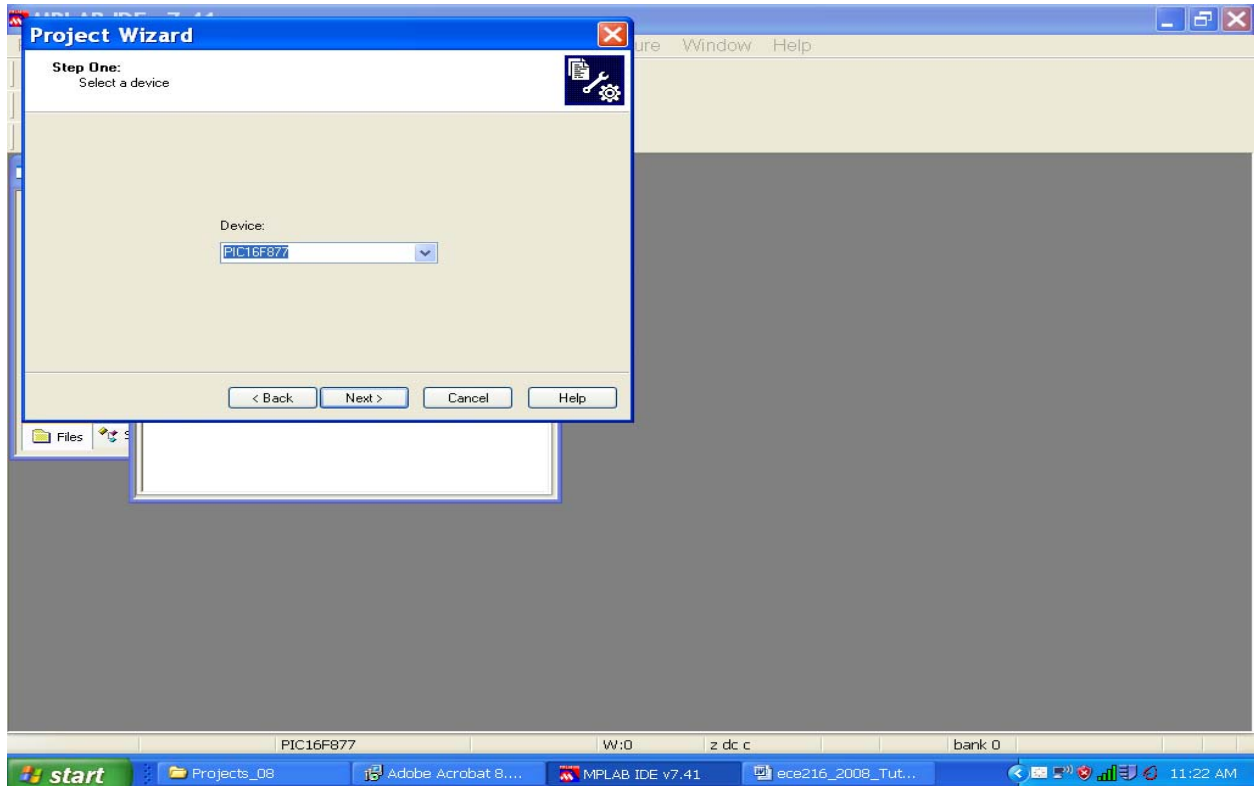
2. To use the compiler and write our code in C, we will still be using MPLAB IDE. Open the MPLAB Integrated Development Environment



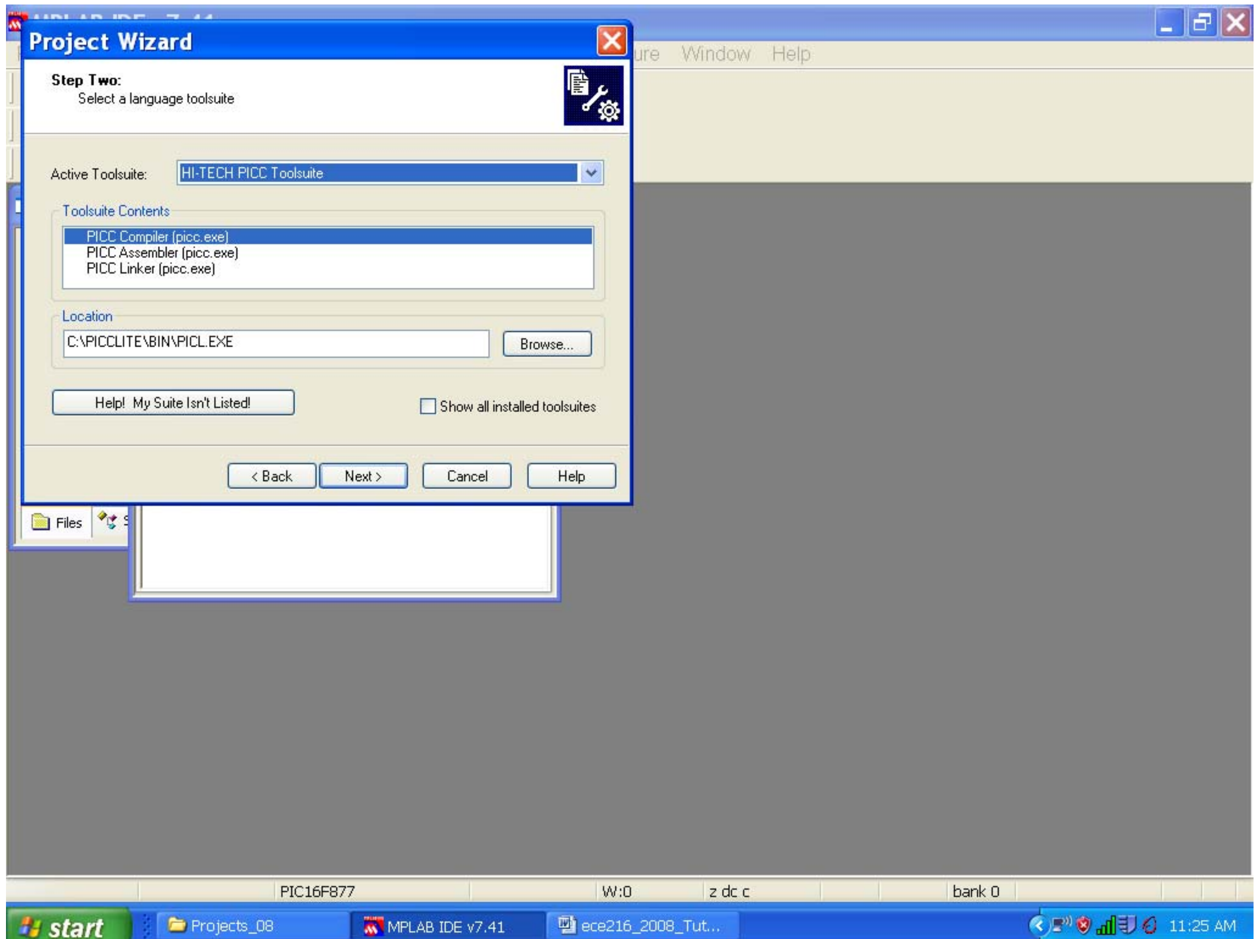
3. We will start a new project using the project wizard, just as in the past.



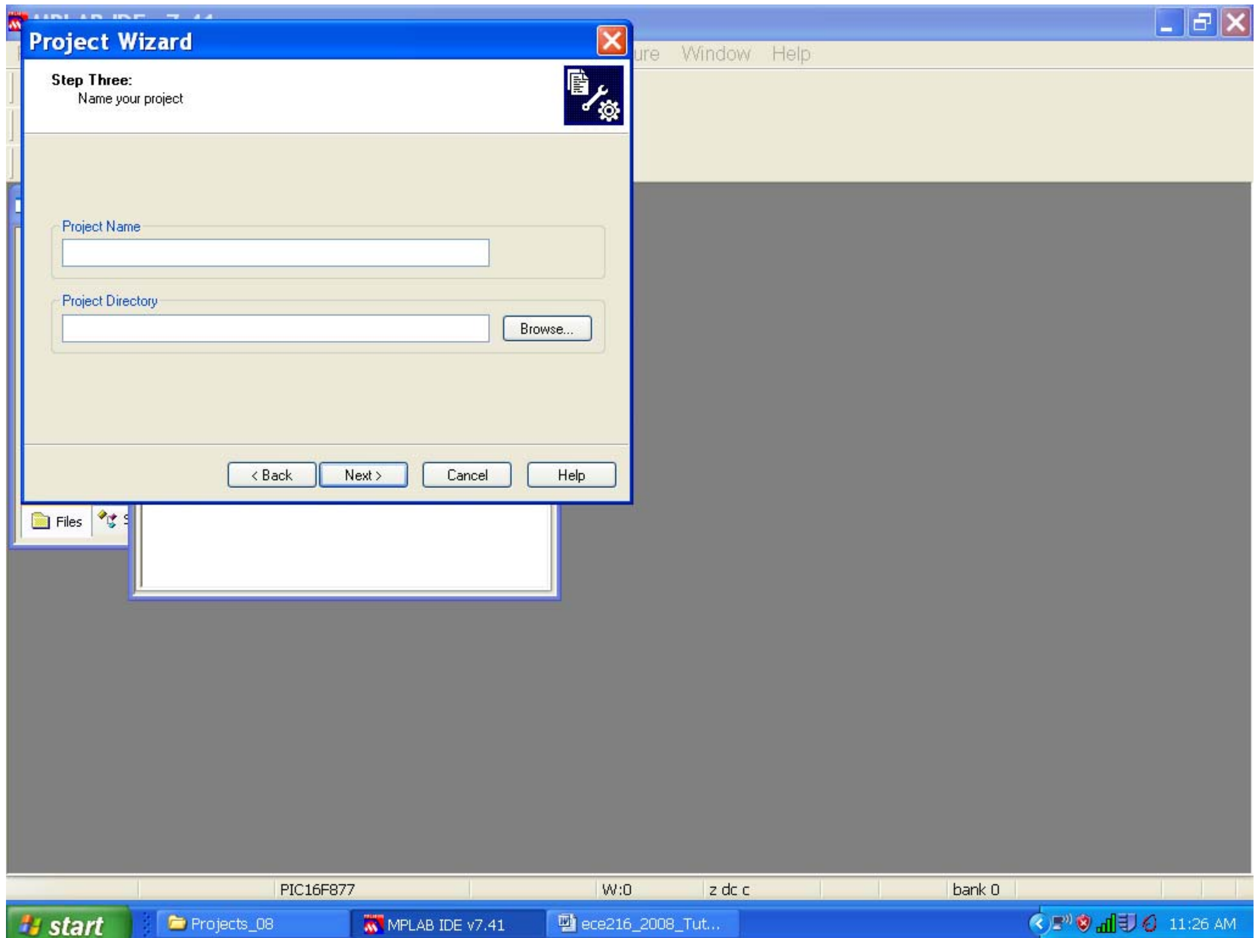
4) Specify the device you will be using, making sure it is one of the devices supported by PICC Lite Compiler.



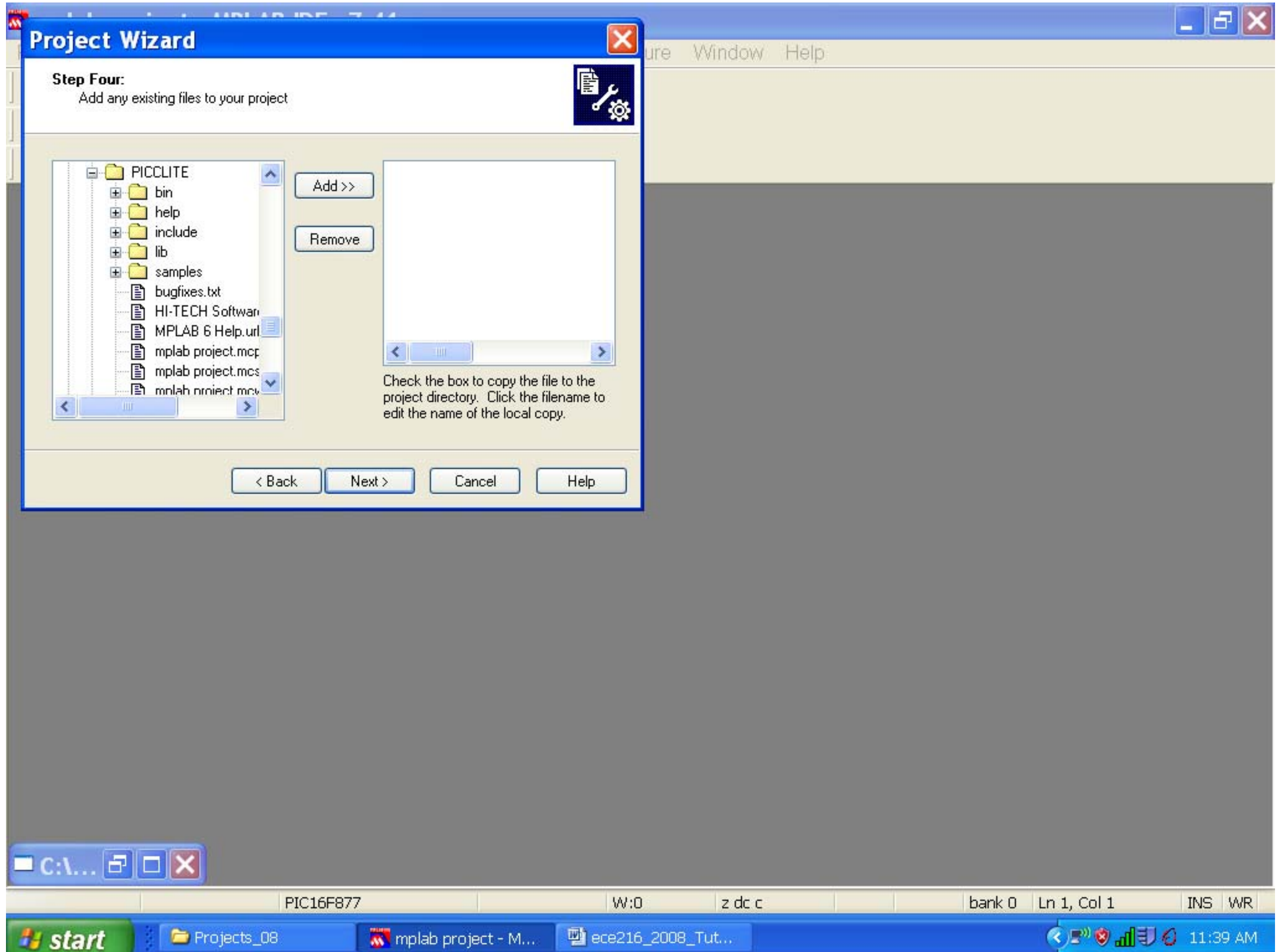
5. This time, when you must select a tool-suite, choose Hi-Tech PICC Toolsuite from the drop down menu. Then set up the directories and project name. If the assembler, linker and compiler are not selected, browse to location C:\PICCLite\Bin\PICL.exe.



6. Now create a Project and Directory just as you've done with the Assembly Language version.



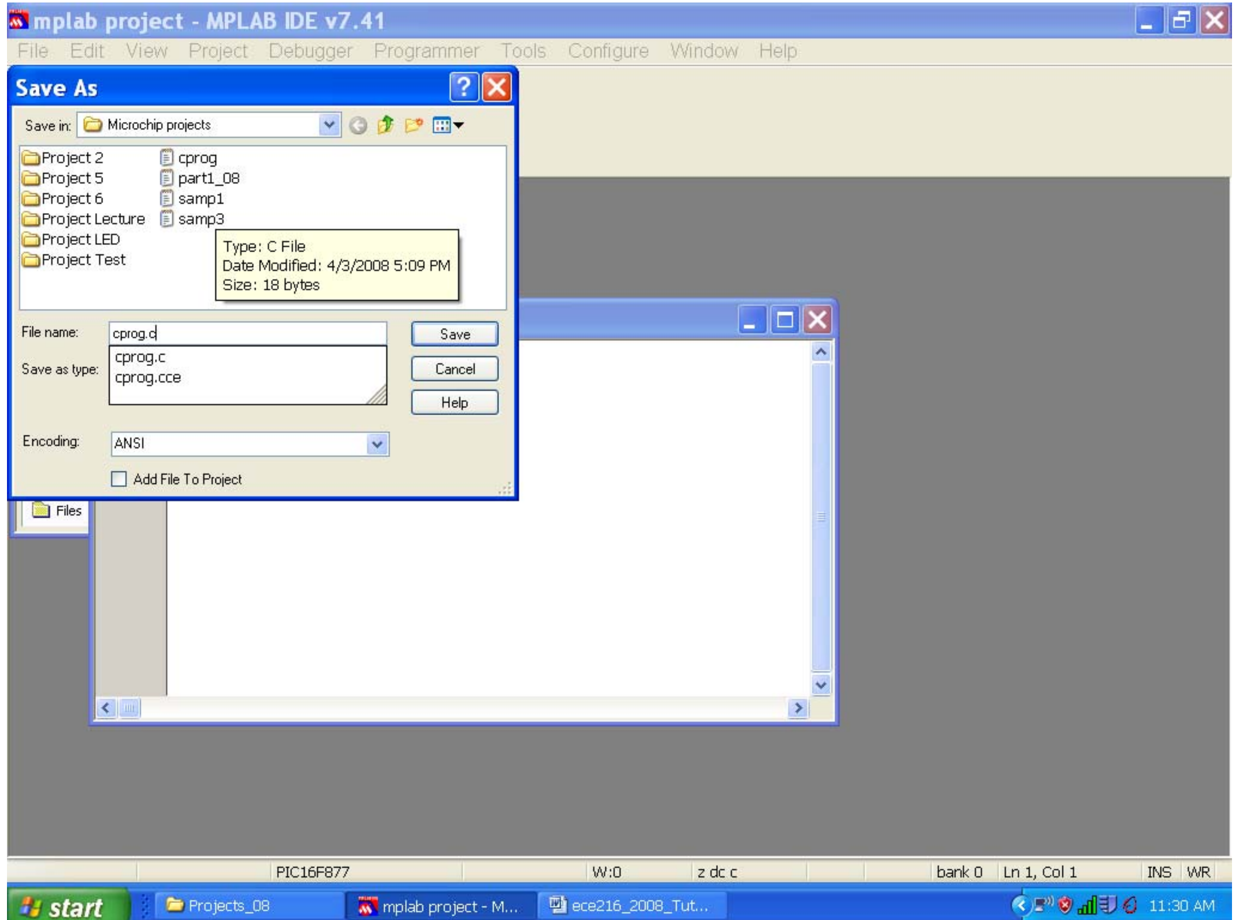
7. Click “NEXT” brings up this window, but there is nothing to add.



Now select “Finish” in Summary window and verify project parameters

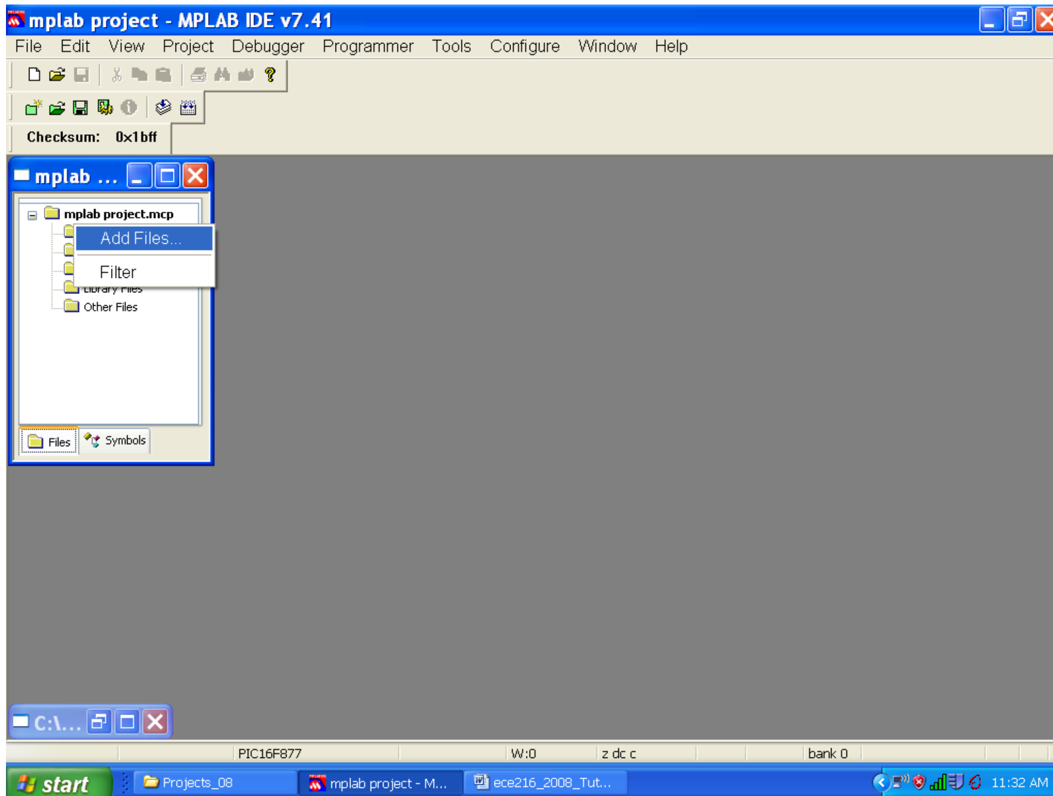
Go to File and selct NEW; open a window “untitled”.

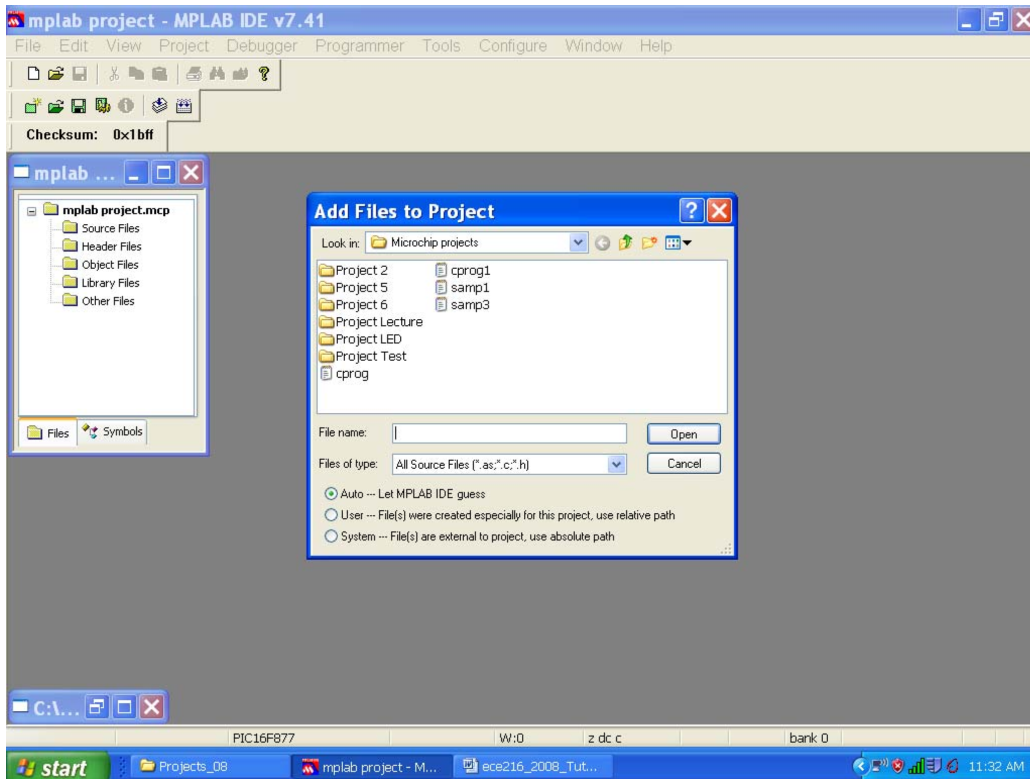
8. Now create a new blank file and save it in your project directory with a ".c" extension.

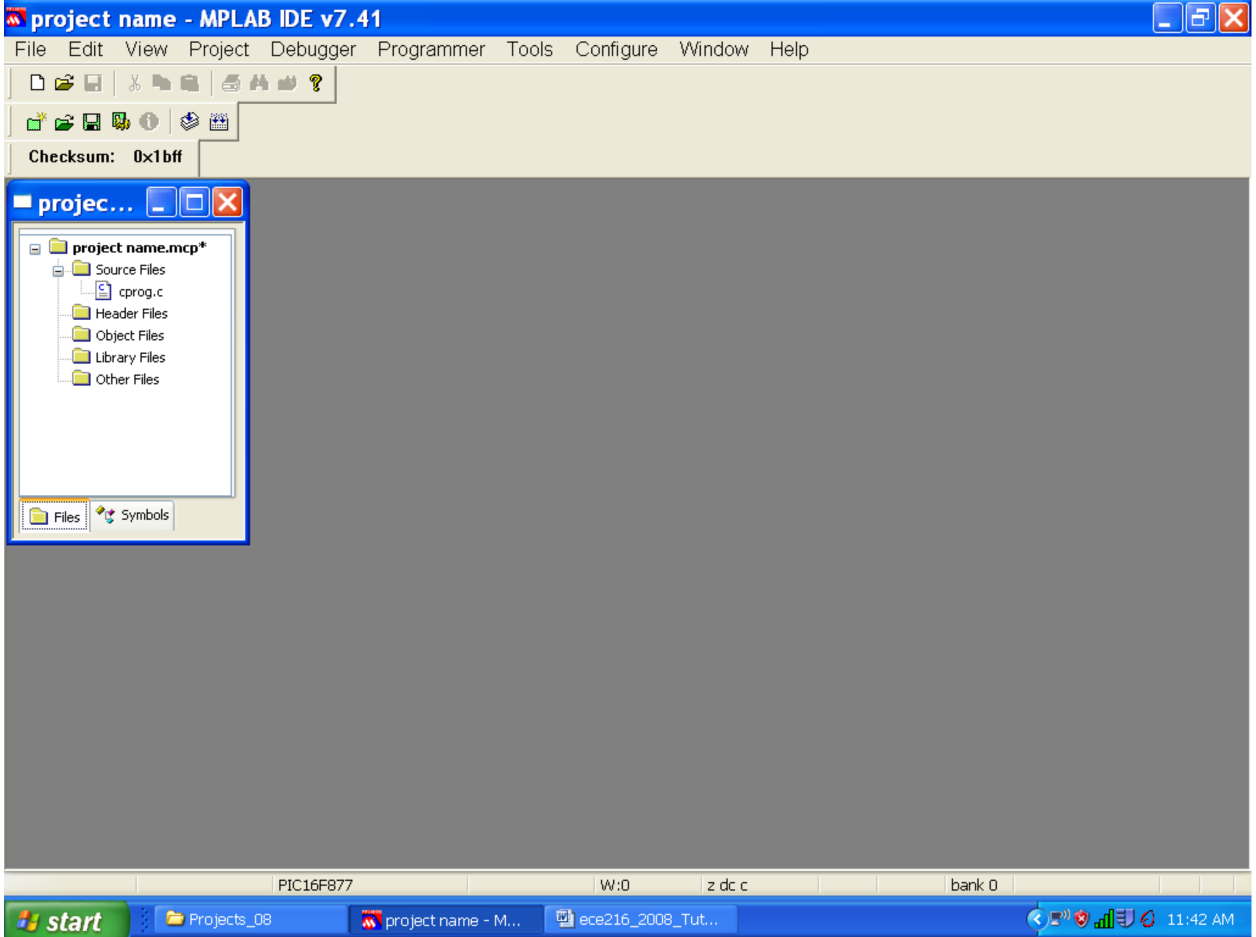


9. Then add this file to your project *Source Files*. By adding the ".c" extension, MPLAB automatically uses C syntax for editing in the IDE.

Right click on source file and select "Add Files" as shown in the following window.



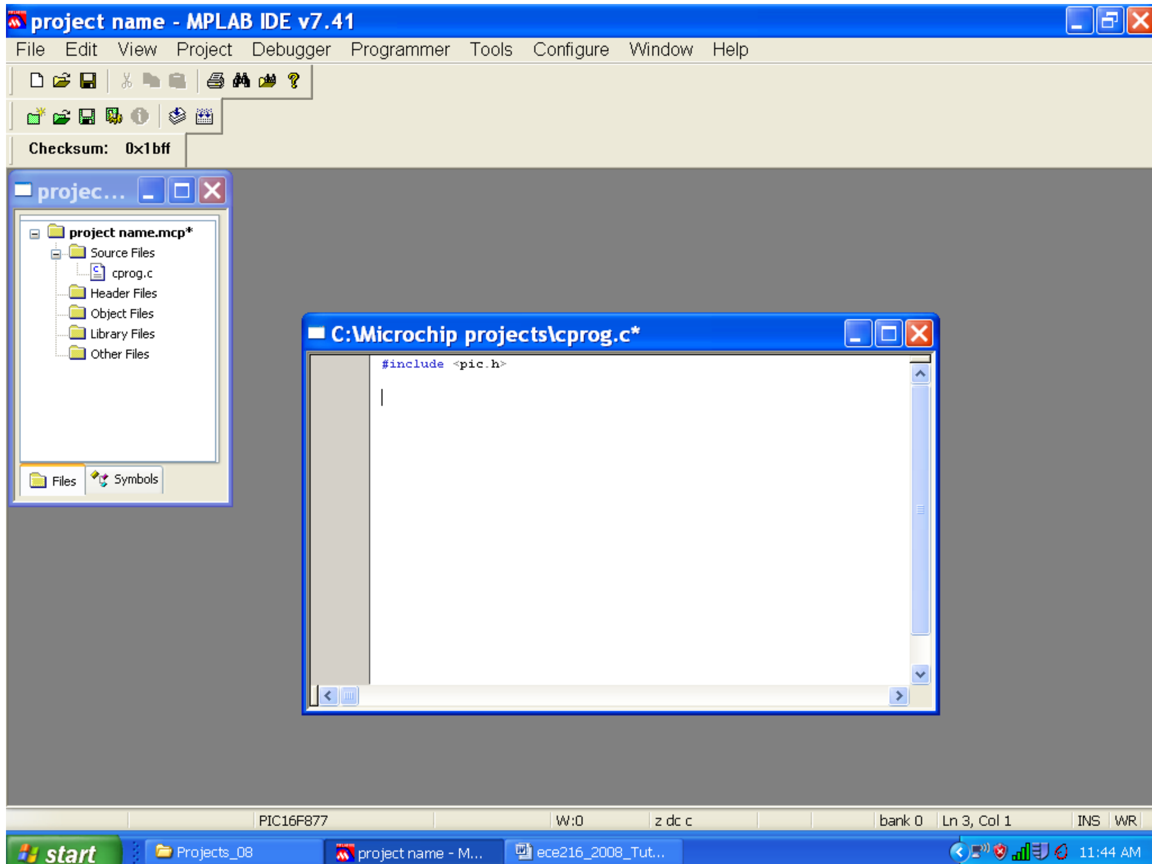

mplab project - MPLAB IDE v7.41
File Edit View Project Debugger Programmer Tools Configure Window Help
Checksum: 0x1bff
mplab project.mcp
Source Files
Header Files
Object Files
Library Files
Other Files
Files
Symbols
Add Files to Project
Look in: Microchip projects
Project 2
Project 5
Project 6
Project Lecture
Project LED
Project Test
cprog
cprog1
samp1
samp3
File name:
Open
Files of type: All Source Files (*.as;*.c;*.h)
Cancel
Auto --- Let MPLAB IDE guess
User --- File(s) were created especially for this project, use relative path
System --- File(s) are external to project, use absolute path
PIC16F877
W:0
z dc c
bank 0
start
Projects_08
mplab project - M...
ece216_2008_Tut...
11:32 AM


project name - MPLAB IDE v7.41
File
Edit
View
Project
Debugger
Programmer
Tools
Configure
Window
Help
Checksum: 0x1bff
projec...
project name.mcp*
Source Files
cprog.c
Header Files
Object Files
Library Files
Other Files
Files
Symbols
PIC16F877
W:0
z dc c
bank 0
start
Projects_08
project name - M...
ece216_2008_Tut...
11:42 AM

Part 2: Setting up the Microcontroller Some Code

1. Now, open the blank file you've created, cprog.c, and the first line of code to add to your blank page is:

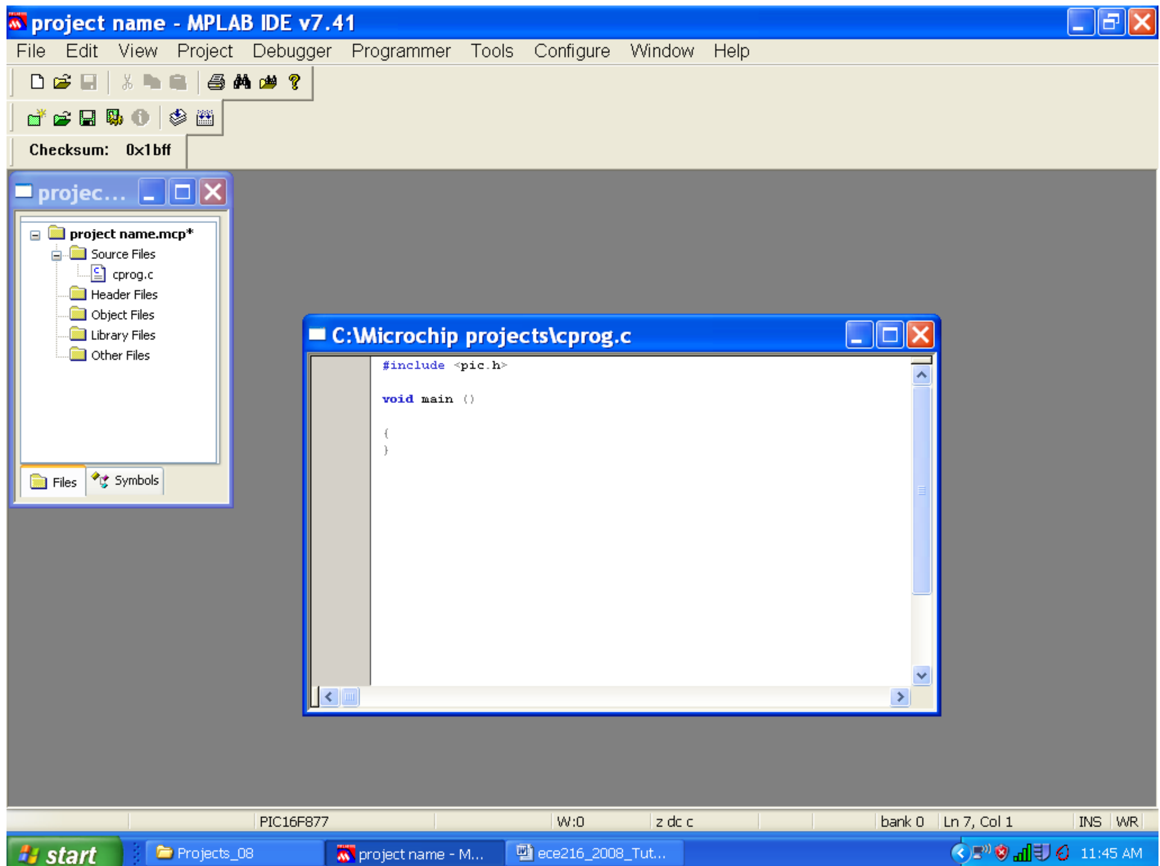
```
#include <pic.h>
```



Navigate to the Hi-Tech directory and look at this include file. It basically sets up all the definitions for the particular chip you are using, similar to the 16F877 inc file we used in assembly.

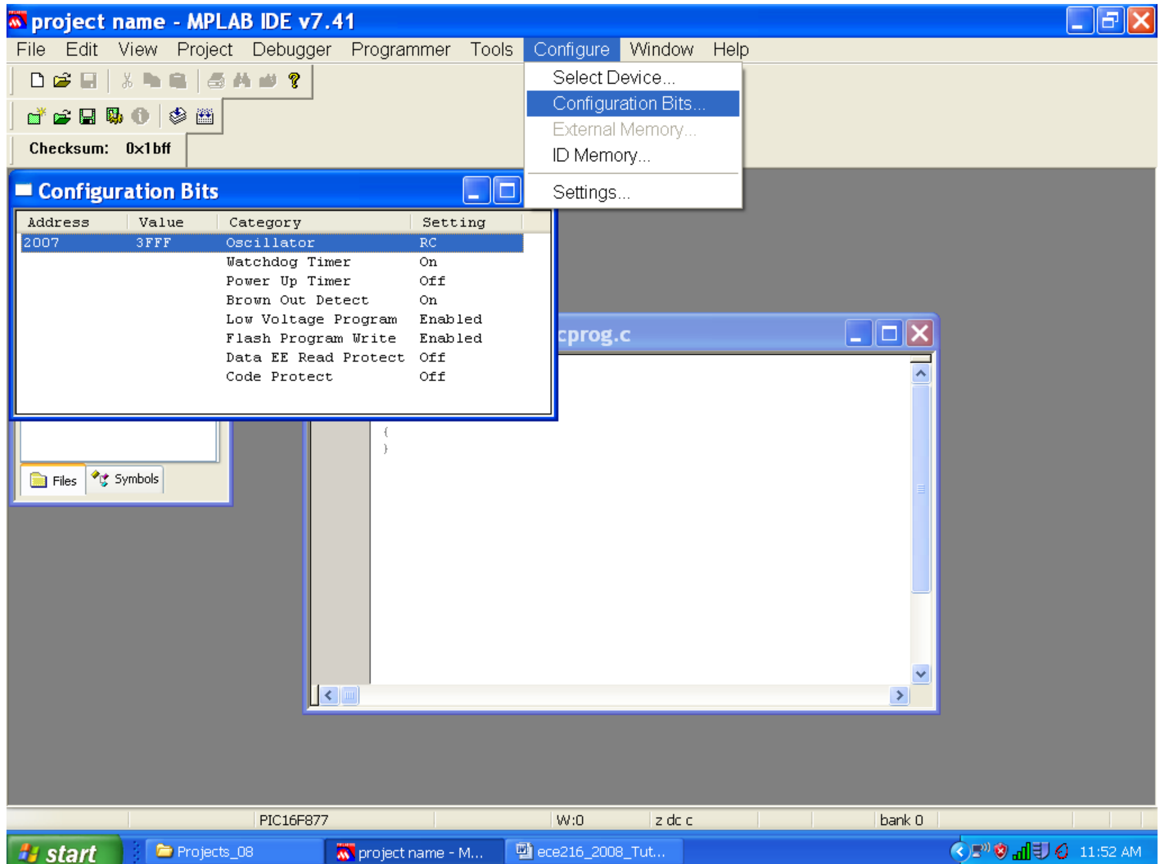
2. Next, we will add our main program body. Everything in PICC is just like ANSI C, so we have:

```
void main (void)
{
}
```

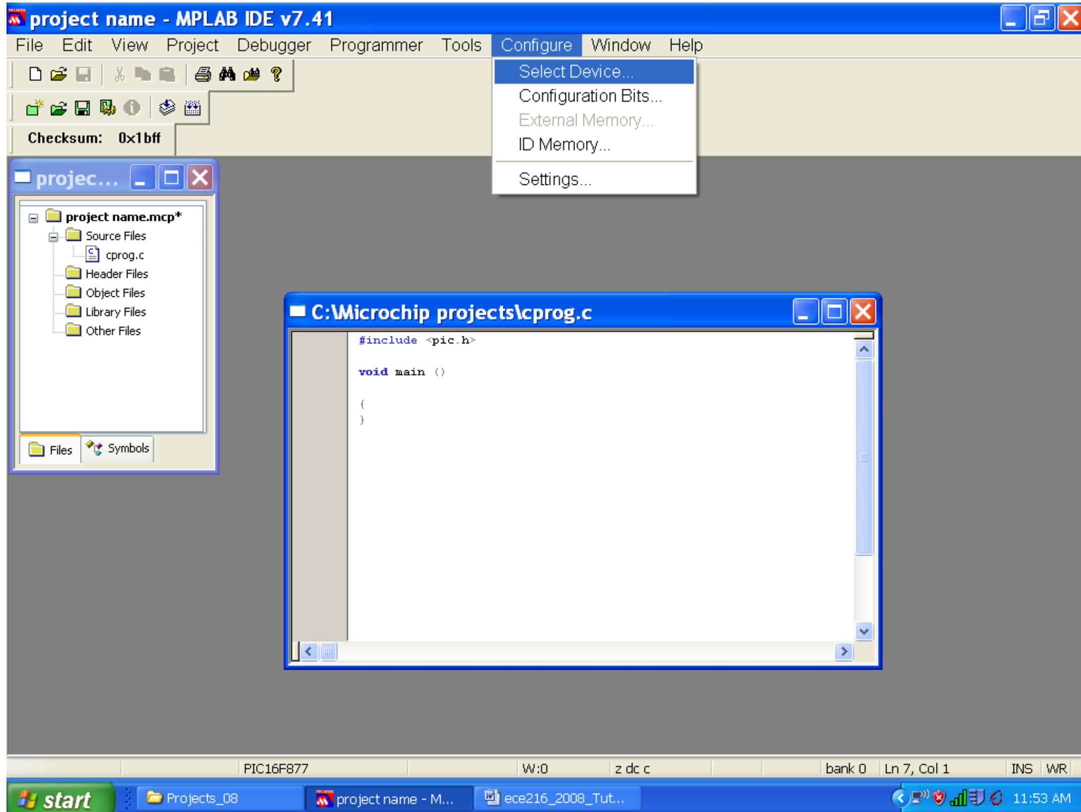


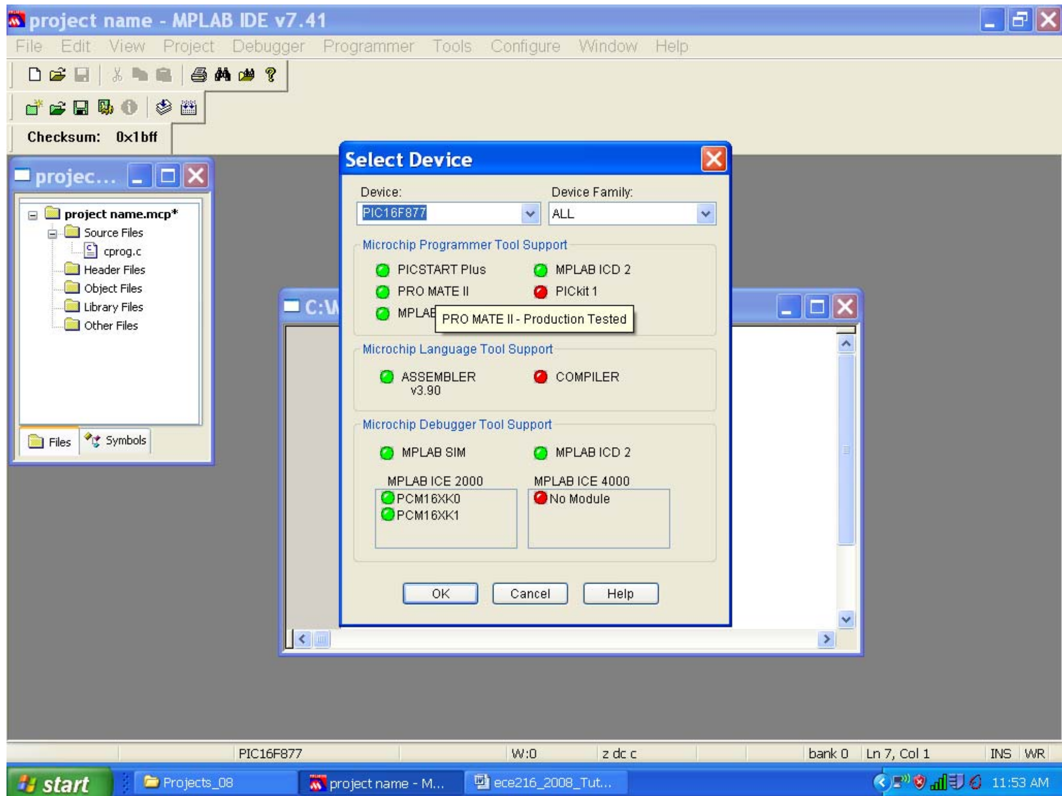
3. Now, just like in our assembly programs, we must set up all of the registers associated with our device - only in C, it is much easier!

To set up the configuration bits, just select the CONFIGURATION Pull-Down Menu and set the configuration you want.

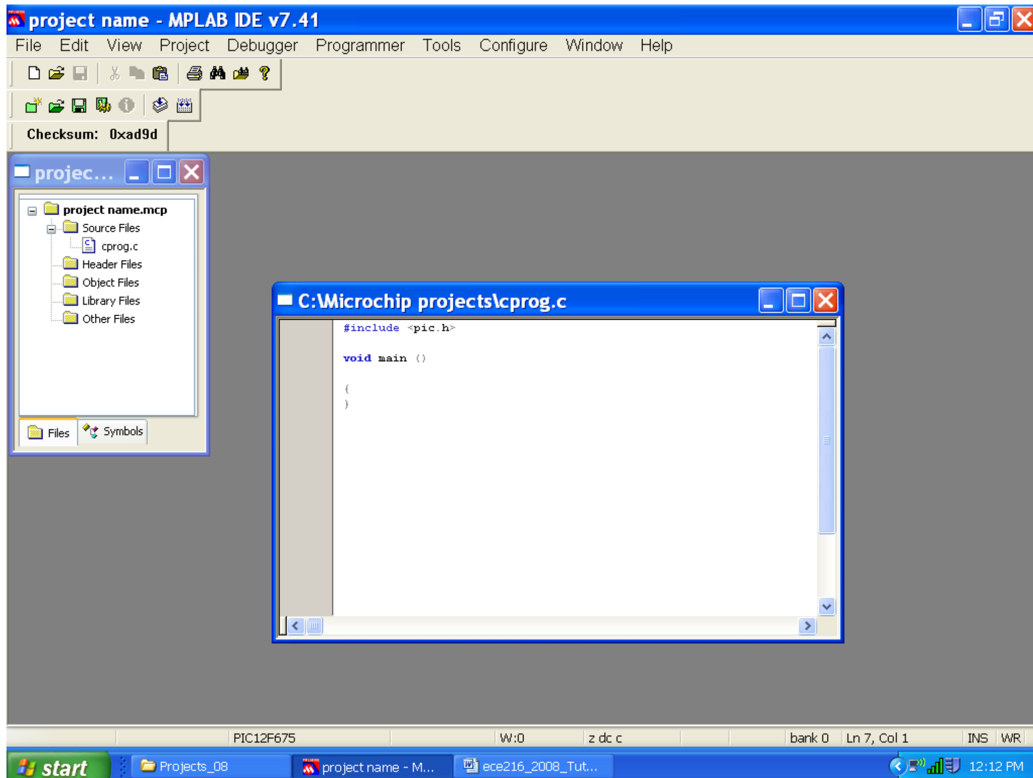


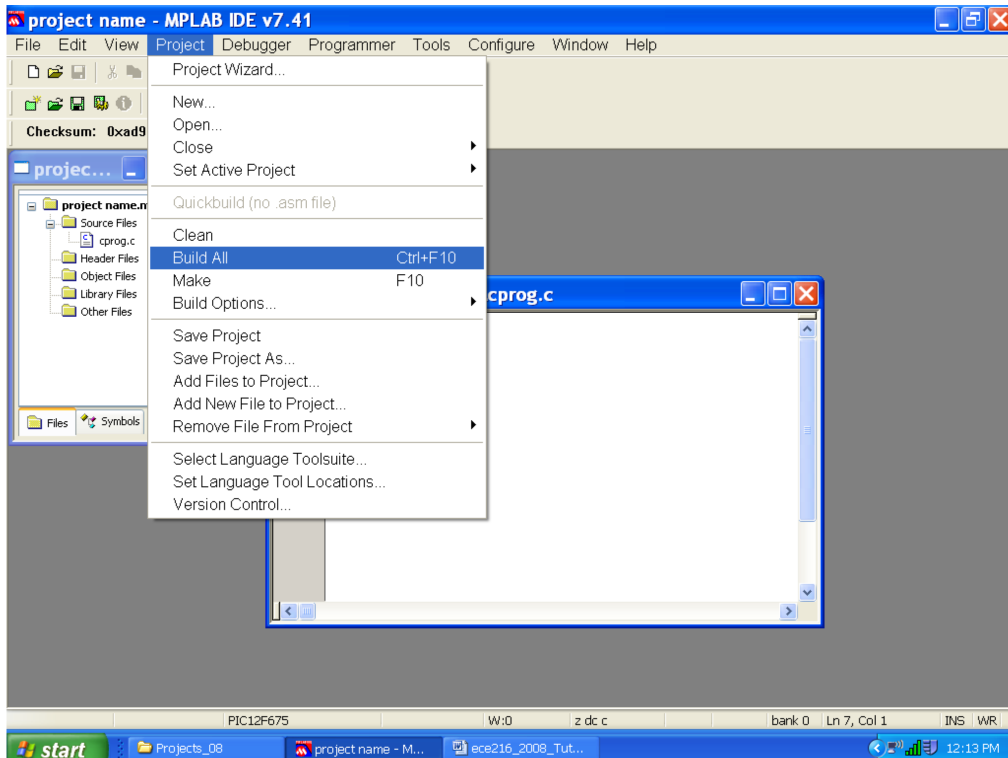
Also, if you want to change devices, simply go to the CONFIGURATION Pull-Down Menu and select the device

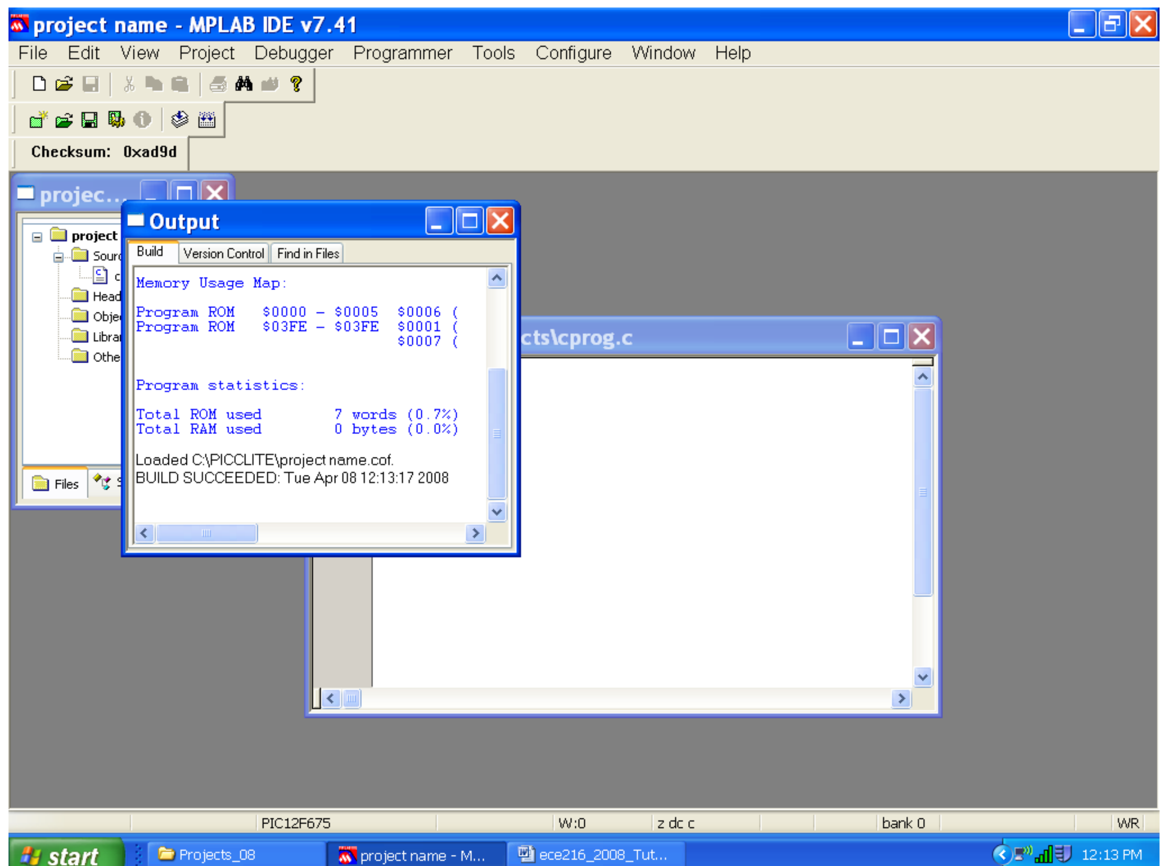



project name - MPLAB IDE v7.41
File Edit View Project Debugger Programmer Tools Configure Window Help
Checksum: 0x1bff
Select Device
Device:
PIC16F877
Device Family:
ALL
Microchip Programmer Tool Support
PICSTART Plus
MPLAB ICD 2
PRO MATE II
PICkit 1
PRO MATE II - Production Tested
Microchip Language Tool Support
ASSEMBLER
v3.90
COMPILER
Microchip Debugger Tool Support
MPLAB SIM
MPLAB ICD 2
MPLAB ICE 2000
PCM16XK0
PCM16XK1
MPLAB ICE 4000
No Module
OK
Cancel
Help
project name.mcp*
Source Files
cprog.c
Header Files
Object Files
Library Files
Other Files
Files
Symbols
PIC16F877
W:0
z dc c
bank 0
Ln 7, Col 1
INS
WR
start
Projects_08
project name - M...
ece216_2008_Tut...
11:53 AM

4. To verify that the device you've selected is supported by the compiler, go to the PROJECTS pull-down menu and select BUILD ALL. It should successfully build the program as show in the slides below.







Part3: Some Code

Now, to enter some code and run a program, begin by selecting the 12F675 Device.

1. To set up the registers, simply set the name of the register equal to some value. For example: GPIO = 4. Easy enough? The compiler takes care of all bank switching, register size considerations, etc. Since our first program will (of course) be a blinking light, let's turn off the ADC, turn off the comparators, and set TRISIO to all outputs. The following code takes care of that, and can be placed right inside the main braces.

```
TRISIO = 0;
ANSEL = 120;
CMCON = 7;
ADCON0 = 140;
```

2. Now, let's attach an LED to pin 2 (GPIO 5). If we want to turn this light on, we just set GPIO to 0b100000 or 0x20 in hex. In addition, we need some sort of

infinite program loop. An elegant way to do this is with a while(1) statement. Add the following to the main program code to make the light blink:

```
while(1)
{
GPIO = 0x20;
GPIO = 0x00;
}
```

3. Now this will obviously switch the light on and off too quickly to see. Let's add a delay! In C, there are many ways to do this, but we will do a quick and dirty one for now. Just add an empty for loop in between each change of state. The only problem you would anticipate is that the 8 bit registers we use for variables can only count to 255. In assembly, we got around this by nesting two loops together. In C, we can simply define our counter as something larger that 8 bits, say…. an UNSIGNED INT. This will let us count much higher and PICC will take care of the assembly code to do it. Add the following line to the main code (before our loop of course):

```
unsigned int index;
```

Then, modify the While loop to look like this:

```
while(1)
{
GPIO = 0x20;
for(index = 0;index<10000;index++);
GPIO = 0x00;
for(index = 0;index<10000;index++);
}
```

4. Before we can compile and test this code, we must set the configuration of the device. In assembly, we did this with some code at the beginning of the program. In C, we are going to do it using MPLAB. From the configure menu, select configuration bits. Set these up as you deem appropriate based on our last couple labs. Explain your choices. Now build your project and test it out.

Part4: Exploiting C

1. C makes programming the PIC incredibly easy, at the cost of some code efficiency. Let's exploit some of the high-level functions and make our code more interested. First, we should make a delay routine that will delay by an amount specified. Add the following code before the main routine.

```
void delay(unsigned int amount)
{
```

```
unsigned int index;
for(index=0;index<amount;index++);
}
```

This routine will count up to whatever number we pass to it. The variable index is local as is the variable amount declared in the function.

Remove the variable index declared in main, and replace the for loops with calls to our delay function. Try two different values:

```
While(1)
{
GPIO = 0x20;
delay(10000);
GPIO = 0x00;
delay(30000);
}
```

2. Now lets add the following subroutine to read a value from the ADC:

```
unsigned int adc_read(void)
    {
    unsigned int result;
    GODONE = 1;
    while(GODONE); // wait for conversion complete
    result=(ADRESH << 8) + ADRESL;
    return result;
    }
```

This routine sets the GODONE bit high to start a conversion, and then waits for the PIC to set it low indicating completion. Then we take ADRESH (the upper 8 bits of the result) and put it into “result” while shifting it by 8. At the same time, we add the lower 8 bits (ADRESL) to get out full 16 bit conversion. Then, result is returned to the caller.

To use this subroutine, be sure to change TRISIO to 0b010000 so that we have an input pin on pin 3. Also, change ADCON0 to 141 so that the ADC is turned on.

3. Let’s modify the main program to call adc_read and then delay accordingly. In C, this is quite simple. Since the function returns an unsigned int, we can just move it into the call to delay. (Add a multiplication by 4 to get a little more delay.)

```
While(1)
{
GPIO = 0x20;
delay(adc_read() * 4);
GPIO = 0x00;
delay(adc_read() * 4);
}
```

4. Attach a voltage divider POT, or a variable voltage source to pin 3. As the voltage input changes, the light will flash at different rates.

So now we have implemented the same function as in Lab #3 using C. **How does using a high-level language compare from the programming perspective? How about in terms of efficiency (code size, reliability, etc.)? Compare the code in this lab to the code from lab #3.**