TLS 2130

Getting Started with MPLAB® C

for dsPIC® and PIC24

Embedded C Programming Series

Lab Manual

Rob Ostapiuk
Microchip Technology Inc.

Getting Started with MPLAB® C
for PIC24 MCUs and dsPIC DSCs

Table of Contents

Reference Materials

All of these documents may be found in the **\docs** subdirectory of your MPLAB C installation. It will typically be located at: **C:\Program Files\Microchip\MPLAB C30\docs**

Electronic Documentation
Release Notes for MPLAB® C30
MPLAB® C for PIC24 and dsPIC Help File
ASM30 Help File
LINK30 Help File
16-bit Peripheral Libraries Master Index

PDF Documentation
MPLAB® C30 User's Guide
MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide
16-bit Language Tool Libraries
16-bit Language Tool Quick Reference Card

The content of this class is applicable to the following products:

SW006012—MPLAB® C for PIC24 MCUs and dsPIC® DSCs (also called MPLAB® C30)
SW006013—MPLAB® C for dsPIC DSCs
SW006014—MPLAB® C for PIC24 MCUs

Lab Exercise 1

Creating an MPLAB C Based Project

Purpose

The purpose of this lab is to illustrate the steps required to create an MPLAB® C based project within the MPLAB Integrated Development Environment. You will learn how to select the compiler as the build tool, which files must be included in your project, how to allocate a heap and what code must be included in your source file.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3 • Proteus Board Level Simulator
Lab files on class PC:	C:\MTT\TLS2130\Lab1\...

Objective

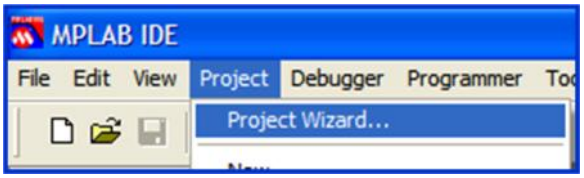
For this first lab, the instructor will walk you through the process of creating an MPLAB® C based project and describe the various decisions that must be made along the way. The steps are described in their entirety throughout this section. Upon completion of this exercise, you will have a complete C project setup in MPLAB with bare minimum required framework in place so that all you need to do is add your application code. More advanced features such as setting configuration bits in code and interrupts will be covered later in the class.

Procedure

This is a follow-along lab. Follow the steps along with the instructor.

Setup the Core MPLAB® Project

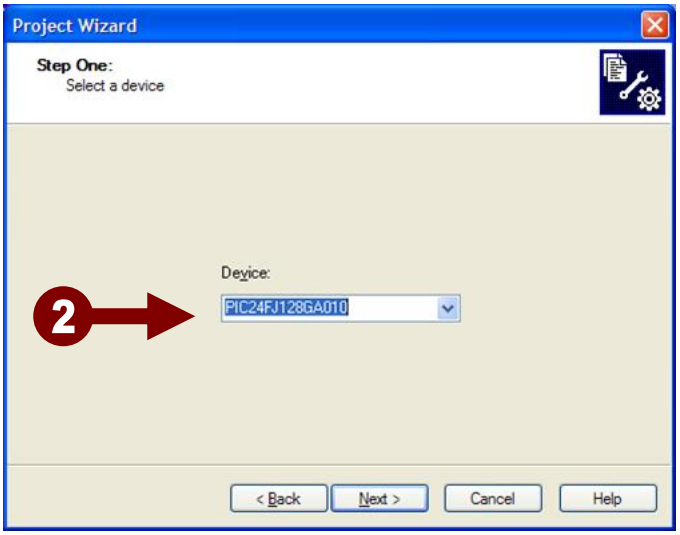
1 Open MPLAB and start the project wizard by selecting from the menu:
Project ▸ Project Wizard...



Click Next > to continue.

2 In the **Device** combo box, select the PIC24FJ128GA010.

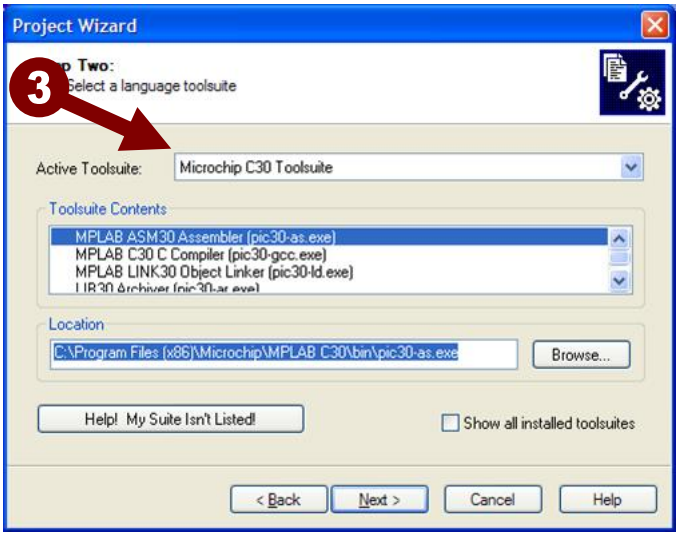
Click Next > to continue.



Although we will be using the PIC24FJ128GA010 for this class, this process applies equally to all PIC24, dsPIC30 and dsPIC33 family devices.

3 In the **Active Toolsuite** combo box, select the MPLAB C30 Toolsuite.

Click Next > to continue.



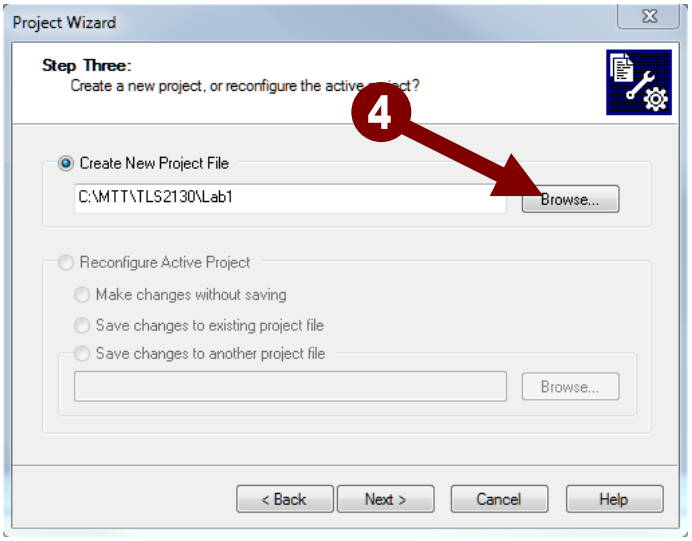
The locations for the various executables listed under **Toolsuite Contents** should be found automatically unless you have a non-standard installation.

4 Click Browse... and navigate to:

C:\MTT\TLS2130\Lab1

Then name the file **Lab1.mcp**

Click Next > to continue.



5 **Add files to the project:**
In the left hand list box, navigate to:
C:\MTT\TLS2130\Lab1

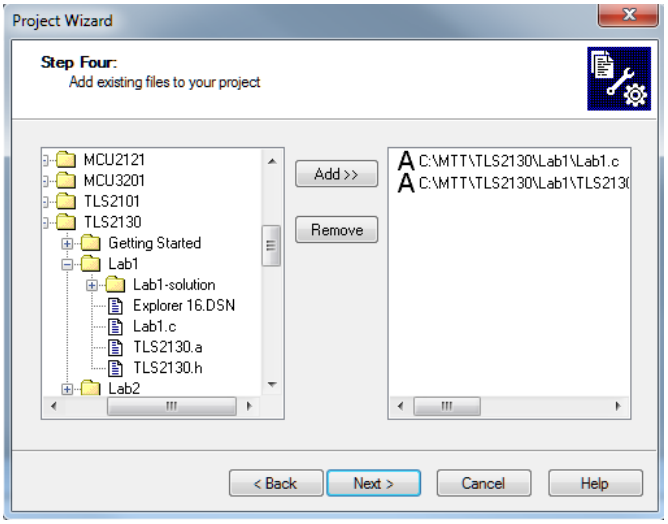
Select the file **Lab1.c**

Click Add>>

6 Select the file **TLS2130.a**

Click Add>>

Click Next > to continue.



This library file (TLS2130.a) is not a required component of all C projects. It was created for this class to simplify LCD interfacing and switch debouncing on the Explorer 16 Development Board. The complete source code for this library file is included on the class CD.

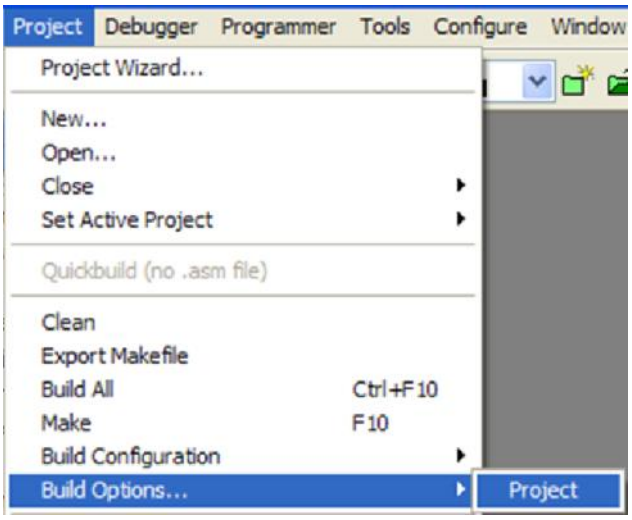
Until recently, you would have been required to add a linker script to the project at this point. As of MPLAB 8.10, the correct linker script will be automatically selected and used in the background, eliminating the need to add it to the project tree manually. While the link step will work in the same way as before, you will not see the linker script in the project tree.

7 Click Finish

At this point, you have a complete C30 project with the minimum configuration—including some items in the main source file which we will discuss shortly. For many applications, we could stop here and simply add in our source code. However, to cover a broader range of applications, we will next allocate a heap so that we can freely use the standard C libraries which include many functions that require a heap.

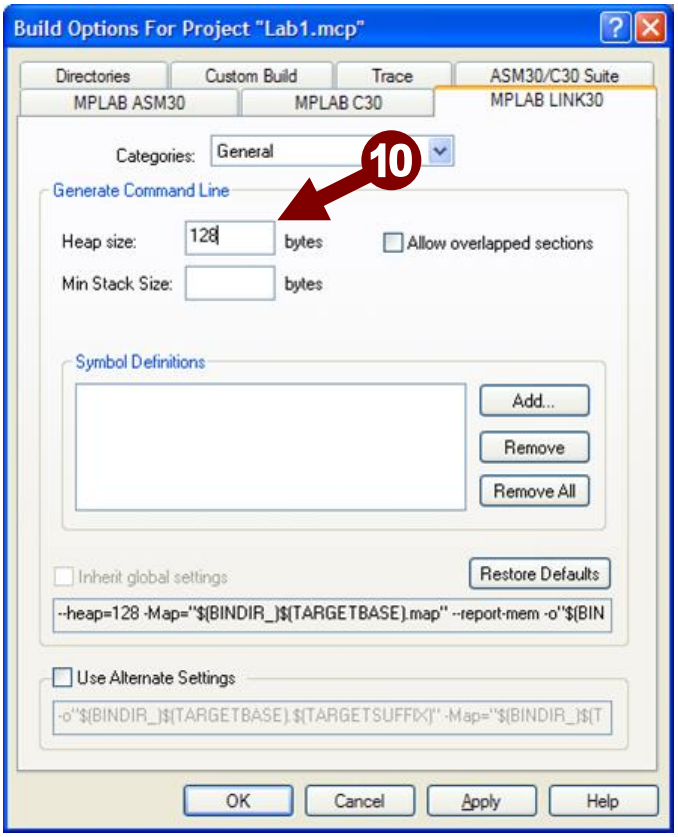
> A heap is simply a block of RAM where we can dynamically allocate and deallocate variables as needed. Think of it as a scratch pad, or a temporary storage area.

8 Open the project build options by selecting from the menu:
Project ▸ Build Options... ▸ Project



9 Select the **MPLAB LINK30** Tab
Allocate a Heap
In the **Heap Size** box, enter a value of ***128*** bytes.

The choice of 128 bytes is somewhat arbitrary. It was selected as a good starting point but may need to be altered depending on your application. If you do not use any of the standard C library functions (such as printf or the string manipulation routines) or you never use the malloc function yourself, then you probably don't need a heap and may skip this step. If your application requires a heap, but none is declared, you will get a compile error. If you allocate a heap that is too small, you will likely encounter runtime errors due to the malloc function returning null as a result of too little space being available.



10 (OPTIONAL)

Select the **Directories** Tab

If you are content with the suite defaults for the various directory paths, you may skip this step or click on the Suite Defaults button.

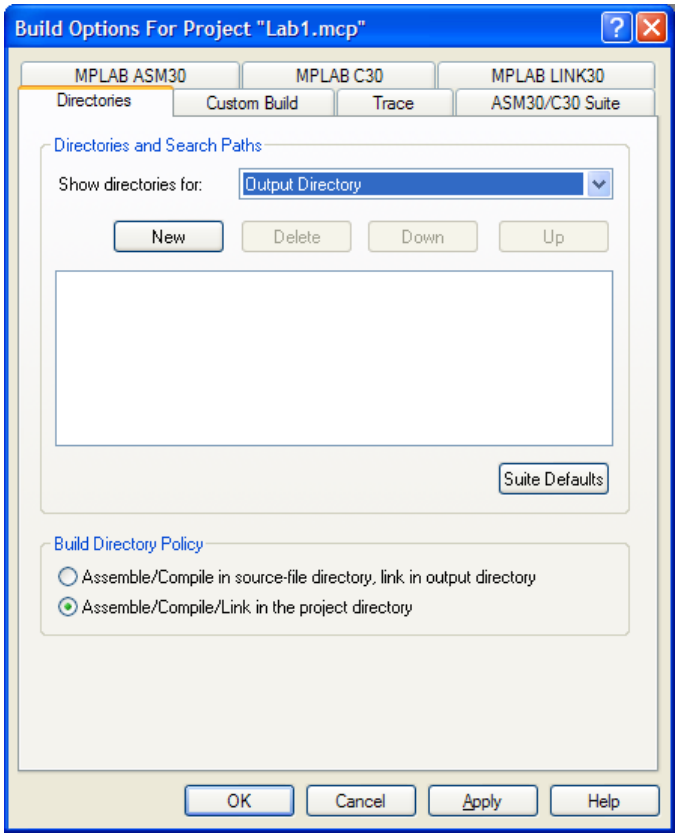
However, if you have a non-standard installation or a more complex project directory structure, you have the ability to setup paths for:

- Output Directory
- Intermediary Directory
- Assembler Include Search Path
- Include Search Path
- Library Search Path

If you have project files in locations other than the standard C30 installation directories or your project directory, paths to those files must be added here.

For this class, we will be using the suite defaults only.

Click OK when done.



11 (OPTIONAL)

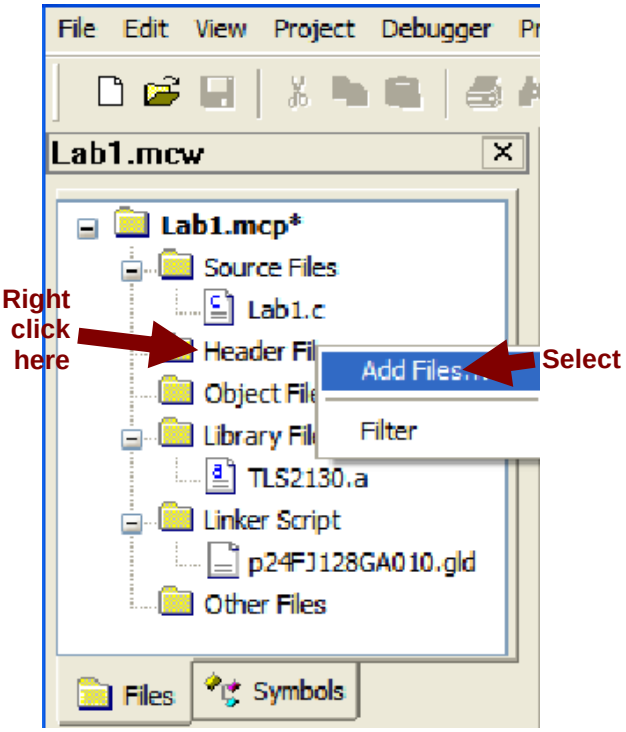
Add Header Files to the Project

Right click on the **Header Files** node of the project tree and select **Add Files...** from the popup menu.

> Adding header files to the project tree is **completely optional**. This simply provides quick access to them from within your project. The compiler accesses header files via the `#include` directive in source files and ignores them in the project tree.

We will be including three header files into this project: the device specific header file, the standard I/O header file, and the TLS2130 header file for the library TLS2130.a. Simply repeat the process above three times—once for each of the files.

Continued on next page...

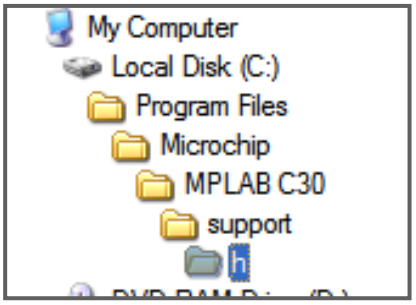


Add Header Files to the Project (Continued)

Device specific header files are in the directory:
C:\Program Files\Microchip\MPLAB C30\Support\h

We will be using the file: **p24fj128ga010.h**

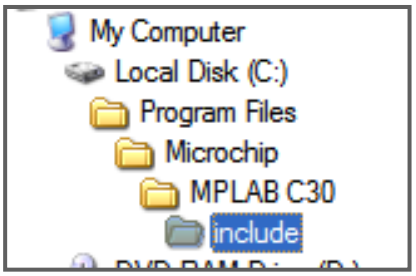
This file contains definitions for register names and configuration settings.



C Standard Library header files are in the directory:
C:\Program Files\Microchip\MPLAB C30\include

For Lab1 we will be using the file: **stdio.h**

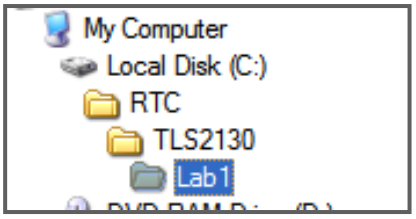
This file contains function prototypes for the standard C library's I/O functions (Standard I/O Library). We need this for the printf() function that is used in this program.



We will also need the header file associated with the TLS2130.a library file which is in our project directory:
C:\MTT\TLS2130\Lab1

The file we need is: **TLS2130.h**

This file contains function prototypes for the LCD and switch debounce routines written for this class. This is not a part of the standard C library or the libraries included with MPLAB C30, but it is included on the class CD along with the source code for the library itself.



12 **Add the Minimum Required Code Framework**

`#include` the required header files near the top of your source code.

The include files need not be at the very top of your code, but they must be declared before anything in them is used in your code.

Lab1.c (Line 15-17)

```
#include <p24fj128ga010.h>
#include <stdio.h>
#include "TLS2130.h"
```

`#include <file>`
Angle brackets indicate that a file exists within the MPLAB C30 search path (at or below the MPLAB C30 directory). If the file is not found in the compiler's search path, it will result in a compile error.

`#include "file"`
`#include "C:\...\file"`
Quotes indicate that a file exists within the project directory, unless a fully qualified path is provided. In that case, the compiler will look for the file at the specified location. If the file is not found in the project directory or in the specified location, it will result in a compile error.

Header files must be included in your source code with `#include`. Including them in the project tree has no effect. This is an ANSI C requirement – not an MPLAB® requirement.

Set Device Configuration Options

Using the `_CONFIG1` and `_CONFIG2` macros, setup the configuration bits for your application. For this exercise, we will use the following minimum settings to work with the Explorer 16 Demonstration Board. Any settings that are not explicitly specified are left in their default states. (See the data sheet for the default values.) We will discuss how to use these configuration macros in detail in the next section.

Lab1.c (Line 19)

```
_CONFIG1(FWDTEN_OFF &
```

Finished (with the setup)

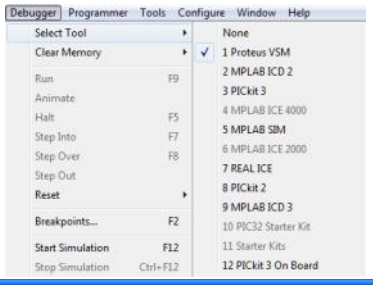
The source code below is the program that we will run on the Explorer 16 board in just a moment:

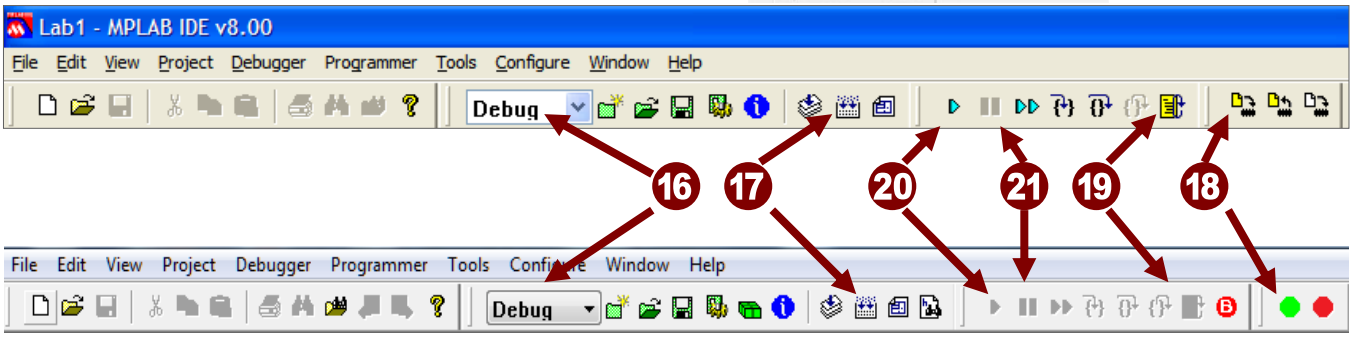
Lab1.c

```
#include <p24fj128ga010.h>          // Not used in this program
#include <stdio.h>                  // Required for printf() function
#include "TLS2130.h"                // Required for LCD routines

_CONFIG1(FWDTEN_OFF & JTAGEN_OFF)

int main(void)
{
    /*-------------------------------------------------------------------
      Setup and write text to LCD to show program is working
    -------------------------------------------------------------------*/
    lcdInit();                      // Setup PMP and initialize LCD
    lcdPutStr("Hello, world!");     // Write text string to LCD

    /*-------------------------------------------------------------------
      "Dummy" call to printf() to show that heap was allocated correctly.
      If a heap is not declared, this will result in a compile error.
      This function will produce no output (UART is not enabled).
    -------------------------------------------------------------------*/
    printf("Nothing to see here.");

    /*-------------------------------------------------------------------
      Loop forever...
    -------------------------------------------------------------------*/
    while(1);
}
```

There are three functions called within our main program before we go into an infinite loop. First is `lcdInit()` which sets up the Parallel Master Port to communicate with the LCD module on the Explorer 16 board. Next is `lcdPutStr()` which will write a string of text out to the LCD module at the current cursor position. Finally, the `printf()` function is called. Normally, this function would write its argument to UART1. But, since we haven't configured the UART, it will not transmit any data written to its transmit register. Its only purpose here is to generate a compile error if we haven't properly allocated a heap. While `printf()` may be used to write data to the UART, it isn't the most efficient way to do so. If you are concerned about code space and execution speed, then you are much better off using the UART libraries or writing directly to the UART yourself.

15 **Enable a debugger**
If not already enabled, From the menu bar select:
Debugger ▸ Select Tool ▸ 7 REAL ICE or
Debugger ▸ Select Tool ▸ 1 Proteus VSM



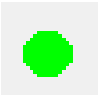


16 Select **Debug** mode.

17 Click on the **Build All** button.

18a If using hardware; Click on the **Build All** button.

18b If using Proteus; Click on the **Start Simulation** button.



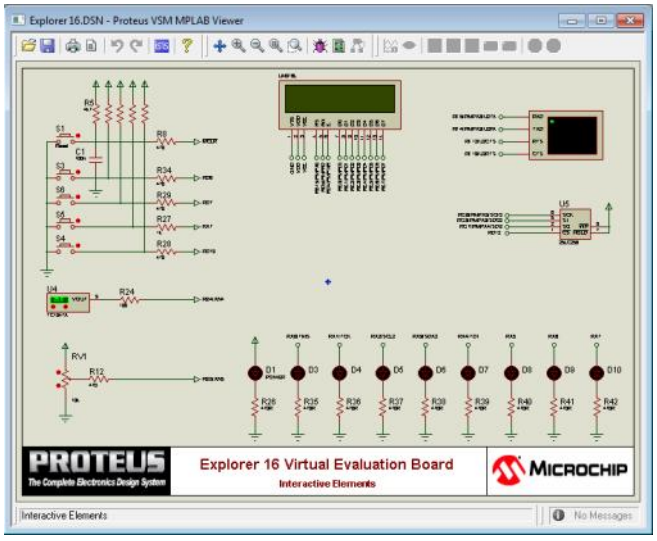
19 When programming completes, click on the **Reset** button.

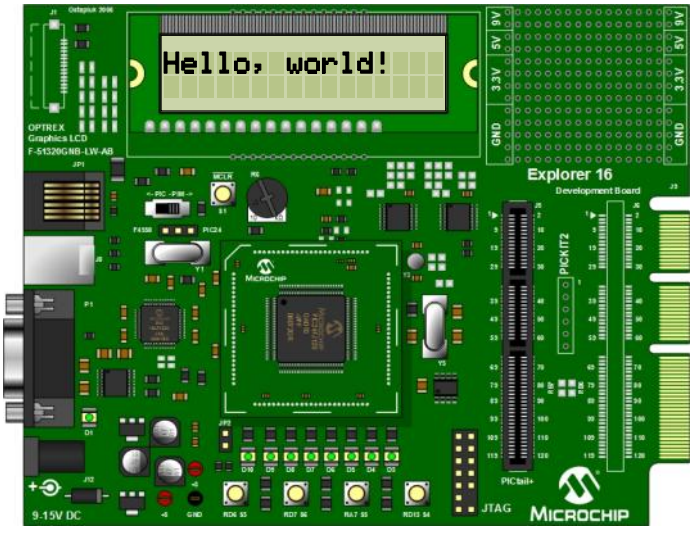
20 Click on the **Run** button.

21 Click on the **Halt** button.

Results

The text “Hello, world!” should appear on the LCD of the Explorer 16 Development Board.





Conclusions

You should now know how to setup an MPLAB® C30 project within MPLAB. It may be accomplished most easily through the use of the Project Wizard, and requires the following steps:

1. Select the device
2. Select MPLAB C30 as the active tool suite
3. Create a new project file
4. Add source files to project (if available—may be added later if they haven't been written yet)
5. Add any required library files to project (you *might* not need any libraries for your project)
6. Add linker script for the selected device—this is absolutely required.
7. Allocate a heap (if needed)
8. Setup directories (if desired)
9. Add header files to project tree (if desired)
10. #include header files in your source code (generally required)
11. Set device configuration bits in code (strongly recommended)

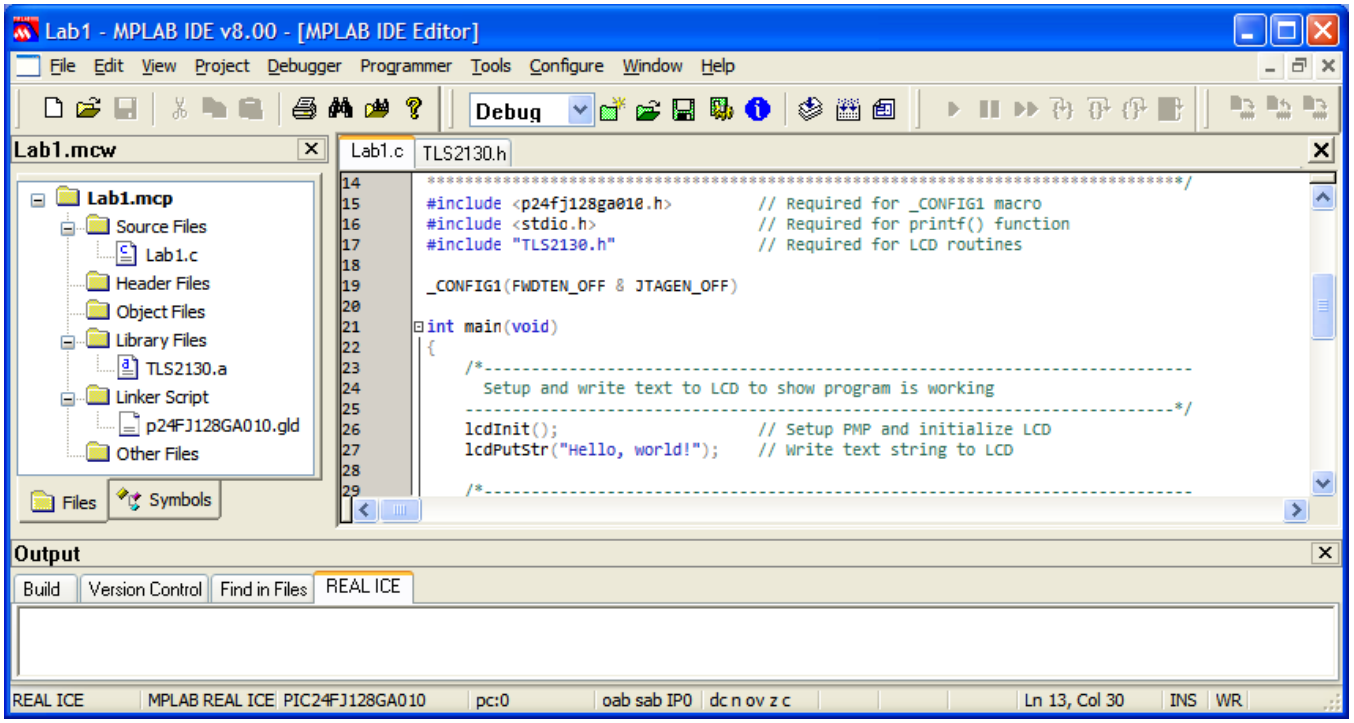
Once the project has been setup correctly, the Build All button will launch the LINK30 linker, which in turn runs the MPLAB C30 compiler for each C source file in your project.

If you are coming to C30 from MPASM, there are fundamentally only two differences between setting up a project for MPASM and for MPLAB C30:

1. The selection of MPLAB C30 as the active tool suite
2. The inclusion of a linker script

Other items are either optional, or may not be required for all projects, but should be considered if you run into problems with your setup:

1. Inclusion of libraries
2. Allocation of a heap
3. #include header files (analogous to include files: *.inc) for any libraries you use, including standard C libraries which need not be included in the project tree.



Lab Exercise 2

Setting PIC® Configuration Bits in Code

Purpose

The purpose of this lab is to test your understanding of the use of the `_CONFIG1` and `_CONFIG2` macros to setup the device configuration bits in code, based on a given set of desired configuration options.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC:	C:\MTT\TLS2130\Lab2\...

Objective

Based on what you learned in this section, setup the PIC® configuration words in code with the following set of configuration options by using the **`_CONFIG1`** and **`_CONFIG2`** macros:

- Oscillator = ***HS*, *Primary with PLL***
- OSC2/RC15 Function = ***OSC2***
- Clock Switching & Monitor = ***Both Disabled***
- JTAG = ***Disabled***
- Watchdog Timer = ***Disabled***

A table of the macro parameter labels that you will need is provided in the Procedure section.

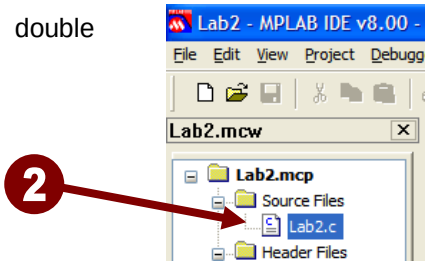
Procedure

1 If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open Lab 2 by selecting from the menu:
File ▸ Open Workspace...
and opening the workspace file located at:

C:\MTT\TLS2130\Lab2\Lab2.mcw

2 In the project tree, under the **Source Files** heading, double click the file **Lab2.c** to open it in the editor.



3 Edit the lines in Lab2.c marked with the comment "`//### Your Code Here ###`" to initialize the configuration bits with the settings at right. Note that the oscillator requires two options to be specified (e.g. Oscillator Type and Oscillator Selection).

- Oscillator = ***HS***, ***Primary with PLL***
- OSC2/RC15 Function = ***OSC2***
- Clock Switching & Monitor = ***Both Disabled***
- JTAG = ***Disabled***
- Watchdog Timer = ***Disabled***

Lab2.c

```
#include <p24fj128ga010.h>          // Required for _CONFIGx macros

//### Your Code Here ###           //### (3.1) Use _CONFIG1 macro here
//### Your Code Here ###           //### (3.2) Use _CONFIG2 macro here

int main(void)
{
    while(1);
}
```

Step 3.1: Call the _CONFIG1 Macro

The _CONFIG1 macro is used to setup the CONFIG1 register, which controls the following features: JTAG, Code Protection, Write Protection, Background Debug, Clip-on Emulation, ICD Pins Select, and all Watchdog Timer options. Table 2-1 on the following page provides all of the macro's parameter labels as defined in the PIC24FJ128GA010.h header file.

Table 2-1: _CONFIG1(x) Macro Parameter List Options for PIC24FJ128GA010

JTAG	JTAGEN_OFF	Disabled
	JTAGEN_ON	Enabled
Code Protect	GCP_ON	Enabled
	GCP_OFF	Disabled
Write Protect	GWRP_ON	Enabled
	GWRP_OFF	Disabled
Background Debug	BKBUG_ON	Enabled
	BKBUG_OFF	Disabled
Clip-On Emulation	COE_ON	Enabled
	COE_OFF	Disabled
ICD Pins Select	ICS_PGx1	EMUC/EMUD share PGC1/PGD1
	ICS_PGx2	EMUC/EMUD share PGC2/PGD2
Watchdog Timer	FWDTEN_OFF	Disabled
	FWDTEN_ON	Enabled
Windowed WDT	WINDIS_ON	Enabled
	WINDIS_OFF	Disabled
Watchdog Prescaler	FWPSA_PR32	1:32
	FWPSA_PR128	1:128
Watchdog Postscaler	WDTPS_PS1	1:1
	WDTPS_PS2	1:2
	WDTPS_PS4	1:4
	WDTPS_PS8	1:8
	WDTPS_PS16	1:16
	WDTPS_PS32	1:32
	WDTPS_PS64	1:64
	WDTPS_PS128	1:128
	WDTPS_PS256	1:256
	WDTPS_PS512	1:512
	WDTPS_PS1024	1:1024
	WDTPS_PS2048	1:2048
	WDTPS_PS4096	1:4096
	WDTPS_PS8192	1:8192
	WDTPS_PS16384	1:16384
	WDTPS_PS32768	1:32768

_CONFIG*x* Macro Syntax

```
_CONFIGx(OPTION1 & OPTION2 & … & OPTIONn)
```

To form the parameter list for the macros, simply bitwise AND (&) together the desired options from the tables above. Any parameters you leave out will retain their default settings as per the data sheet.

Step 3.2: Call the _CONFIG2 Macro

The _CONFIG2 macro is used to setup the CONFIG2 register, which controls the following features: Two-Speed Startup, Oscillator Selection, Oscillator Type, Clock Switching & Monitor, and the RC15 Function. Table 2-2 below provides all of the macro's parameter labels as defined in the PIC24FJ128GA010.h header file.

Table 2-2: _CONFIG2(x) Macro Parameter List Options for PIC24FJ128GA010

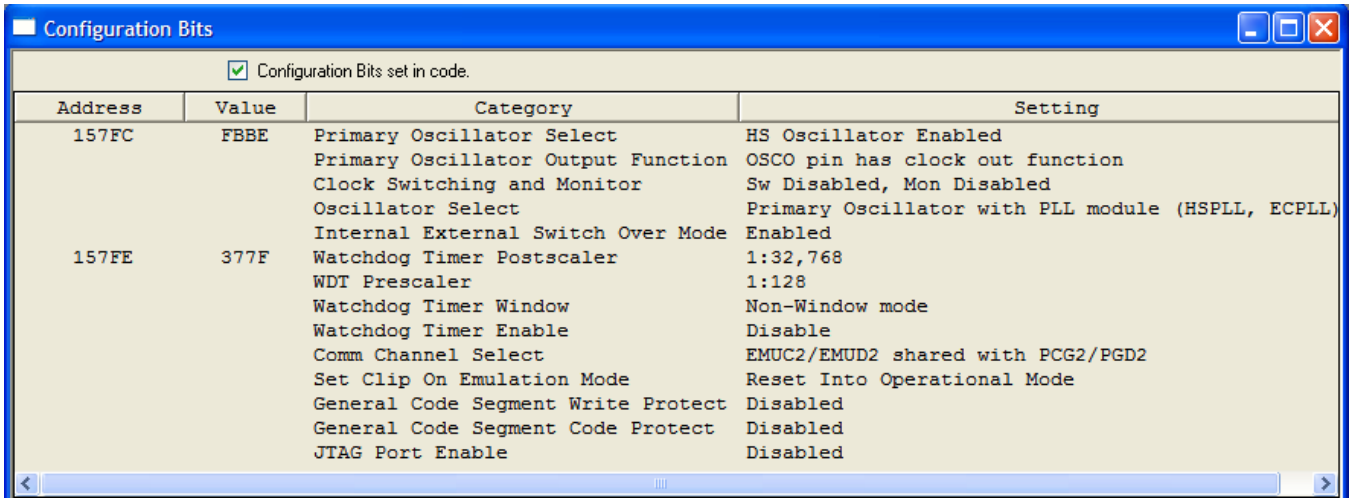
Two Speed Startup	`IESO_OFF`	Disabled
	`IESO_ON`	Enabled
Oscillator Selection	`FNOSC_FRC`	Fast RC Oscillator
	`FNOSC_FRCPLL`	Fast RC Oscillator with divide and PLL
	`FNOSC_PRI`	Primary Oscillator (XT, HS, EC)
	`FNOSC_PRIPLL`	Primary Oscillator (XT, HS, EC) with PLL
	`FNOSC_SOSC`	Secondary Oscillator
	`FNOSC_LPRC`	Low Power RC Oscillator
	`FNOSC_FRCDIV`	Fast RC Oscillator with divide
Clock Switching & Monitor	`FCKSM_CSECME`	Both Enabled
	`FCKSM_CSECMD`	Only Clock Switching Enabled
	`FCKSM_CSDCMD`	Both Disabled
OSC2/RC15 Function	`OSCIOFNC_ON`	RC15
	`OSCIOFNC_OFF`	OSC2 or $F_{OSC}/2$
Oscillator Type	`POSCMOD_EC`	External Clock
	`POSCMOD_XT`	XT Oscillator
	`POSCMOD_HS`	HS Oscillator
	`POSCMOD_NONE`	Primary Disabled

4 Click on the **Build All** button.

5 Check your results by opening up the Configuration Bits window. From the menu select:
Configure ▸ Configuration Bits
If the Configuration Bits window is not visible, it may be hidden behind the source code window.

Results

After building the code, the Configuration Bits window will show the settings for the device. The settings should match those we specified in the `_CONFIG1` and `_CONFIG2` macros.
Your Configuration Bits window should match the one shown on the next page.



Address	Value	Category	Setting
157FC	FBBE	Primary Oscillator Select	HS Oscillator Enabled
		Primary Oscillator Output Function	OSCO pin has clock out function
		Clock Switching and Monitor	Sw Disabled, Mon Disabled
		Oscillator Select	Primary Oscillator with PLL module (HSPLL, ECPLL)
		Internal External Switch Over Mode	Enabled
157FE	377F	Watchdog Timer Postscaler	1:32,768
		WDT Prescaler	1:128
		Watchdog Timer Window	Non-Window mode
		Watchdog Timer Enable	Disable
		Comm Channel Select	EMUC2/EMUD2 shared with PCG2/PGD2
		Set Clip On Emulation Mode	Reset Into Operational Mode
		General Code Segment Write Protect	Disabled
		General Code Segment Code Protect	Disabled
		JTAG Port Enable	Disabled

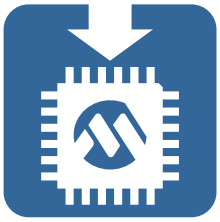
Conclusions

Device configuration options can easily be set in code so that no matter who you send your source files to, they will be able to replicate the correct settings when programming the part. Two macros are required to set the two configuration registers in the 16-bit PIC® microcontrollers. `_CONFIG1(x)` is used to setup the 24-bit CONFIG1 register and `_CONFIG2(x)` is used to setup the 24-bit CONFIG2 register. The macro parameters are formed by bitwise ANDing the desired configuration values together. The configuration values are represented by a series of labels defined in the device specific header files (near the end of the file), along with examples and a limited amount of documentation. A fuller description of the configuration settings is contained in the device specific data sheets.

It is strongly recommended that you setup device configuration bits using the `_CONFIG1(x)` and `_CONFIG2(x)` macros because of the many advantages they have over setting them manually in the IDE.

Lab Exercise 3

"Hello, world!" for Microcontrollers



Purpose

The purpose of this lab is to test your understanding of the methods used to read and write I/O pins in C. Upon completion of this lab, you should be able to set or clear an output pin and read a switch by using a switch debounce function.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC:	C:\MTT\TLS2130\...

Objective

Write a short program that will turn on LED D3 (RA0) while switch S3 (RD6) is pressed. The LED should turn off only when the switch is released.

A switch debounce function will be provided for you, and it should be used to read the switch on RD6 to prevent the false detection of a button press or release. The full source code of this function is provided on the class CD.

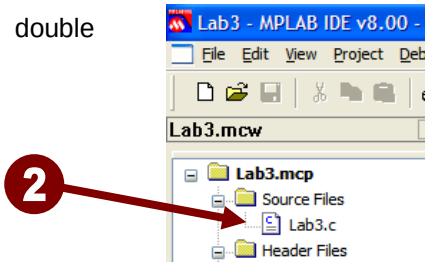
Procedure

1 If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open Lab 3 by selecting from the menu:
File ▸ Open Workspace...
and opening the workspace file located at:

C:\MTT\TLS2130\Lab3\Lab3.mcw

2 In the project tree, under the **Source Files** heading, double click the file **Lab3.c** to open it in the editor.



Lab2.c

```
13  #include <p24fj128ga010.h>
14  #include "TLS2130.h"
15
16  _CONFIG1(FWDTEN_OFF & JTAGEN_OFF)
17
18  int main(void)
19  {
20   3 //### YOUR CODE HERE ###          //### (3) Clear RA0 output latch
21   4 //### YOUR CODE HERE ###          //### (4) Make RA0 an output
22     while(1)
23     {
24   5     //### YOUR CODE HERE          //### (5) Write 0 to RA0
25   6     while(### YOUR CODE HERE###)  //### (6)Read RD0
26   7         //### YOUR CODE HERE ###  //### (7) Write 1 to RA0
27     }
28  }
```

Note: The line numbers above may be different from those in the actual source code file.

3 Clear the RA0 output latch. At this point, all of the I/O pins are configured as inputs. However, the output latches which are not presently connected to an output pin, may contain random data. Because of this, it is desirable to set them to a known value (logic 0 in this case) so that LEDs don't unexpectedly light up the moment we connect an output latch to a pin by clearing the appropriate bit in the TRIS register for this port.

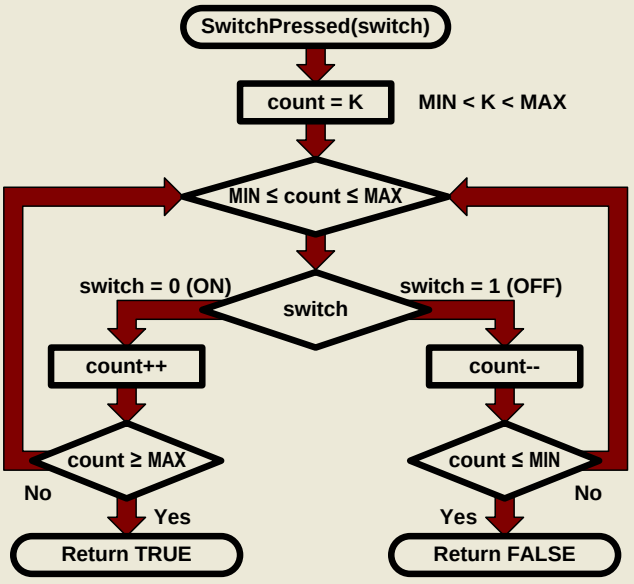
4 Make RA0 an output. (Hint: TRISx)

5 Write a 0 to RA0.

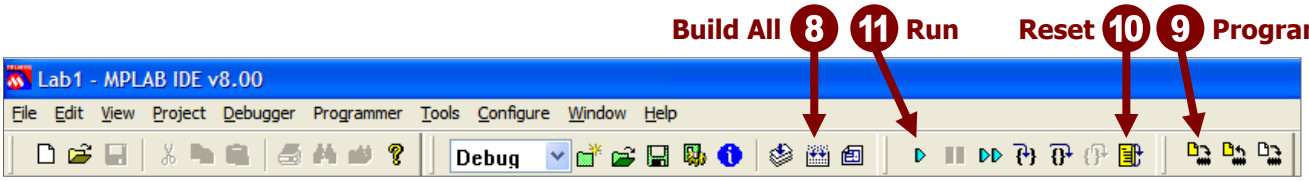
6 Read the switch on pin RD6. Because RD6 is connected to a mechanical switch as opposed to a digital signal, a debounce routine is required to filter out the noise generated by a switch press or release to prevent false detection of a state change. The function **`SwitchPressed()`** has been provided for this purpose. The function takes the name of an input pin that is connected to a mechanical switch, and returns the state of that switch: 1 for on (pressed), 0 for off (released). Use this function to determine the current state of switch S3. This function is included in the archive TLS2130.a, and the source code is provided on the class CD.

SwitchPressed() Function from TLS2130.c

```
unsigned char SwitchPressed(volatile unsigned int *sw, int bit)
{
    unsigned char count = COUNT;

    while ((MIN_COUNT <= count) && (count <= MAX_COUNT))
    {
        if (! (*sw & (1 << bit)))
        {
            count++;
            if (count > MAX_COUNT)
            {
                count = MAX_COUNT;
                return 1;
            }
        }
        else
        {
            count--;
            if (count < MIN_COUNT)
            {
                count = MIN_COUNT;
                return 0;
            }
        }
    }
}
```



7 Write a 1 to RA0.



8 Click on the **Build All** button.

9 If no errors are reported, click on the **Start Simulation** or **Program** button.

OR

10 When programming completes, click on the **Reset** button.

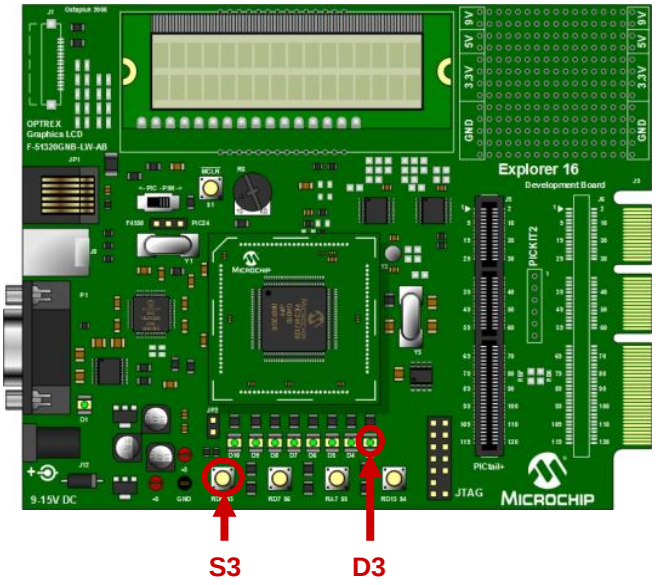
11 Click on the **Run** button.

Results

Press and hold down switch S3. LED D3 should illuminate and stay on as long as you hold down S3.

When you release S3, the LED should turn off and stay off until you press and hold S3 again.

Click on the **Halt** button in MPLAB when you are finished.



Conclusions

Working with I/O pins in C is very straight forward as long as the appropriate device header file is included (e.g. **`p24fj128ga010.h`**). Pin configuration is handled almost entirely by the **`TRISx`** register associated with a particular I/O port. In circumstances where an I/O pin is multiplexed with an analog input, you may need to modify another register to have digital access to the pin since they are configured as analog by default. Reading an input pin is accomplished when the name of the pin is used on the right side of an assignment operator or anywhere a variable value would be read. When you read an input pin, you use the **`PORTx.Rxn`** construct, or the shorthand **`_Rxn`** macro label. When you want to read an entire port, just use the entire port name such as **`PORTA`**. Writing to an output pin occurs when the name of the pin is on the left side of an assignment operator. When you write an output pin, you can use either the **`PORTx.Rxn`** or **`_Rxn`** constructs, or preferably the **`LATx.LATxn`** or **`_LATxn`** constructs. When you want to write to an entire port, just use the port name or port latch name such as **`PORTA`** or preferably **`LATA`**. All of these pin names/labels may be used as any other variable in C. Though, unlike an ordinary C variable, the values of the I/O ports may be changed by forces outside the control of your program (i.e. the hardware).

Lab Exercise 4

Interrupts

Purpose

The purpose of this lab is to reinforce this section's lessons on how to write an interrupt service routine that will properly handle an event generated by one of the microcontroller's internal peripherals.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC or CD:	C:\MTT\TLS2130\... or D:\TLS2130\...

Objective

Using the Timer 1 interrupt, make LED D3 (RA0) blink at a rate determined by the internal RC oscillator's default configuration and Timer 1's period with a 1:8 prescaler and Fosc/2 as Timer 1's clock source. The code that configures Timer 1 is already written for you, and accomplishes the following:

- Turns on Timer 1
- Configures Timer 1's prescaler for a 1:8 ratio
- Selects $F_{OSC}/2$ as Timer 1's clock input
- Clears Timer 1's interrupt flag
- Enables the Timer 1 interrupt with a priority level of 7

By default, the internal RC oscillator has a frequency of 8MHz with a 1:2 postscaler which divides the frequency down to 4MHz. This is the F_{OSC} that should be used in any formulas.

Your task will be to write the Timer 1 interrupt service routine which will toggle the LED and clear the interrupt flag each time the interrupt is triggered by the Timer 1 registers rolling over from 65535 (2^{16}-1) to 0.

Procedure

1

If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open Lab 4 by selecting from the menu:
File ▸ Open Workspace...
and opening the workspace file:

C:\MTT\TLS2130\Lab4\Lab4.mcw

2

Open Lab4.c by double clicking on its icon in the project tree. Complete the assigned tasks by adding your code anywhere you see the comments "`//### Your Code Here ###`". All required reference information is included in this section or in one of the previous labs (e.g. Lab 3—Working with I/O Ports)

Lab4.c

```
#include <p24fj128ga010.h>

_CONFIG1(FWDTEN_OFF & JTAGEN_OFF)

/*############################################################################
 Timer 1 Interrupt Service Routine
 Task A: Write the Timer 1 interrupt function header
 Task B: Toggle the LED on RA0 (LATA0)
 Task C: Clear the Timer 1 interrupt flag
############################################################################*/

//### Your Code Here ###            //###(2.1) T1 Interrupt Function Header
{
      //### Your Code Here ###      //###(2.2) Toggle LED on RA0
      //### Your Code Here ###      //###(2.3) Clear Timer 1 Interrupt Flag
}

/*****************************************************************************
 Main Function
*****************************************************************************/
int main(void)
{
    T1CON = 0b1000000000010000;     // T1 ON, 1:8 Prescale, Fosc/2 input
    IFS0bits.T1IF = 0;              // Clear TMR1 interrupt flag
    IEC0bits.T1IE = 1;              // Enable TMR1 interrupt
    IPC0bits.T1IP = 7;              // Set interrupt priority

    _LATA0 = 1;
    _TRISA0 = 0;

    while(1);
}
```

Task 2.1: Write the Timer 1 Interrupt Function Header

Write the first line (header) of the interrupt function. Remember that interrupt functions cannot take any parameters nor can they return any data. Also, remember to use the pre-defined function name and that you must tag the interrupt with the appropriate attribute, otherwise it will not be handled properly by the compiler.

Task A Reference Information

Interrupt Attribute Syntax

```
void __attribute__((interrupt)) ISRName(void)
{
      <ISR Code Here>
}
```

Section 7.3.2 of MPLAB C30 User's Guide DS51284E

Timer 1 Interrupt Function Name

```
_T1Interrupt
```

Table 7-3 of MPLAB C30 User's Guide DS51284E

Task 2.2: Toggle the LED on RA0 (LATA0)

There are several ways this may be done. The most common method of toggling a bit is to use the exclusive or (XOR) operator (^). Any bit that is exclusive or'ed with the value 1 will yield the complement of that bit. In other words: `X ^ 1` → `~X`. Consider using the compound assignment operator `^=`. This isn't the most efficient way to toggle a bit, but it works—we'll cover an optimization trick later in the class.

Task 2.3: Clear the Timer 1 Interrupt Flag (T1IF)

Simply write a value of zero to the appropriate interrupt flag bit.

Task C Reference Information

IFS0: Interrupt Flag Status Register 0

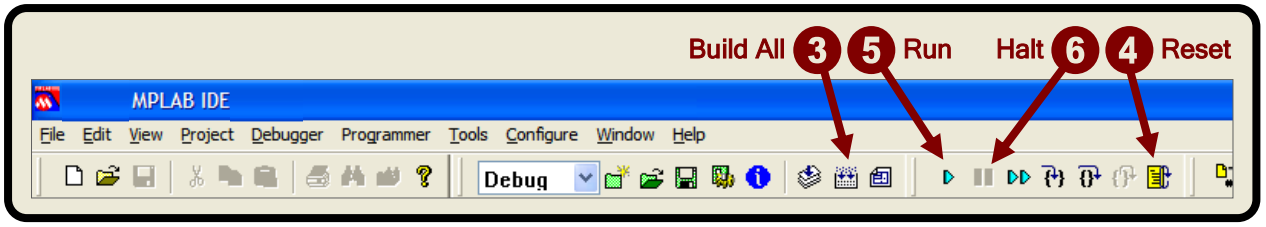
U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
-	-	AD1IF	U1TXIF	U1RXIF	SPI1IF	SPF1IF	T3IF
bit 15							bit 8

R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
T2IF	OC2IF	IC2IF	-	T1IF	OC1IF	IC1IF	INT0IF
bit 7							bit 0

bit 3 **T1IF:** Timer1 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

Table 6-2 and Register 6-5 of PIC24FJ128GA010 Family Data Sheet DS39747D

3. Once you have added your three lines of code, build the project by clicking on the Build All button. Fix any errors that are reported before you continue.



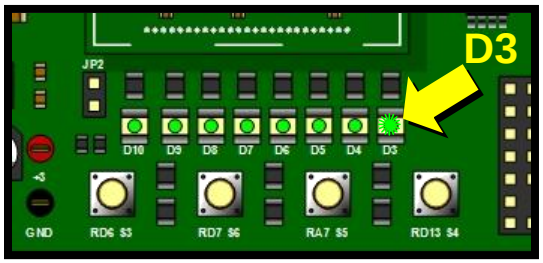
4. If no errors are reported, click on the Reset button.

5. Click on the run button.

6. When you are done, click on the halt button.

Results

LED D3 should be blinking at a rate of approximately twice per second.



By default, the oscillator selection is the internal RC oscillator with a 1:2 postscaler, so the final value of the clock oscillator is given by:
F_{OSC} = F_{RC} / CLKDIV = 8MHz / 2 = **4MHz**

By default, Timer 1's clock source is the instruction cycle clock (F_{OSC} / 2) with a 1:8 prescaler. So, one tick of Timer 1 is given by:
$Tick_{T1}$ = $(F_{OSC} / 2)^{-1} \cdot 8 = (4MHz / 2)^{-1} \cdot 8 =$ **4µs**

Since Timer 1 is a 16 bit timer, with a max count of 65536 (2^{16}), its timeout period is given by:
T_{T1} = $2^{16} \cdot Tick_{T1} = 65536 \cdot 4\mu s =$ **0.52s**

Conclusions

Aside from setting up the interrupts (which varies slightly from one peripheral to another), there are three things you need to do to handle interrupts:
1. Use the pre-defined interrupt name for the interrupt function identifier.
2. Tag the interrupt function with the interrupt attribute.
3. Clear the interrupt flag inside the interrupt function before you return.

There are many other things you can do with the interrupt attribute, such as doing a context save of variables you specify. For more details on how to take advantage of these additional features, see the compiler's user's guide for more information on the interrupt attribute.

Lab Exercise 5

Working with Peripheral Libraries

Purpose

The purpose of this lab is to reinforce this section's lessons on how to configure a peripheral with a given set of parameters by using the peripheral library functions included with MPLAB® C.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC or CD:	C:\MTT\TLS2130\... or D:\TLS2130\...

Objective

In the previous lab, Timer 1 was configured for you and you had to write the interrupt function. This time, the interrupt function is already written, and you have to configure Timer 1 with a given set of parameters using the configuration functions included in the 16-bit peripherals library that comes with the compiler. The end result should be the same as in the previous lab.

The `OpenTimer1()` function will be used to setup the following options:

- Turn on Timer 1
- Continue in Idle Mode
- Gate Off
- External Sync Off
- 1:8 Prescaler
- Clock source is $F_{OSC}/2$ (instruction cycle clock)

Procedure

1

If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open Lab 5 by selecting from the menu:
File ▸ Open Workspace...
and opening the workspace file:

C:\MTT\TLS2130\Lab5\Lab5.mcw

2

Open Lab5.c by double clicking on its icon in the project tree. Complete the assigned tasks by adding your code anywhere you see the comments "`//### Your Code Here ###`". All required reference information is included in this section or in one of the previous labs. Note that the **`timer.h`** header file has already been included for you.

Lab5.c

```
#include <p24fj128ga010.h>
#include <timer.h>

_CONFIG1(FWDTEN_OFF & JTAGEN_OFF)

void __attribute__((interrupt)) _T1Interrupt(void)
{
    __builtin_btg(&LATA, 0);
    IFS0bits.T1IF = 0;
}

int main(void)
{
    /*#####################################################################
      Using the OpenTimer1() and ConfigIntTimer1() functions, setup
      Timer 1 to operate with the settings outlined in the lab manual.
    #####################################################################*/

    //### Your Code Here ###      // ### Task (2.1) Setup Timer 1
    //### Your Code Here ###      // ### Task (2.2) Configure Interrupts

    _LATA0 = 1;
    _TRISA0 = 0;
    while(1);
}
```

Task 1: Configure and Enable Timer 1

Use the OpenTimer1() and function to setup Timer 1 with the following features / parameters:

- Timer 1 = ON
- Timer 1 Idle = Continue in Idle Mode
- Timer 1 Gate = OFF
- Timer 1 Prescale = 1:8
- Timer 1 Synchronization = External OFF
- Timer 1 Clock Source = Internal
- Timer 1 Period = 0xFFFF

Task 1 Reference Information

OpenTimer1() Function Prototype

```
void OpenTimer1(unsigned int config, unsigned int period);
```

OpenTimer1() *config* Parameter Options

Bitwise AND (**&**) the labels for the desired options to form the ***config*** parameter value:

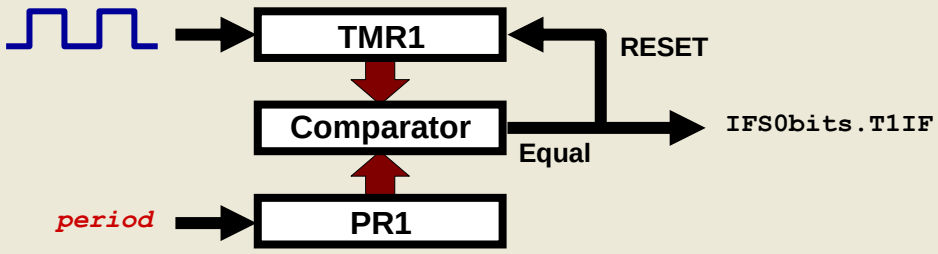
Timer Module On/Off	**Timer prescaler**
T1_ON	T1_PS_1_1 T1_PS_1_64
T1_OFF	T1_PS_1_8 T1_PS_1_128
Timer Module Idle mode On/Off	**Timer Synchronous clock enable**
T1_IDLE_CON	T1_SYNC_EXT_ON
T1_IDLE_STOP	T1_SYNC_EXT_OFF
Timer Gate time accumulation enable	**Timer clock source**
T1_GATE_ON	T1_SOURCE_EXT
T1_GATE_OFF	T1_SOURCE_INT

Example:

```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_GATE_OFF & T1_PS_1_8 &
           T1_SYNC_EXT_OFF & T1_SOURCE_INT, 0xFFFF);
```

OpenTimer1() *period* Parameter

The ***period*** parameter determines the value that will be loaded into the **PR1** register.



Example:

```
OpenTimer1(T1_ON & T1_IDLE_CON & T1_GATE_OFF & T1_PS_1_8 &
           T1_SYNC_EXT_OFF & T1_SOURCE_INT, 0xFFFF);
```

C:\Program Files\Microchip\MPLAB C30\docs\periph_lib\PIC24F TIMER Library Help File.htm

Task 2: Configure and Enable Timer 1 Interrupt

Use the ConfigIntTimer1() and function to setup the Timer 1 interrupt with the following features / parameters:

- Timer 1 Interrupt = Enabled
- Timer 1 Interrupt Priority Level = 7

Task 2 Reference Information

ConfigIntTimer1() Function Prototype

```
void ConfigIntTimer1(unsigned int config);
```

ConfigIntTimer1() *config* Parameter Options

Bitwise AND (**&**) the labels for the desired options to form the ***config*** parameter value:

Timer 1 Interrupt On/Off	Timer 1 Interrupt Priority
T1_INT_ON	T1_INT_PRIOR_7
T1_INT_OFF	T1_INT_PRIOR_6
	T1_INT_PRIOR_5
	T1_INT_PRIOR_4
	T1_INT_PRIOR_3
	T1_INT_PRIOR_2
	T1_INT_PRIOR_1
	T1_INT_PRIOR_0

Example:

```
ConfigIntTimer1(T1_INT_ON & T1_INT_PRIOR_4);
```

While the information above and on the previous page should be sufficient to complete the exercise, pages 5-5 through 5-11 contain the full documentation for the timer library routines. This text came from an html file included with the C compiler, and is installed in the following directory by default (along with documentation for all the other peripheral library routines):

C:\Program Files\Microchip\MPLAB C30\docs\periph_lib\

Link to: Peripheral Libraries Master Index (HTML)

PIC24F
TIMER Peripheral Module Library Help

C:\Program Files\Microchip\MPLAB C30\docs\periph_lib\PIC24F TIMER Library Help File.htm

Table Of Contents

1 Library Features

For this peripheral library module:

- The Timer1 module is a 16-bit timer , Timer2/3 and Timer4/5 modules are 32-bit timers, which can also be configured as four independent 16-bit timers with selectable operating modes.
- Timer 1 operates in CPU Idle modes and Sleep modes.
- Individually, Timer2/3 and Timer4/5 of the 16-bit timers can function as synchronous timers or counters.
- ADC Event Trigger is implemented only with Timer5.

2 Using Library Functions in Your Code

Library routine parameters can be constructed using either AND based mask or AND_OR based mask setting. For more information on these masks, see 16-bit Peripheral Libraries.

Examples of use for both the methods are below.

Example of Use (AND mask)

This is the method you should use for Lab Exercise 5

```
OpenTimer1((T1_ON & T1_IDLE_CON & T1_GATE_ON & T1_PS_1_8 &
            T1_SYNC_EXT_ON & T1_SOURCE_INT), PR1_VALUE);
```

Example of Use (AND_OR mask)

Function Call:

```
#define USE_AND_OR        /* To enable AND_OR mask setting */
#include<timer.h>
```

```
/*
 …
 User code
 …
*/
 OpenTimer1((T1_ON | T1_IDLE_CON | T1_GATE_ON | T1_PS_1_8 |
             T1_SYNC_EXT_ON | T1_SOURCE_INT), PR1_VALUE);

/*
 …
 User code
 …
*/
```

3 Functions

3.1 CloseTimerX (X = 1 … 5)

Function Prototype	`void CloseTimer1(void);`
	`void CloseTimer2(void);`
	`void CloseTimer3(void);`
	`void CloseTimer4(void);`
	`void CloseTimer5(void);`

Include	timer.h
Description	This function turns off the 16-bit timer module.
Arguments	None
Return Value	None
Remarks:	This function first disables the 16-bit timer interrupt and then turns off the timer module. The Interrupt Flag bit (TxIF) is also cleared.
Source File:	CloseTimer1.c CloseTimer2.c CloseTimer3.c CloseTimer4.c CloseTimer5.c

3.2 ConfigIntTimerX (X = 1 … 5)

Function Prototype	`void ConfigIntTimer1(unsigned int config);`
	`void ConfigIntTimer2(unsigned int config);`
	`void ConfigIntTimer3(unsigned int config);`
	`void ConfigIntTimer4(unsigned int config);`
	`void ConfigIntTimer5(unsigned int config);`

Include	timer.h
Description	This function configures the 16-bit timer interrupt.

Arguments *config* - Timer interrupt priority and enable/disable information as defined below:

```
Tx_INT_PRIOR_7
Tx_INT_PRIOR_6
Tx_INT_PRIOR_5
Tx_INT_PRIOR_4
Tx_INT_PRIOR_3
Tx_INT_PRIOR_2
Tx_INT_PRIOR_1
Tx_INT_PRIOR_0
Tx_INT_ON
Tx_INT_OFF
```

Return Value None

Remarks: This function clears the 16-bit Interrupt Flag (TxIF) bit and then sets the interrupt priority and enables/disables the interrupt.

Source File: ConfigIntTimer1.c
ConfigIntTimer2.c
ConfigIntTimer3.c
ConfigIntTimer4.c
ConfigIntTimer5.c

3.3 OpenTimerX (X = 1 … 5)

Function Prototype

```
void OpenTimer1(unsigned int config, unsigned int period);
void OpenTimer2(unsigned int config, unsigned int period);
void OpenTimer3(unsigned int config, unsigned int period);
void OpenTimer4(unsigned int config, unsigned int period);
void OpenTimer5(unsigned int config, unsigned int period);
```

Include timer.h

Description This function configures the 16-bit timer module.

Arguments *config* - The parameters to be configured in the TxCON register as defined below:

Timer Module On/Off
```
Tx_ON
Tx_OFF
```
Timer Module Idle mode On/Off
```
Tx_IDLE_CON
Tx_IDLE_STOP
```
Timer Gate time accumulation enable
```
Tx_GATE_ON
Tx_GATE_OFF
```
Timer prescaler
```
Tx_PS_1_1
Tx_PS_1_8
Tx_PS_1_64
Tx_PS_1_128
```
Timer Synchronous clock enable
```
Tx_SYNC_EXT_ON
Tx_SYNC_EXT_OFF
```
Timer clock source
```
Tx_SOURCE_EXT
Tx_SOURCE_INT
```

period - The period match value to be stored into the PR register

Return Value	None
Remarks:	This function configures the 16-bit Timer Control register and sets the period match value into the PR register.
Source File:	OpenTimer1.c OpenTimer2.c OpenTimer3.c OpenTimer4.c OpenTimer5.c

3.4 ReadTimerX (X = 1 … 5)

Function Prototype	`unsigned int ReadTimer1(void);`
	`unsigned int ReadTimer2(void);`
	`unsigned int ReadTimer3(void);`
	`unsigned int ReadTimer4(void);`
	`unsigned int ReadTimer5(void);`

Include	timer.h
Description	This function reads the contents of the 16-bit Timer register.
Arguments	None
Return Value	None
Remarks:	This function returns the contents of the 16-bit TMR register.
Source File:	ReadTimer1.c ReadTimer2.c ReadTimer3.c ReadTimer4.c ReadTimer5.c

3.5 WriteTimerX (X = 1 … 5)

Function Prototype	`void WriteTimer1(unsigned int timer);`
	`void WriteTimer2(unsigned int timer);`
	`void WriteTimer3(unsigned int timer);`
	`void WriteTimer4(unsigned int timer);`
	`void WriteTimer5(unsigned int timer);`

Include	timer.h
Description	This function writes the 16-bit value into the Timer register.
Arguments	*timer* - The 16-bit value to be stored into TMR register.
Return Value	None
Remarks:	None
Source File:	WriteTimer1.c WriteTimer2.c WriteTimer3.c WriteTimer4.c WriteTimer5.c

3.6 CloseTimer23 , CloseTimer45

Function Prototype
```
void CloseTimer23 (void);
void CloseTimer45 (void);
```
Include timer.h

Description This function turns off the 32-bit timer module.

Arguments None

Return Value None

Remarks: This function disables the 32-bit timer interrupt and then turns off the timer module.The Interrupt Flag bit (TxIF) is also cleared.
CloseTimer23 turns off Timer2 and disables Timer3 Interrupt.
CloseTimer45 turns off Timer4 and disables Timer5 Interrupt.

Source File: CloseTimer23.c
CloseTimer45.c

3.7 ConfigIntTimer23 , ConfigIntTimer45

Function Prototype
```
void ConfigIntTimer23(unsigned int config);
void ConfigIntTimer45(unsigned int config);
```
Include timer.h

Description This function configures the 32-bit timer interrupt.

Arguments *config* - Timer interrupt priority and enable/disable information as defined below:

```
Tx_INT_PRIOR_7
Tx_INT_PRIOR_6
Tx_INT_PRIOR_5
Tx_INT_PRIOR_4
Tx_INT_PRIOR_3
Tx_INT_PRIOR_2
Tx_INT_PRIOR_1
Tx_INT_PRIOR_0
Tx_INT_ON
Tx_INT_OFF
```

Return Value None

Remarks: This function clears the 32-bit Interrupt Flag (TxIF) bit and then sets the interrupt priority and enables/disables the interrupt.

Source File: ConfigIntTimer23.c
ConfigIntTimer45.c

3.8 OpenTimer23 , OpenTimer45

Function Prototype
```
void OpenTimer23(unsigned int config,
   unsigned long period);
void OpenTimer45(unsigned int config,
   unsigned long period);
```
Include timer.h

Description This function configures the 32-bit timer interrupt.

Arguments	*config* - This contains the parameters to be configured in the TxCON register as defined below: Timer module On/Off **Tx_ON** **Tx_OFF** Timer Module Idle mode On/Off **Tx_IDLE_CON** **Tx_IDLE_STOP** Timer Gate time accumulation enable **Tx_GATE_ON** **Tx_GATE_OFF** Timer prescaler **Tx_PS_1_1** **Tx_PS_1_8** **Tx_PS_1_64** **Tx_PS_1_128** Timer Synchronous clock enable **Tx_SYNC_EXT_ON** **Tx_SYNC_EXT_OFF** Timer clock source **Tx_SOURCE_EXT** **Tx_SOURCE_INT** *period* - This contains the period match value to be stored into the 32-bit PR register.
Return Value	None
Remarks:	This function configures the 32-bit Timer Control register and sets the period match value into the PR register.
Source File:	OpenTimer23.c OpenTimer45.c

3.9 ReadTimer23 , ReadTimer45

Function Prototype	**unsigned long ReadTimer23(void);** **unsigned long ReadTimer45(void);**
Include	timer.h
Description	This function reads the contents of the 32-bit Timer register.
Arguments	None
Return Value	None
Remarks:	This function returns the contents of the 32-bit TMR register.
Source File:	ReadTimer23.c ReadTimer45.c

3.23b WriteTimer23 , WriteTimer45

Function Prototype	**void WriteTimer23(unsigned long *timer*);** **void WriteTimer45(unsigned long *timer*);**
Include	timer.h
Description	This function writes the 32-bit value into the Timer register.

Arguments	*timer* - The 32-bit value to be stored into TMR register.
Return Value	None
Remarks:	This function returns the contents of the 32-bit TMR register.
Source File:	ReadTimer23.c ReadTimer45.c

4 Macros

4.1 EnableIntTX (X = 1 to 5)

Macro	**`EnableIntT1`** **`EnableIntT2`** **`EnableIntT3`** **`EnableIntT4`** **`EnableIntT5`**
Include	uart.h
Description	This macro enables the timer interrupt.
Arguments	None
Remarks	This macro sets Timer Interrupt Enable bit of Interrupt Enable Control register.

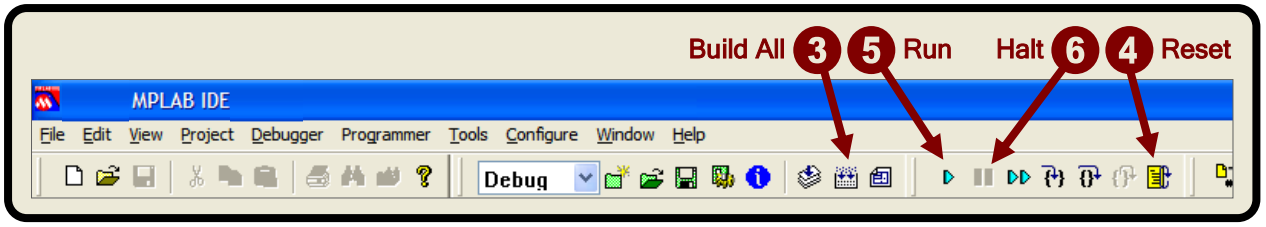
4.2 DisableIntTX (X = 1 to 5)

Macro	**`DisableIntT1`** **`DisableIntT2`** **`DisableIntT3`** **`DisableIntT4`** **`DisableIntT5`**
Include	uart.h
Description	This macro disables the timer interrupt.
Arguments	None
Remarks	This macro clears Timer Interrupt Enable bit of Interrupt Enable Control register.

4.3 SetPriorityIntTX (X = 1 to 5)(priority)

Macro	**`SetPriorityIntT1`** **`SetPriorityIntT2`** **`SetPriorityIntT3`** **`SetPriorityIntT4`** **`SetPriorityIntT5`**
Include	uart.h
Description	This macro sets priority for timer interrupt.
Arguments	priority
Remarks	This macro sets Timer Interrupt Priority bits of Interrupt Priority Control register.

3 Once you have added your two lines of code, build the project by clicking on the Build All button. Fix any errors that are reported before you continue.



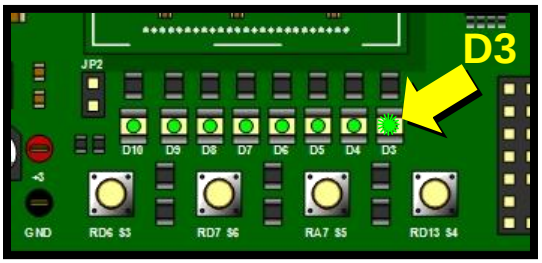
4 If no errors are reported, click on the Reset button.

5 Click on the run button.

6 When you are done, click on the halt button.

Results

LED D3 should be blinking at a rate of approximately twice per second.



By default, the oscillator selection is the internal RC oscillator with a 1:2 postscaler, so the final value of the clock oscillator is given by:
F_{osc} = F_{RC} / CLKDIV = 8MHz / 2 = **4MHz**

By default, Timer 1's clock source is the instruction cycle clock (F_{OSC} / 2) with a 1:8 prescaler. So, one tick of Timer 1 is given by:
$Tick_{T1}$ = $(F_{OSC} / 2)^{-1} \cdot 8 = (4MHz / 2)^{-1} \cdot 8 =$ **4µs**

Since Timer 1 is a 16 bit timer, with a max count of 65536 (2^{16}), its timeout period is given by:
T_{T1} = $2^{16} \cdot Tick_{T1} = 65536 \cdot 4\mu s =$ **0.52µs**

Conclusions

Using the peripheral libraries included with the compiler make peripheral configuration and use in C much easier. Many of the function parameters are formed by bitwise ANDing constants together to form the value(s) that ultimately will be loaded into the appropriate register(s). In order to use the libraries, all you need to do is include the appropriate header file associated with the peripheral of interest at the top of any file in your project that will make use of the library routines. The library / archive file is automatically included in all projects by the linker, so there is no need for you to manually add it to the project tree.

Lab Exercise 6

Creating Custom Libraries

Purpose

The purpose of this lab is to illustrate the steps required to create a custom library for use in your projects. You will learn how to open up multiple projects simultaneously in MPLAB to facilitate the coding and testing of custom library modules, and how to configure the compiler to build a library, rather than an ordinary hex file.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC or CD:	C:\MTT\TLS2130\... or D:\TLS2130\...

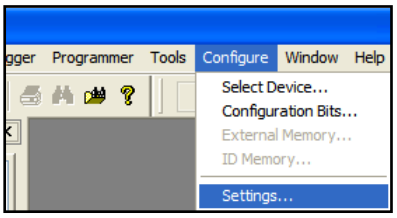
Objective

Build a library (archive) file from two simple C source files that contain a single function each. Then, use those functions in a separate project that includes the newly created library file in its project tree.

Procedure

1 If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open the MPLAB Settings by selecting from the menu:
Configure ▸ Settings...

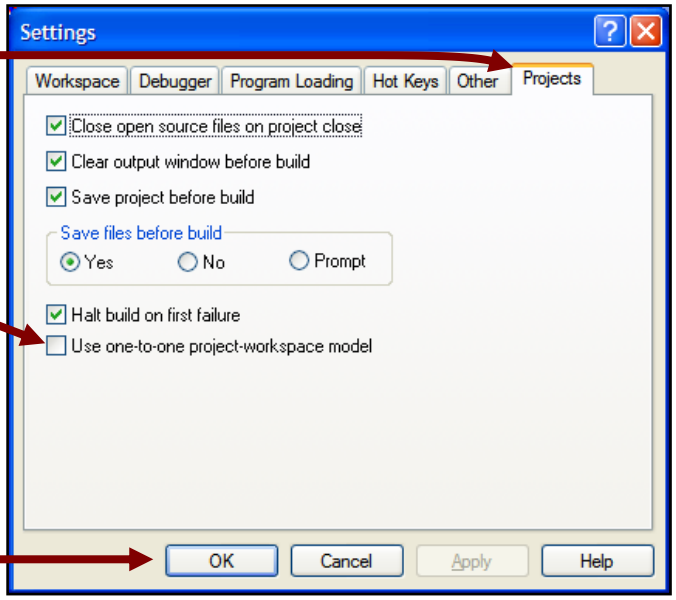


Configure MPLAB® so that multiple projects may be opened within one workspace

2 **Select the Project Settings**
On the tabs, select: **Projects**

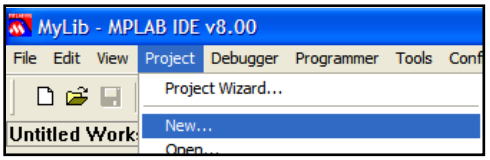
3 **Deselect One-to-One Project-Workspace Model**
Uncheck the last checkbox

Click OK **when done**



Create the Library Source Project

4 Create a new project by selecting from the menu:
Project ▸ New...



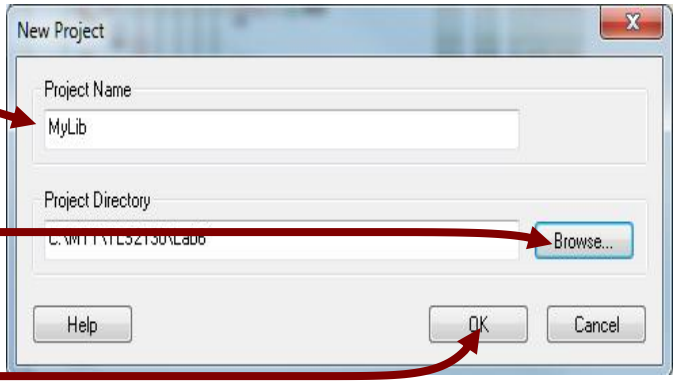
5 **Name the Project**
In the Project Name box, enter: **MyLib**

6 **Select the Project Directory**

Click Browse… and select

C:\MTT\TLS2130\Lab6

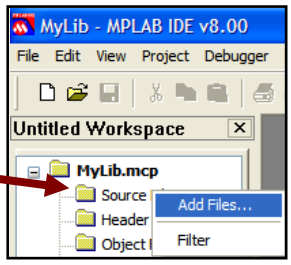
Click OK **when done**



The name you give the project ("MyLib"), will be the name of the library / archive file that will be created ("MyLib.a").

Add the library source code to the project

7 **Add Source File to Project**
In the project tree, right click on **Source Files** and select from the popup menu: **Add Files...**

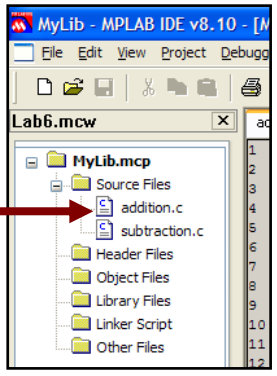


8 **Select the Library Source File**
In the add file dialog box, select the library source files from the lab 6 directory (the library code has already been written for you):

C:\MTT\TLS2130\Lab6\addition.c

C:\MTT\TLS2130\Lab6\subtraction.c

addition.c and subtraction.c should appear in the project tree under Source Files.



A Look at the Library Source Code and the Associated Header File

addition.c

```
int add(int a, int b)
{
        return (a + b);
}
```

An ordinary C function...

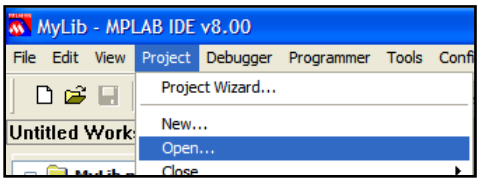
MyLib.h

```
int add(int a, int b);
int sub(int a, int b);
```

An ordinary function prototype...

Open or Create a Test Project

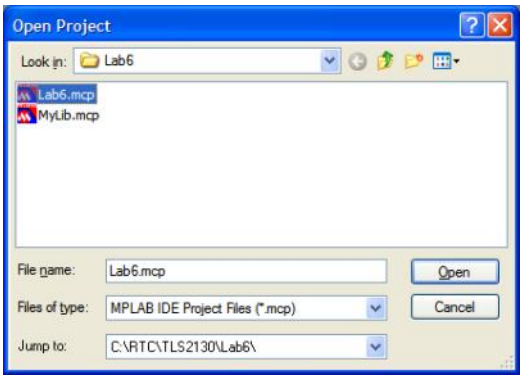
13 **Open Test Project**
From the menu bar select:
Project ▸ Open...



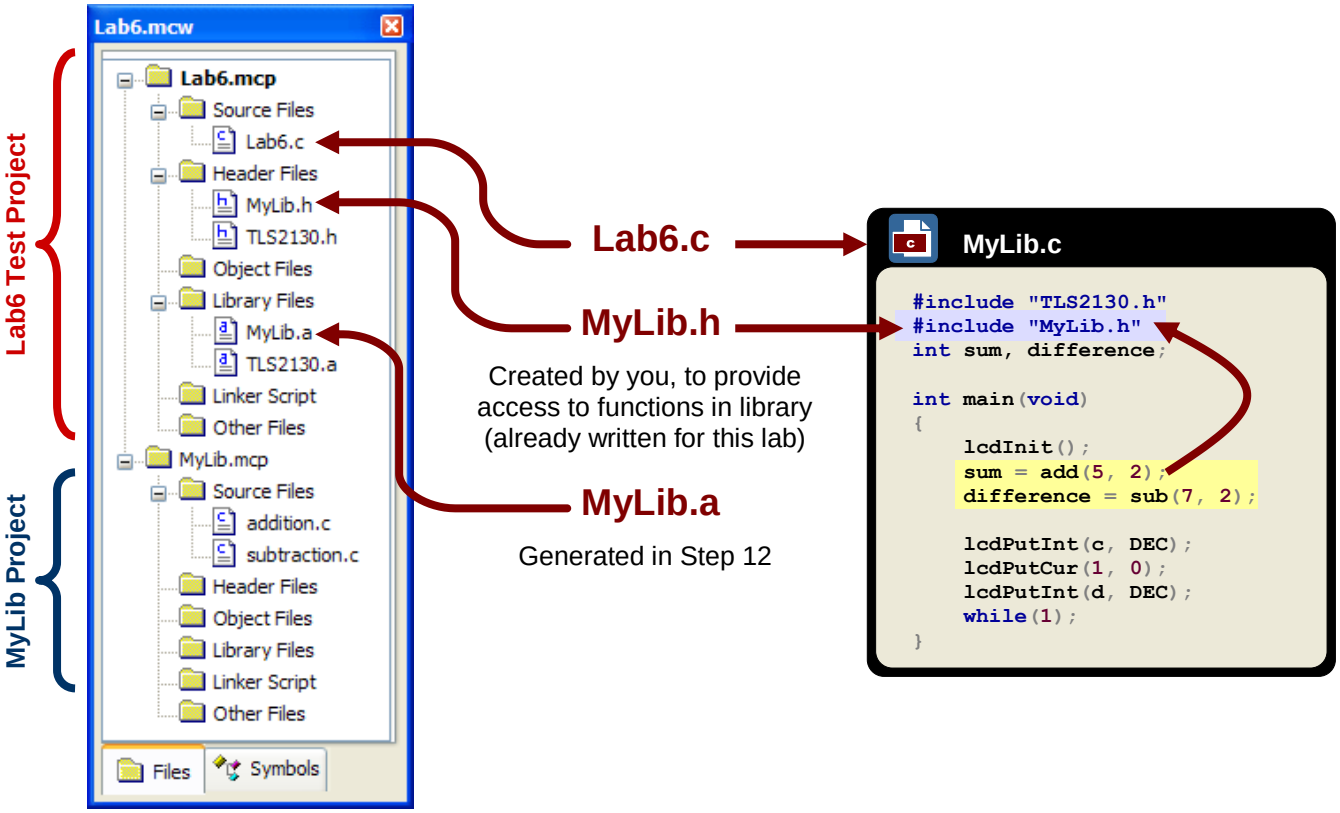
If doing this on your own, at this point you may create a new C30 based project following the steps from Lab 1. Include the library file just created into the new project.

14 **Select the Lab6 Project**
In the Open Project dialog box, select:

C:\MTT\TLS2130\Lab6\Lab6.mcp

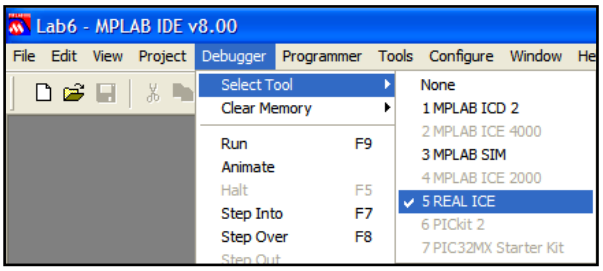


A Look at the Lab6 Test Project



Enable Your Debug Tool

15 **Enable MPLAB® REAL ICE™**
If not already enabled, From the menu bar select:
Debugger ▸ Select Tool ▸ 5 REAL ICE



Build and Run the Program

16 **Select Debug mode.**

17 **Click on the Build All button.**

18 If no errors are reported, click on the **Start Simulation** or **Program** button.

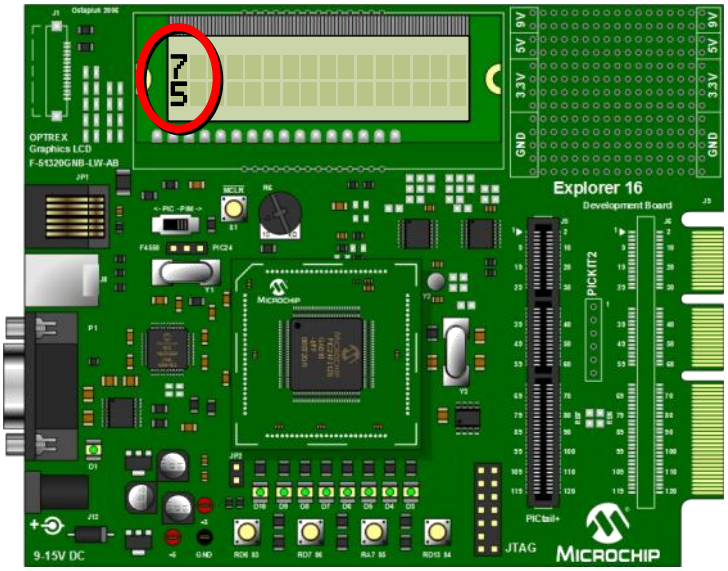
19 **When programming completes, click on the Reset button.**

20 **Click on the Run button.**

21 **Click on the Halt button.**

Results

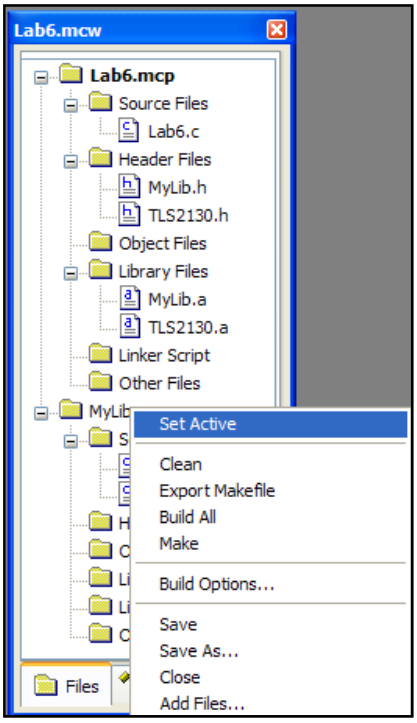
The number 7 should appear in the first position of the first line of the LCD and 5 should appear on the second line.



Conclusions

Library / Archive files can be created easily within MPLAB. As a general rule, nothing special needs to be done with the source code or the associated header file of a library file. The main concern is when you choose to build a generic library, you must be careful not to use features that are specific to a particular device. So, generic libraries are best suited for general algorithms and code that is not hardware specific.
Once the library code is written, simply build the project with a library target as the selected output. Once the *.a file is generated, it may be used in another project, assuming you have a header file with the appropriate function prototypes or extern declarations for variables that exist in the library file.

OPTIONAL—Continue Library Development



1. **Make Library Project Active**

   Right click on the MyLib project and select:
   Set Active

2. **Add to or modify library source code (and its associated header)**

3. **Rebuild Library as in Step 12**

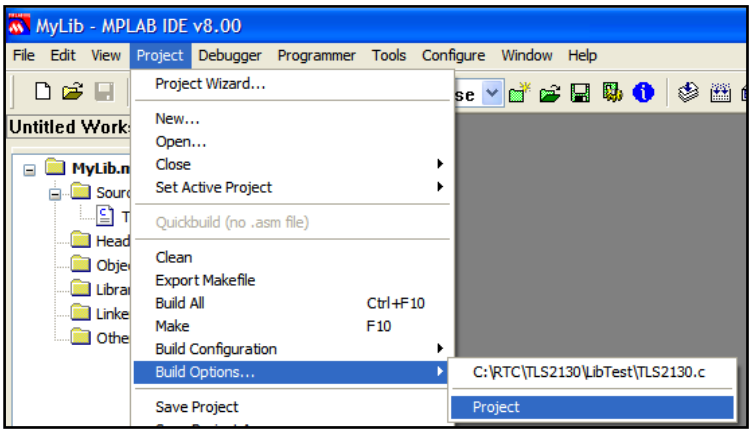
4. **Make Test Project Active**

   Right click on the Lab6 project and select:
   Set Active

5. **Rebuild and Run Lab6 as in Step 17 through 21**

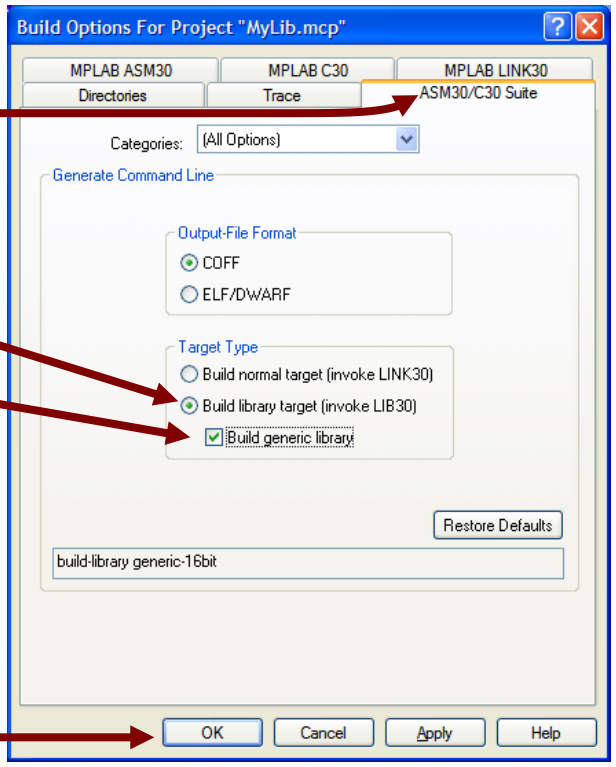
Configure the Project Build Options

9 **Open the Project Build Options Dialog Box**
From the menu bar select:
Project ▸ Build Options... ▸ Project



10 **Setup Tool Suite Options**
From the tabs select: ASM30 / C30 Suite

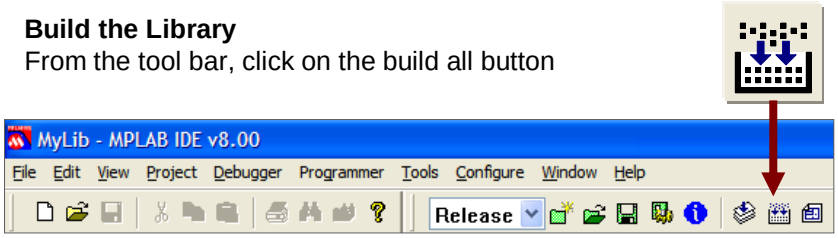
11 **Set the Target Type**
In the Target Type box select:
Build library target (invoke LIB30)
and check:
Build generic library



Click OK **when done**

Build the Library

12 **Build the Library**
From the tool bar, click on the build all button



This step will generate the library / archive file MyLib.a which may be used in any PIC24 based project.

Lab Exercise 7

Mixing C and Assembly

Purpose

The purpose of this lab is to illustrate how C and assembly files may be used in the same project, how C code can call assembly functions, and how parameters may be passed between them.

Requirements

Development Environment:	MPLAB 8.60 or later
C Compiler:	MPLAB C for PIC24, or MPLAB C for PIC24 and dsPIC (lite or better) v3.23b +
Hardware Tools:	• Explorer16 Development Board with PIC24FJ128GA010 • MPLAB Real ICE™ or MPLAB ICD3
Lab files on class PC or CD:	C:\MTT\TLS2130\... or D:\TLS2130\...

Objective

Given a project that contains an assembly language source file with two subroutines (AddFunction and MostSignificant1), and a C source file with a main function, add the necessary elements in the C source file to make the assembly subroutines callable from C.

Procedure

1

If you still have the previous project open, you must first close it by selecting from the menu:
File ▸ Close Workspace

Open Lab 4 by selecting from the menu:
File ▸ Open Workspace...
and opening the workspace file:

C:\MTT\TLS2130\Lab7\Lab7.mcw

A Look at the Assembly Language File

The assembly language file Lab7_asm.s include two complete subroutines plus all of the interface elements required to make them callable from C, assuming the appropriate things are done in the C file itself, which will be the focus of this lab. There are several elements of this file that deserve a detailed explanation.

a The name of the file includes the "_asm" because you cannot have both a C file and Assembly file in a project with the same name (not including the file extensions .c and .s). This is due to the fact that the compiler will generate an object file with the same name as the source file. For example, the file Lab7.c will be compiled into a file called Lab7.o, so we cannot use the name Lab7.s for our own assembly source file in this project because it too will be compiled into a file called Lab7.o. If we were to use the name Lab7.s, the project would compile, but would fail at the link step, producing an error reporting multiple definitions of our assembly subroutines.

b The second and third lines that contain the **`.global`** directives make the subroutine labels available to any file in the project. Without these directives, the assembly subroutines would be strictly local to this file.

c The subroutine names must start with an underscore '_' character to make them "visible" from the C code. Without the underscore, an assembly label may not be referenced from a C program.

Lab7_asm.s a

```
        .include "p24fj128ga010.inc"
    b   .global _AddFunction
        .global _MostSignificant1

        .text                          d
_AddFunction:                    d   d
        add w0,w1,w0            ; Add two integers and return the result
    c   return

_MostSignificant1:
        ff1l [w0], w0           ; Find first 1 from left (result is counted
    e                           ; from left starting with 1)
        subr w0, #16, w0        ; Adjust to give traditional bit position
                                ; number from right starting from 0
        return

        .end
```

d The **_AddFunction** subroutine works in conjunction with the compiler's parameter passing methodology. Since C functions that take two integer parameters pass them in the W0 and W1 registers, the first instruction of **_AddFunction** uses W0 and W1 as its two source operands. Similarly, C will return a single integer value from a function in the W0 register. So, the first instruction of **_AddFunction** uses W0 as its destination operand (add source1, source2, destination). The `return` statement corresponds to the closing bracket of a C function, so this will return the program to the place from where the function was called.

e The **_MostSignificant1** subroutine also works in conjunction with the compiler's parameter passing methodology and will retrieve its first and only parameter from the W0 register. However, note that in the assembly code, the register name is enclosed in square brackets: **[W0]** . This means that the value in W0 is being used as an address for indirect addressing—what is known as a pointer in C. Therefore, the function header that you will define shortly should ideally have its parameter defined as a pointer, since the function expects you to pass an address to it. (e.g. `foo(int *p);`)
Since the return value of this function is also an integer, it will be returned in the W0 register, just like was done for `_AddFunction` .

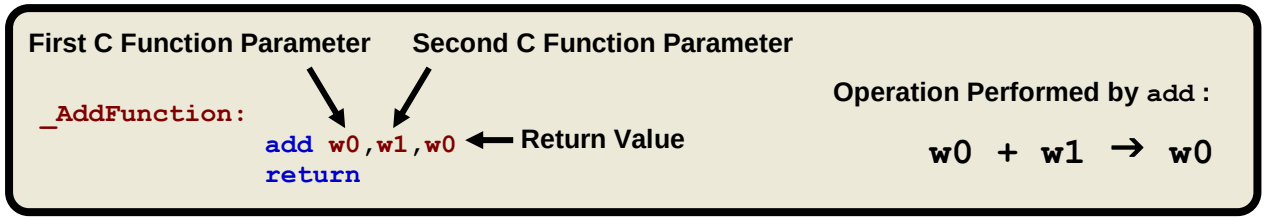
2 Open **Lab7.c** by double clicking on its icon in the project tree. Complete the assigned tasks by adding your code anywhere you see the comments "`//### Your Code Here ###`". All required reference information is included in this section or in one of the previous labs.

Lab7.c

```
#include <p24fj128ga010.h>
#include "TLS2130.h"

int a, b;
int c;

//### Your Code Here ###    // ### Task (2.1): Function prototype for
                            // _AddFunction (two int parameters)
//### Your Code Here ###    // ### Task (2.2): Function prototype for
                            // _MostSignificant1 (pointer parameter)
int main(void)
{
    lcdInit();
    a = 5;
    b = 3;

    //### Your Code Here ###  // ### Task (2.3): Call _AddFunction.
                              // Pass it a & b, store result in c

    lcdPutInt(c, DEC);        // Display value of c on first line of LCD
    lcdPutCur(1,0);           // Move cursor to second line of LCD

    //### Your Code Here ###  // ### Task (2.4): Call _MostSignificant1.
                              // Pass it the address of the variable a
                              // and store the result in c

    lcdPutInt(c, DEC);        // Display value of c on second line of LCD
    while (1);
}
```

Task 2.1: Write the _AddFunction C Prototype

Write a function prototype for the assembly subroutine `_AddFunction`. Because the assembly routine takes its inputs from the W0 and W1 registers, we can simply pass two integer parameters to `_AddFunction` since the compiler will use W0 and W1 for that purpose. The names of the parameters are irrelevant since they will not be used in the assembly function itself. Also, remember that the function name in C should NOT have the underscore in front of it as the assembly name does. The assembly function stores the results of the operation in the W0 register (16-bit value), which the compiler uses to pass return values back to the calling program in C.



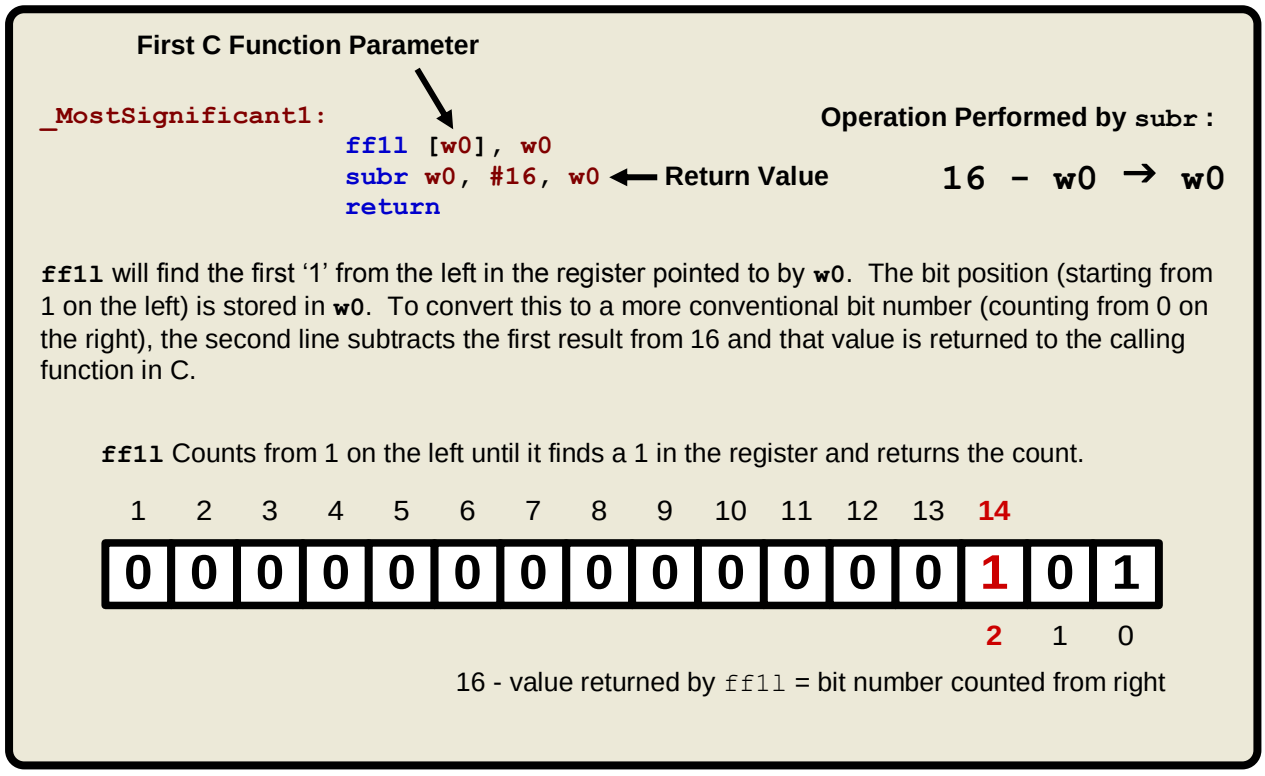
Function Prototype Refresher

Function prototypes are syntactically the same as a function header (1st line) with a semicolon at the end:

```
type functionName(type param1, type param2, … type paramN);
```

Task 2.2: Write the _MostSignificant1 C Prototype

Write a function prototype for the assembly subroutine `_MostSignificant1`. Because the assembly function uses the value in W0 in an indirect addressing operation, it would be best if we define this functions sole parameter as a pointer to integer (e.g. `int *p`). That way, when we call the function, we will pass an address to it, which can readily be used in assembly language for indirect addressing. Like `_AddFunction`, `_MostSignificant1` also returns a single 16-bit integer in the W0 register.



Task 2.3: Call _AddFunction

Based on the function prototype you wrote for Task A, call the assembly routine **`_AddFunction`**, and pass the variables **`a`** and **`b`** as its parameters.

Task 2.4: Call _MostSignificant1

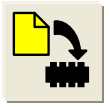
Based on the function prototype you wrote for Task B, call the assembly routine **`_MostSignificant1`**, and pass the address of variable **`a`** as its parameter.

Build and Run the Program

3 **Select Debug mode.**

4 **Click on the Build All button.**

5 If no errors are reported, click on the **Start Simulation** or **Program** button.



OR

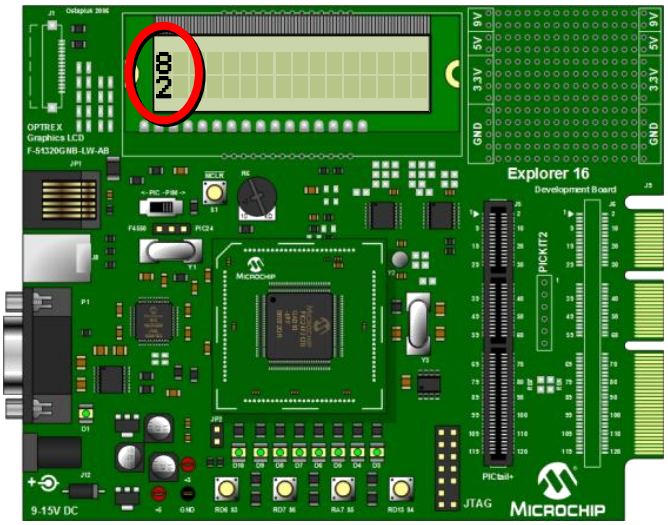
6 **When programming completes, click on the Reset button.**

7 **Click on the Run button.**

8 **Click on the Halt button.**

Results

The number 8 should appear in the first position of the first line of the LCD and the number 2 should appear in the first position of the second line.



Conclusions

Assembly language routines may be easily integrated into a C based project. There are a few straight forward requirements:

1. Assembly language subroutine labels must start with an underscore '_'.
2. Assembly language subroutine names must be made global with the `.global` directive.
3. A C function prototype with the same name of the assembly routine (without the underscore) must be written along with the parameters you wish to pass to the assembly routine via w0 through w7 (and possibly the stack if not enough space is available in the working registers).
4. Parameters are passed from left to right from C to the assembly routine in the lowest numbered, properly aligned working register from w0 to w7. Any data that doesn't fit will be pushed onto the stack. The order of the parameters may be rearranged based on their sizes and alignments.
5. Data is returned from assembly to C via the working registers, using as many as required from w0-w7 to hold the return value. (i.e. char or int in w0, long or float in w0 and w1, etc.)
6. The assembly routines should not modify any working registers from w8 to w15 without restoring them before returning to the C code.

Also, it is important that no two files in a project share the same name, not including the file type extension. In other words, it is not permissible to have MyFile.c and MyFile.s in the same project since the compiler will attempt to compile both of them into MyFile.o.

For more details, and additional cautions, please consult the MPLAB-C30 User's Manual.