MPLAB® XC8 Getting Started Guide

This document provides a starting point for programmers who are just starting out with the MPLAB® XC8 C Compiler, particularly those who are unfamiliar with embedded programming or Microchip devices.

The following headings are linked to sections in this guide:

Creation of a Project in MPLAB X IDE
Foundation Code
Compilation
Specifying Device Configuration Bits
Accessing Device Registers
Disabling Peripherals that Share Port Pins
Downloading and Running Your Code
Implementing a Main Loop
Using Interrupts
Conclusion

While the MPLAB XC8 C Compiler can target hundreds of 8-bit PIC devices, this guide uses the PIC18F87J11 microcontroller (MCU) with a PICDEM PIC18 Explorer Board. However, the information presented in this document can be used in conjunction with the XC8 C Compiler to create and compile equivalent code for almost any 8-bit MCU and hardware.

This guide describes using the compiler from the MPLAB X Integrated Development Environment (IDE); however, you can use it from the command-line, as well. If you have a development board, you can download and run code on your device. You can also use the simulator in MPLAB X IDE to confirm the operation of your code.

To demonstrate getting started with the MPLAB XC8 C Compiler, you will be guided through the creation of a project that you can build and run. The project flashes an LED that is connected to a port pin. To accomplish this, the following actions, presented here in summary, are performed. They are expanded and described in more detail as you progress through the pages of this guide.

- Include `<xc.h>` in your source file(s).
- Set the device Configuration bits using the `config` pragma.
- Disable any peripheral that uses the pin(s) used by the port.
- Initialize the port's data direction register, and write values to the port latch.
- Use a delay to ensure you can see the changes in state.

The guide assumes that you have MPLAB X IDE and MPLAB XC8 C compiler installed and activated (if applicable) before you commence. You could also use an evaluation version of the compiler, or the compiler operating in Free mode.

For assistance installing or activating the compiler, refer to *Installing and Licensing MPLAB XC C Compilers* (DS50002059). The document can be downloaded from the Microchip Technology web site, wwww.microchip.com.

CREATION OF A PROJECT IN MPLAB X IDE

This section describes how to create a project in MPLAB X IDE using the MPLAB XC8 C Compiler.

The process is explained in the following steps:

Step 1 sets the project type.

Step 2 selects the target device.

Step 3 selects the device header.

Step 4 selects the tool to run the project code.

Step 5 is only applicable to some debugger tool selections.

Step 6 selects the tool to compile the source code.

Step 7 specifies the project name and path.

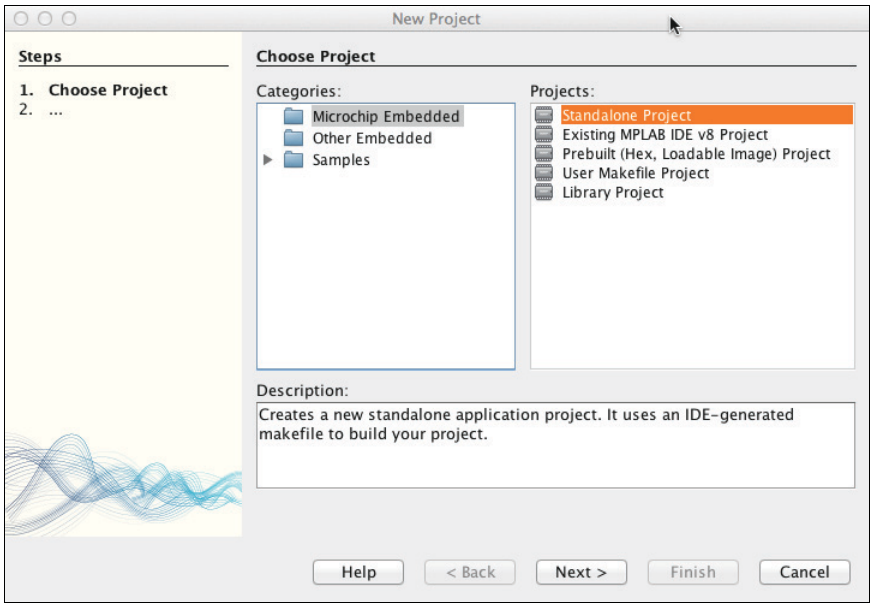
Step 8 completes creation of the project.

If you are not using the MPLAB X IDE, or you are already familiar with the process of creating a project, skip to the next section, **"Foundation Code"**. Full information on MPLAB X IDE is available online in the *MPLAB X IDE User's Guide* (DS52027).

Step 1 sets the project type.

From MPLAB X IDE, choose *File>New Project...*. In the window that opens (as shown in Figure 1-1), select the “Microchip Embedded” category, and a “Standalone Project” from the Projects field.

FIGURE 1-1: NEW PROJECT WINDOW

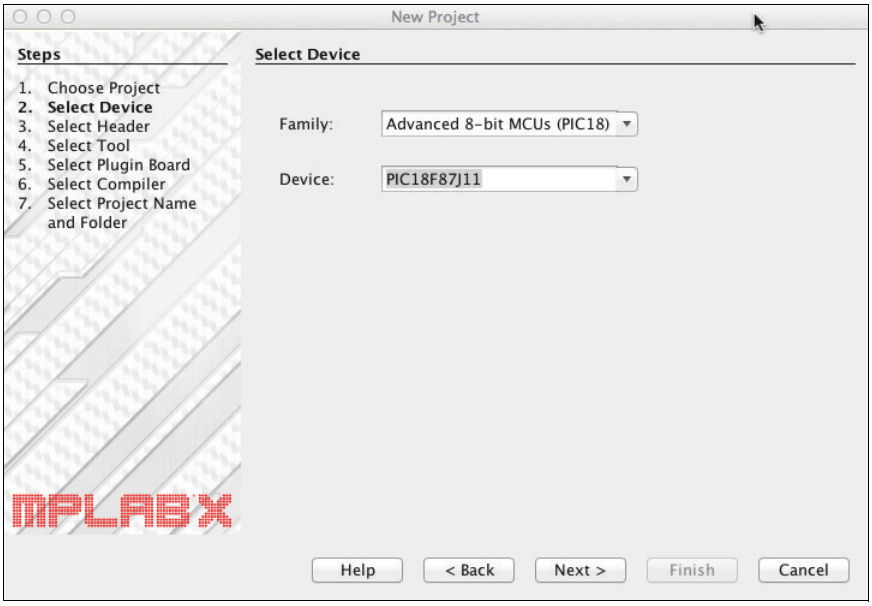


Step 2 selects the target device.

This selection must exactly match the device on your hardware. (If you are using the simulator without hardware, you can choose any device.)

To make selecting a device simpler, devices are organized into families. MPLAB XC8 can compile for any device in the 8-bit-microcontroller families. In Figure 1-2, a PIC18F87J11 has been selected from the PIC18 family.

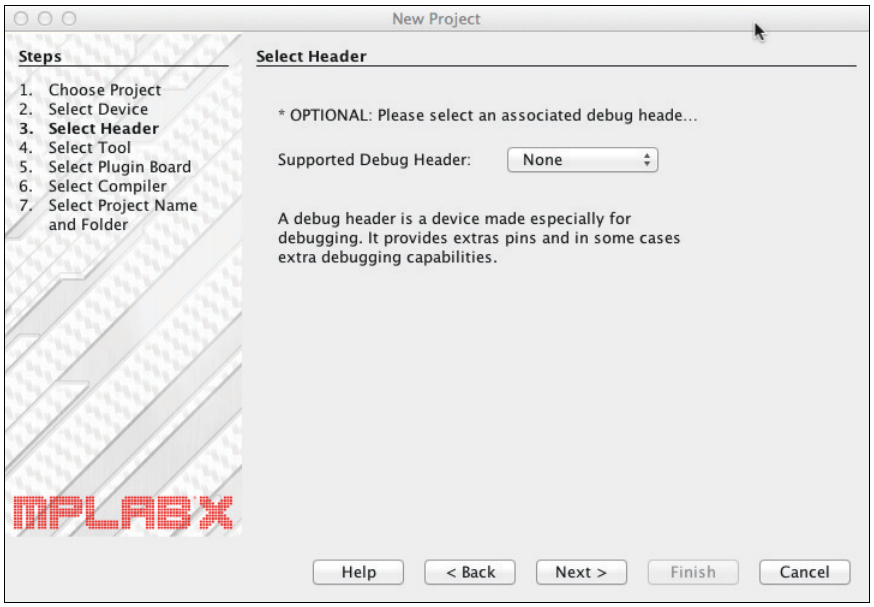
FIGURE 1-2: DEVICE SELECTION DIALOG



Step 3 selects the device header.

The use of debugging features is not required by this guide, so this selection could be None, as shown in Figure 1-3.

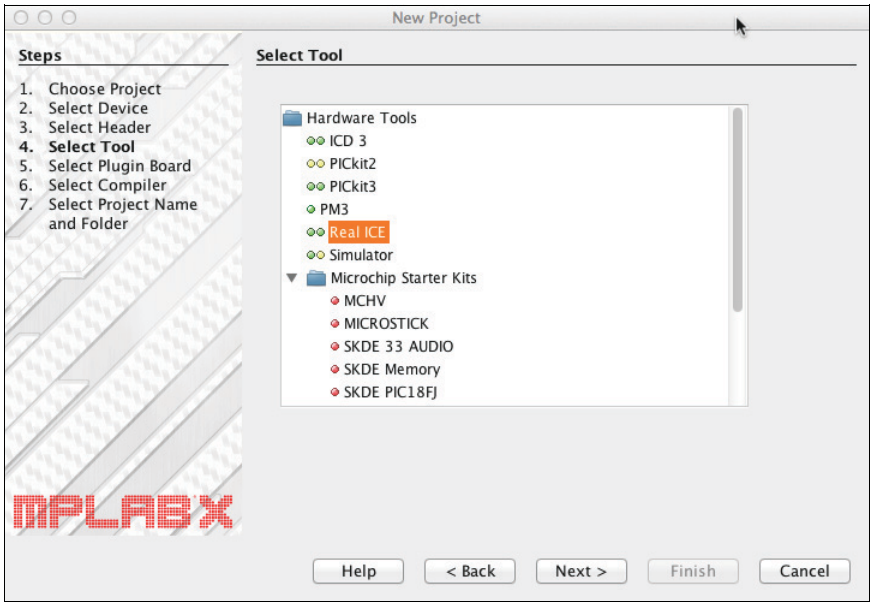
FIGURE 1-3: HEADER SELECTION DIALOG



Step 4 selects the tool to run the project code.

If you have a debugger and wish to use it with your hardware, select it from the list; otherwise, select Simulator. Figure 1-4 shows MPLAB REAL ICE selected as the programmer/debugger to run the generated code.

FIGURE 1-4: TOOL SELECTION DIALOG



Step 5 is only applicable to some debugger tool selections.

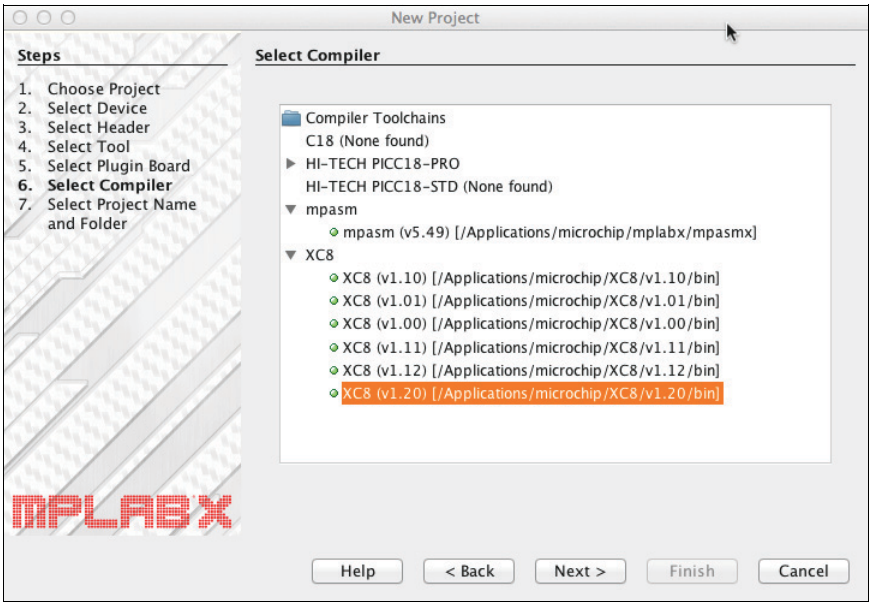
Unless you must use a specific plugin board, select None (when/if the dialog shown in Figure 1-5 appears).

FIGURE 1-5: PLUGIN SELECTION DIALOG

Step 6 selects the tool to compile the source code.

As shown in the Select Compiler window in Figure 1-6, several versions of the MPLAB XC8 compiler might be listed under the XC8 disclosure widget. Select the most recent version. You will be able to change this selection during development.

FIGURE 1-6: COMPILER SELECTION DIALOG

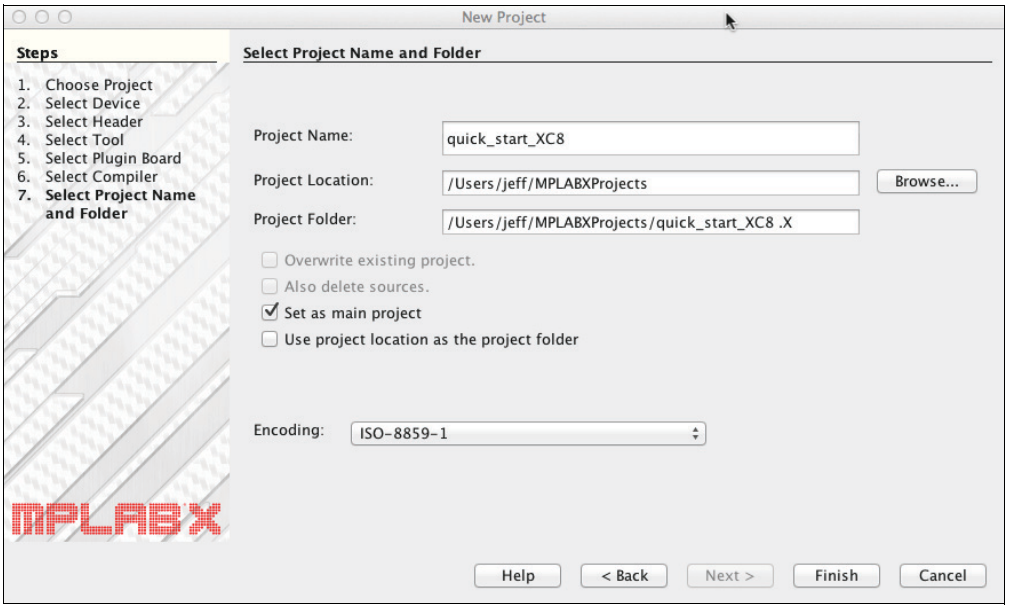


Step 7 specifies the project name and path.

Type a name for the project in the Project Name field. Click **Browse...** if the default project path is not suitable. In this case, the name `quick_start_XC8` has been chosen for illustration purposes and can be seen in Figure 1-7.

To differentiate the current project in the IDE (when multiple projects exist) as the main project, click “Set as main project”.

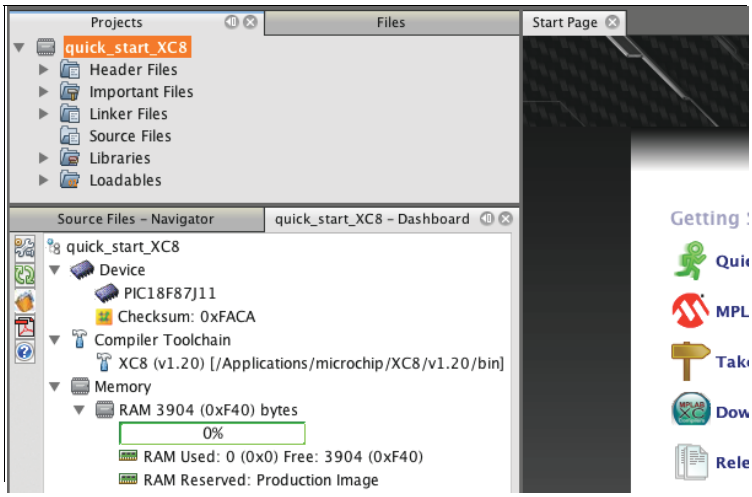
FIGURE 1-7: PROJECT NAME AND PATH DIALOG



Step 8 completes creation of the project.

Click **Finish,** and the project is created. An icon representing the project appears in the Projects window[1], as shown in Figure 1-8. The Projects window is shown at the top left in the figure. Below the Projects window, the Dashboard provides more detailed project information.

FIGURE 1-8: THE PROJECTS WINDOW



1. You may need to select *Windows>Projects* if this pane is not visible by default.

FOUNDATION CODE

The code presented here is actually a small program that could be the basis for all of your MPLAB XC8 projects. Although it could be construed as trivial, the code is entirely valid, and compiles and executes, as required.

The code creation process is explained in the following linked steps:

Step 1 creates a new source file.

Step 2 enters a suitable name for the source file.

Step 3 adds skeleton code to the new file.

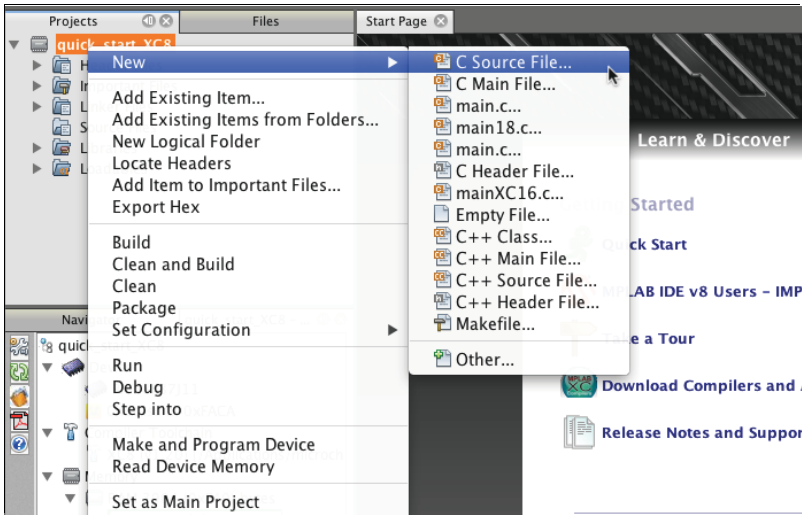
Step 4 saves your work.

Step 1 creates a new source file.

With MPLAB X IDE, there are several ways to create a source file. The following method is the most basic, and moves through the all aspects of the source creation process.

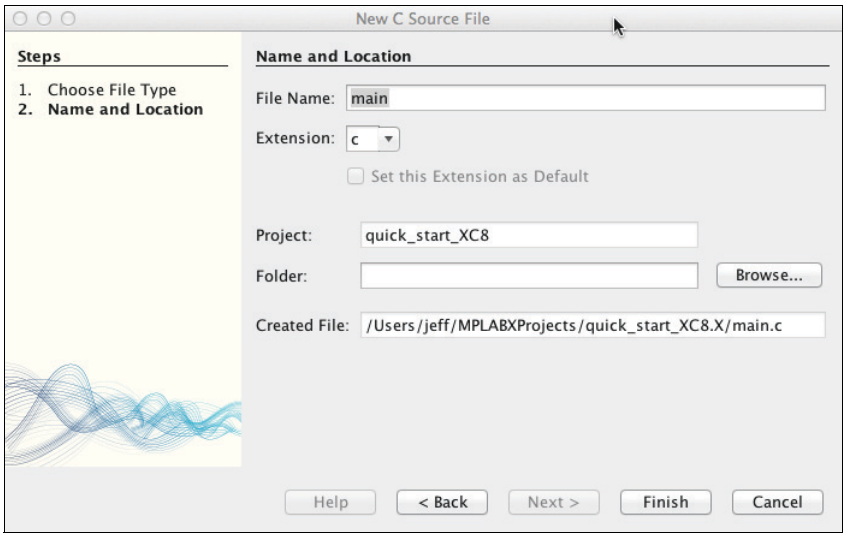
As shown in orange in Figure 1-9, right click on the project icon that represents the new project you created, `quick_start_XC8`. Select *New>C Source File...* on the resulting pop-up command and target lists.

FIGURE 1-9: NEW FILE POPUP



This will open the New C Source File window, as shown in Figure 1-10.

FIGURE 1-10: NEW C SOURCE FILE WINDOW



Step 2 enters a suitable name for the source file.

Ensure that the project name is correct.

As illustrated in Figure 1-10, the settings will create a file called `main.c`. After clicking **Finish**, an icon representing the file should appear in the Project window. The file is also opened in the text editor. At this time, the file is empty.

Step 3 adds skeleton code to the new file.

Copy or type the following text into the new source file, m`ain.c`.

```
#include <xc.h>

int main(void)
{
    return 0;
}
```

This initial code can be used to start every project built with MPLAB XC8. C.

Every C program must have one, and only one, function called `main()`; however, the exact prototype for this function can vary from compiler to compiler. For all MPLAB XC compilers, you may use the prototype shown above. Since `main()` returns an `int`, there must be a `return` statement with a return value specified. The value 0 indicates that `main()` returned successfully.

The inclusion of the header file, `<xc.h>`, allows code in this source file to access compiler- or device-specific features. Since such access is commonplace, you will need to include it into virtually all of your source files.

Step 4 saves your work.

Select *File>Save* to ensure that your work is saved.

If you are not using MPLAB X IDE, you may enter the above program into a file using any editor, as long as it is saved as plain text and the file uses the `.c` extension.

COMPILATION

As mentioned previously, the new program is a valid C program. This means it can be compiled. This section explains how to build code.

MPLAB X IDE knows which compiler to execute when building your source code, but options can be used to alter how the compiler operates. The default options are acceptable for most projects. If you do need to adjust the compiler options, you can do so from the Project Properties dialog. Open this dialog using the top button on the left in the project dashboard, as seen in Figure 1-11. From this dialog you can also change other project attributes, such as the device or compiler associated with your project.

FIGURE 1-11: PROJECT PROPERTIES BUTTON

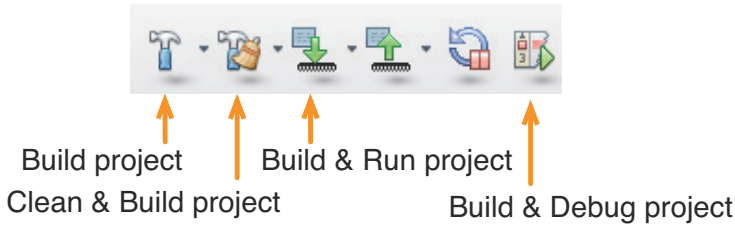
There are several ways to execute the compiler. There are buttons on the toolbar to enable quick access to the different build operations, but you can also access these from the *Run* and *Debug* menus. Some operations only build your code; others build and then *execute* your code. Both the build and run steps can be made in either a release or debug mode.

Debug mode operations enable the debug executive on your device. This allows access to debugging features like breakpoints. For the debug executive to be active, it must utilize some of the device's memory that normally would be available for your code. Debug builds ensure that this memory is reserved for the debug executive.

Release mode operations do not allow any debug features to be used, but all the device memory is available for your project. This is the build mode you would use to produce a production image suitable for products you intend to release.

Figure 1-12 shows the most common toolbar buttons that are used to build code.

FIGURE 1-12: BUILD BUTTONS



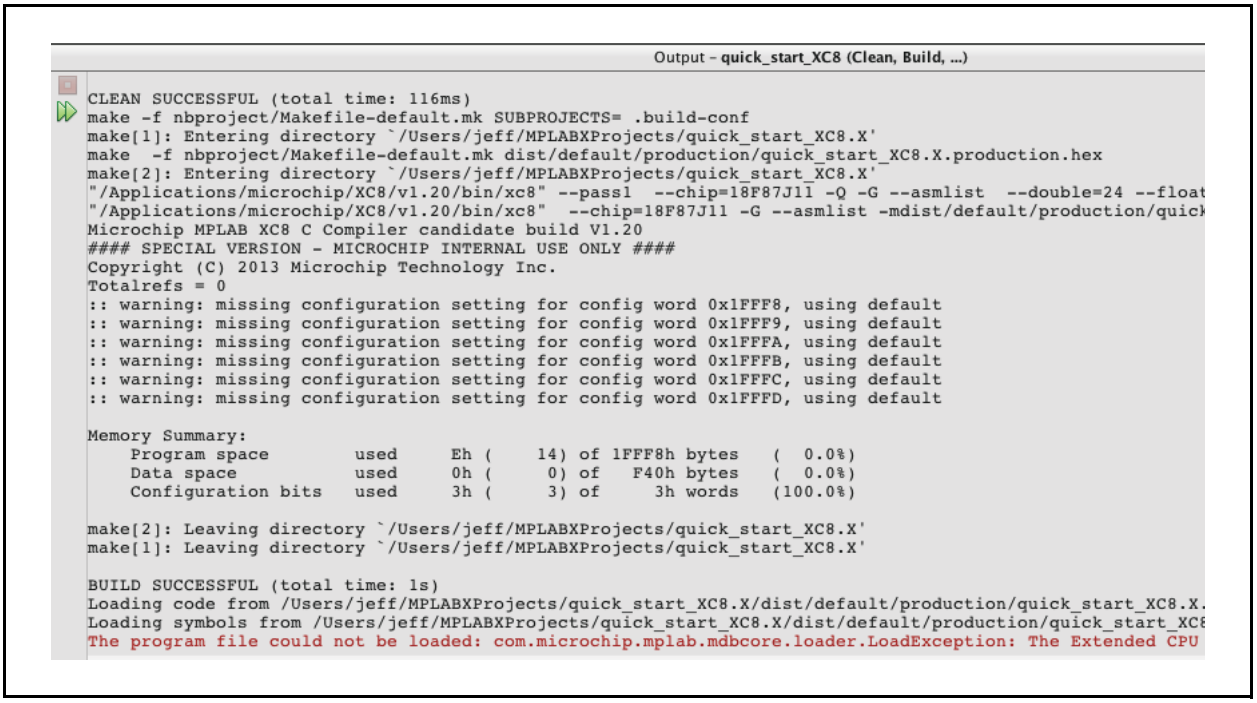
From left to right, the identified buttons perform the following functions:

- Build (release) any project source files modified since the last build, then link
- Build (release) *all* project source files, then link
- Build (release) any project source files modified since the last build, link, then download and run your code
- Build (debug) any project source files modified since the last build, link, then download and run your code with the debug executive enabled

For the purposes of this demonstration, click Build, or Clean and Build.

The compiler associated with your project will be invoked, and every source file in your project (currently only one) will be built and linked into one binary image. You will see a transcript of the build process in the Output window, which will open if it is not already visible in the workspace. This might look similar to that shown in Figure 1-13.

FIGURE 1-13: THE OUTPUT WINDOW



Note that there are several warnings relating to missing configuration settings[1], but these have not stopped the compilation process and the BUILD SUCCESSFUL message at the lower part of the window indicates that the code was built. The red error, at the bottom of the Output window, indicates a mismatch in the code compiled and the target device. The next section explains how to configure the device, which will resolve these warnings and the error.

If you are building from a terminal, use the following command line.

```
xc8 --chip=18f87j11 main.c
```

Adjust the device and source file name, as appropriate. If the compiler is not in the search path, you should use its full path, in addition to the application name, `xc8`.

1. If you are compiling for a device other than the PIC18F87J11, you may see more or fewer of these warnings

SPECIFYING DEVICE CONFIGURATION BITS

Although the new program is a valid C program, it is unlikely that it would run properly on the device. All Microchip 8-bit PIC devices must be configured to ensure correct operation. Some configuration settings affect fundamental operation of the device, like those for the instruction clock. If that setting is incorrect, the clock may not run.

The warnings shown in the Output window in the previous section can indicate potential problems with the device's configuration, but it is important to note that you must specify these configuration settings even if you do not see any warnings issued by the compiler.

The configuration settings are specified by special bits in the device.The MPLAB XC8 C Compiler uses pragmas that allow you to specify the configuration bit settings in your code. The values derived from these pragmas are merged with your project's compiled binary image and downloaded to your device.

The number and type of configuration settings vary from device to device. Consult the data sheet for the MCU you are using to learn what each setting controls. In this case, the device is a PIC18F87J11, the data sheet is the *PIC18F87J11 Family Data Sheet* (DS39778), and can be downloaded from www.microchip.com.

The easiest way to complete the pragmas that are needed to configure your device is to use the Configuration Bits Window, a feature of MPLAB X IDE. The following steps describe how to use the window to get the information to complete the pragmas.

Step 1 opens the Configuration Bits window.

Step 2 reviews every setting.

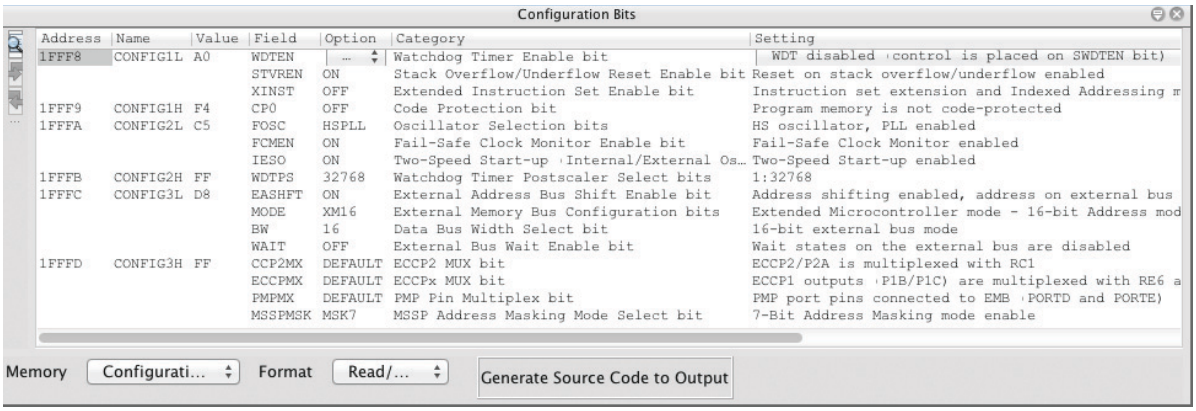
Step 3 generates the pragmas that can implement the settings you have chosen.

Step 4 copies the code from this window to a source file of your choosing.

Step 1 opens the Configuration Bits window.

Select *Window>PIC Memory Views>Configuration Bits* from the menus. The window, similar to the one that is shown in Figure 1-14, lists information that relates to Configuration bit locations and values.

FIGURE 1-14: THE CONFIGURATION BITS WINDOW



The Name and Field columns will help you to find the equivalent settings in the device data sheet.

The Category column describes what the setting controls.

The Setting column shows the current state of that setting.

Step 2 reviews every setting.

Pay particular note to the following settings, which will almost certainly cause runtime failure if not they are not specified correctly:

Oscillator section

This must match that of your hardware's oscillator circuitry. If this is not correct, the device clock may not run. If you are using the simulator as a debug tool, this setting can be ignored. Typically, development boards use high-speed crystal oscillators.

Watchdog timer

It is recommended that you disable this timer, until it is required. This prevents unexpected Resets.

Code protection

Turn off code protection, until it is required. This ensures that the device is fully accessible.

Extended instruction set

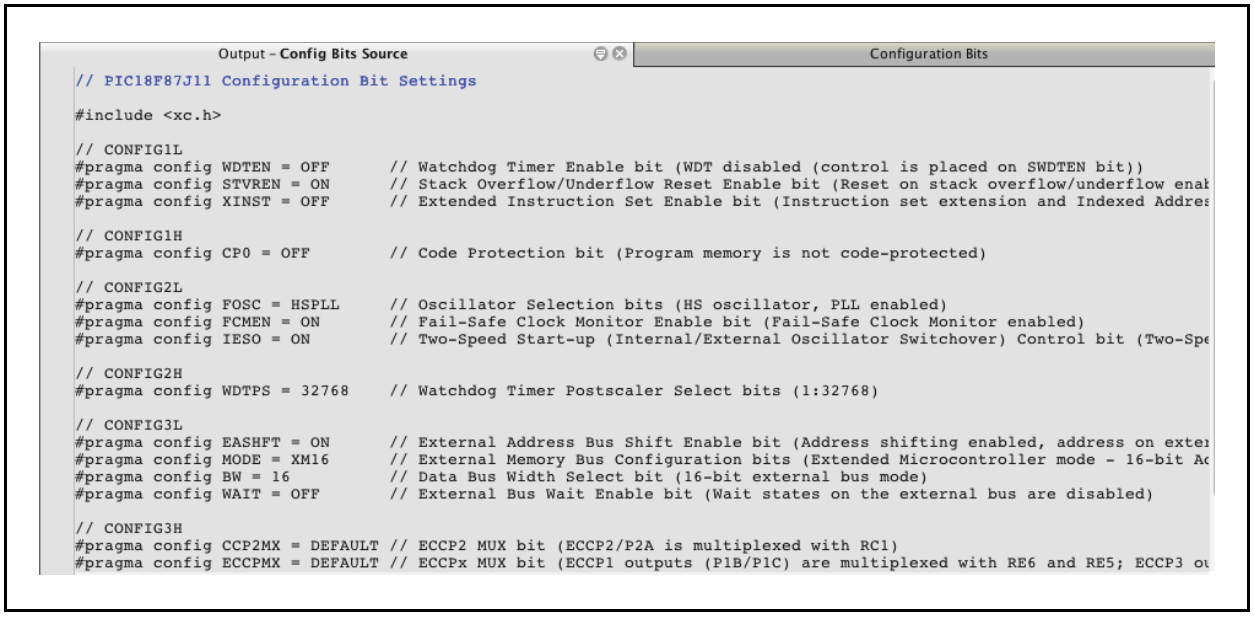
This PIC18 setting *must* be disabled. The MPLAB XC8 C Compiler does not support this instruction set.

Change the settings by clicking on the relevant line in the Settings column and choosing from the appropriate setting in the pull-down list.

Step 3 generates the pragmas that can implement the settings you have chosen.

Click the **Generate Source Code to Output** button. The generated code will appear in the Config Bits Source window, as shown in Figure 1-15.

FIGURE 1-15: THE CONFIGURATION BITS SOURCE WINDOW



Step 4 copies the code from this window to a source file of your choosing.

It is not executable code and should be placed outside of function definitions. The code has been copied to the `main.c` file (comments omitted for clarity), as shown below[1].

```
#include <xc.h>

// CONFIG1
#pragma config WDTEN = OFF
#pragma config STVREN = ON
#pragma config XINST = OFF
#pragma config CP0 = OFF

// CONFIG2
#pragma config FOSC = HSPLL
#pragma config FCMEN = ON
#pragma config IESO = ON
#pragma config WDTPS = 32768

// CONFIG3
#pragma config EASHFT = ON
#pragma config MODE = XM16
#pragma config BW = 16
#pragma config WAIT = OFF
#pragma config CCP2MX = DEFAULT
#pragma config ECCPMX = DEFAULT
#pragma config PMPMX = DEFAULT
#pragma config MSSPMSK = MSK7

int main(void)
{
    return 0;
}
```

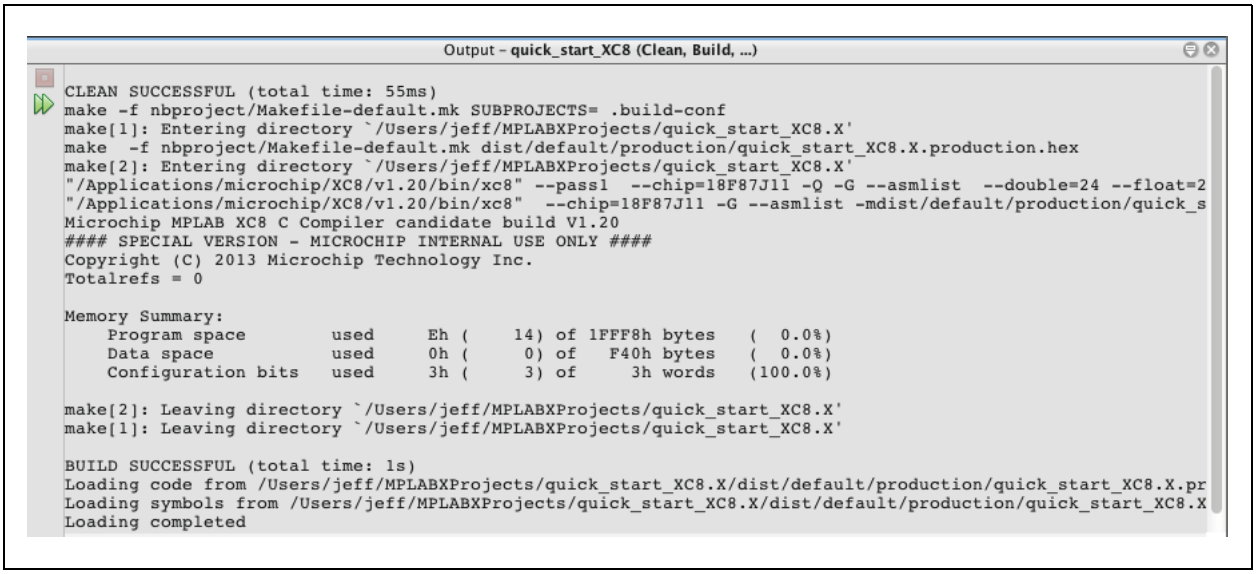
1. This code is specific to a PIC18F87J11 device and the PICDEM PIC18 Explorer board. You must use configuration settings that are specific to your device and hardware.

Note that there is not a link between the configuration pragmas in your source code and the Configuration Bits window. If you need to adjust a configuration setting, you must manually edit the pragmas in your source code. Or, change the settings in the Configuration Bits window, regenerate the source code, and replace your existing pragmas with those newly generated.

There is another place, outside of MPLAB X IDE, that you can find settings and values associated with supported Microchip devices. An HTML guide was included in the download with the compiler. In the DOCS directory of your compiler installation, open the file `pic_chipinfo.html` or `pic18_chipinfo.html`. Click the link to your target device, and the page will show you the settings and values that are appropriate for the `config` pragmas.

With the configuration pragmas included into your source code, you should now be able to build as you did in the previous section, but with no warnings or errors. The successful build is shown in Figure 1-16.

FIGURE 1-16: THE OUTPUT WINDOW CLEAN BUILD



ACCESSING DEVICE REGISTERS

The code that was compiled in the previous section still has no runtime function. It is time to set up the device to perform a task. This section explains how to access the device's special function registers (SFRs) and turn on an LED attached to a port.

The following code sets the data direction for port D, then writes a value to that port's latch[1].

```
#include <xc.h>

// your configuration bit settings go here
// configuration code (indicated earlier) omitted for brevity

int main(void)
{
    // code to access your port replaces the following
    TRISD = 0x0;    // set all port D bits to be output
    LATD = 0x55;    // write a value to the port D latch

    return 0;
}
```

In this code, special identifiers are used to represent the SFRs. These identifiers are associated with variables that are defined by the inclusion of `<xc.h>`. They can be used like any other C variable, but they are each assigned an address that maps them over the register they represent. Note that writing to these variables writes to the register, and consequently could change the state of the device. Simply the act of reading these variables could, on occasion, affect the device.

The identifiers mentioned above are the same as the names of the registers they represent, as specified by the device data sheet. However, that may not always be the case, particularly with the identifiers that represent bits within SFRs.

To learn which names to use when accessing SFRs on the device, follow these steps to find the register names used for any device.

Step 1 creates a source file that includes `<xc.h>`.

Step 2 verifies appropriateness of the device, builds the code, and checks for errors.

Step 3 looks into the preprocessed file that was produced by the compiler in step 2.

1. Some devices do not have a latch register associated with a port and you will need to write to the port directly, for example the `PORTD` register. Check your device data sheet.

Step 1 creates a source file that includes `<xc.h>`.

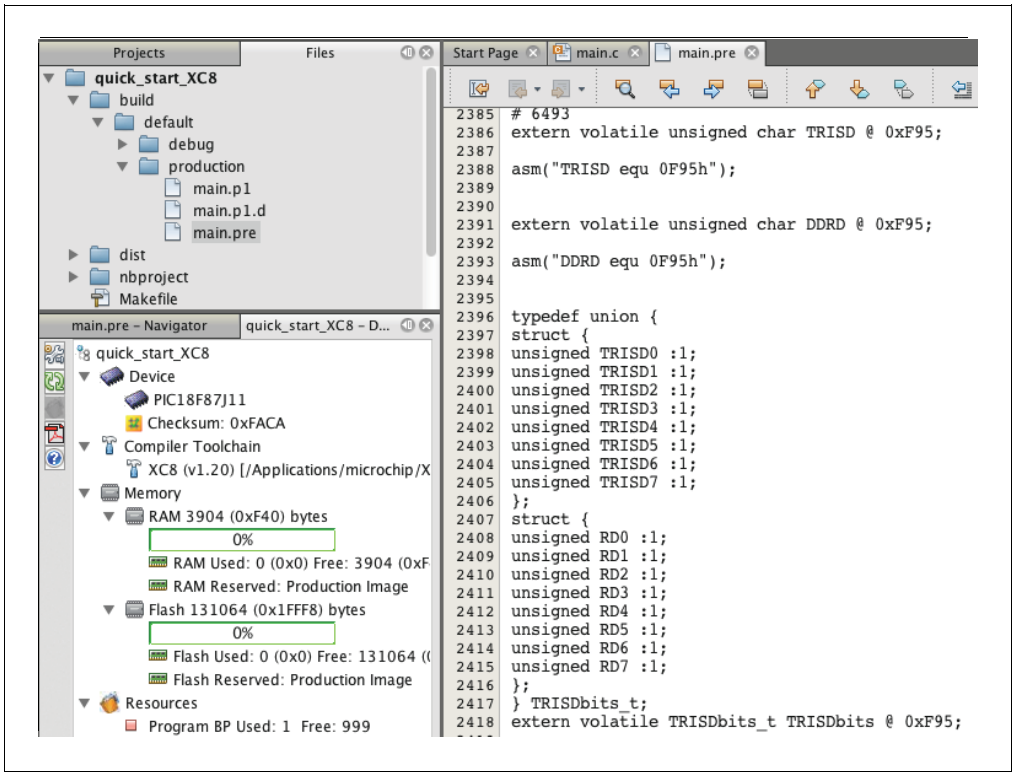
The content of the file is largely irrelevant. The skeleton code we started with in this guide (i.e., an empty `main()` function) is ideal. You can use the file from the previous section.

Step 2 verifies appropriateness of the device, builds the code, and checks for errors.

Step 3 looks into the preprocessed file that was produced by the compiler in step 2.

This file is usually left behind after compilation. If your source file is called `main.c`, the preprocessed file is called `main.pre`. If you are compiling on the command-line, the file will be left in the same directory as the source file. If you are using MPLAB X IDE, click on the Files window, shown in the upper left of Figure 1-17. Look under the project folder in this window, in the `build/default/production` folder[1].

FIGURE 1-17: THE PREPROCESSED FILE SHOWING REGISTER NAMES



The content of this preprocessed file for the PIC18F87J11 is also shown in Figure 1-17. It appears on the right, opened in the editor. It contains the C definitions for all the variables that represent the SFRs.

In Figure 1-17, you can see that the variable `TRISD` is defined as an `unsigned char` and is placed at the address 0xF95. If you check the data sheet for the PIC18F87J11, it will confirm that this is, indeed, the address of the TRISD register. Note also that there is an alias for this variable, `DDRD`. You can also see the bits defined inside this register, so for example, you can use the structure bit-field `TRISDbits.TRISD7`, or its alias, `TRISDbits.RD7`, to access the MSb of the port direction register. Use the addresses of registers to find the corresponding names in the file if you cannot find them otherwise. It is important to note that this preprocessed file is an intermediate file, any changes you make to it will be lost the next time you build.

1. The `production` directory is used to hold intermediate files for a release (production) build; the `debug` directory holds the same files for a debug build, see **"Compilation"**.

The source code presented at the beginning of this section writes a value to the port D latch. If you were to run this code and observe the content of that latch in the simulator or an emulator, you would see that the value 0x55 is stored in that register. However, this does not necessarily mean that the value in the port latch will appear on any device pins—in fact, for the Configuration bits specified earlier, and the PIC18F87J11 device, the port will *not* be mapped to the pins. In other words, if you measure the voltage on the pins corresponding to port D, or connect LEDs to these pins, you may not see the voltages or illuminated LEDs you are expecting. There is something else we must do to ensure that port D is connected to the device pins.

DISABLING PERIPHERALS THAT SHARE PORT PINS

This section describes the additional code that must be added to the sample project to ensure that the value stored in the port latch is presented on the device pins. Failure to include this code is a common reason why many simple getting-started programs do not work as expected.

The 8-bit Microchip PIC devices have a large number of peripherals on-board, but a limited number of pins. The IO lines from many peripherals might share the same pin. But, only one peripheral can use a pin at any given time.

The digital IO ports are treated the same as any other peripheral—unless a port is assigned use of a pin, it is not connected to the outside world. In *many* instances, by default, the ports are not connected to pins.

To learn which peripherals share a pin, consult the device data sheet. The pin diagrams provide quick reference, but many PIC device data sheets have a section that fully explains the usage associated with each pin.

Use the following steps to work out which peripherals may need to be initialized so that the port will be connected to the port pins.

Step 1 locates the Pinout I/O Description or similar table in the device data sheet.

Step 2 focuses on the first alternate peripheral listed in the table.

Step 3 determine what to write to the SFRs to disable the peripheral.

Step 4 repeats this process for any other peripheral listed in the table from step 1.

Step 1 locates the Pinout I/O Description or similar table in the device data sheet.

An extract from this table for the PIC18F87J11 device (an 80-pin package) is shown in Figure 1-18.

FIGURE 1-18: PINOUT TABLE EXTRACT FOR THE PIC18F87J11

Pin Name	Pin Number 80-TQFP	Pin Type	Buffer Type	Description
				PORTD is a bidirectional I/O port.
RD0/AD0/PMD0	72			
RD0		I/O	ST	Digital I/O.
AD0		I/O	TTL	External Memory Address/Data 0.
PMD0(6)		I/O	TTL	Parallel Master Port data.
RD1/AD1/PMD1	69			
RD1		I/O	ST	Digital I/O.
AD1		I/O	TTL	External Memory Address/Data 1.
PMD1(6)		I/O	TTL	Parallel Master Port data.
RD2/AD2/PMD2	68			
RD2		I/O	ST	Digital I/O.
AD2		I/O	TTL	External Memory Address/Data 2.
PMD2(6)		I/O	TTL	Parallel Master Port data.

For the PICDEM PIC18 Explorer Board used in this guide, LEDs are connected to port D. (Check the user's guide for your development board connections, if required.) The pins used by port D are labeled RD0, RD1, RD2, etc. You will notice from the figure that these pin labels also include other references, for example RD0/AD0/PMD0. The table indicates that port D, the external memory bus and the parallel master port all share the same pins. The other pins used by port D (not shown in the Figure 1-18) are also shared with the SPI peripherals.

Step 2 focuses on the first alternate peripheral listed in the table.

Find the section in the device data sheet that is relevant to that peripheral and look for the SFRs that control the peripheral.

Step 3 determine what to write to the SFRs to disable the peripheral.

For example, the section relating to the external memory bus indicates the EBDIS bit in the MEMCON SFR controls the external memory module. As indicated in the device data sheet extract shown in Figure 1-19, the value at POR (Power-on Reset) for EBDIS is 0, which implies the bus is active. This module must be disabled by setting the EBDIS bit.

FIGURE 1-19: EXTERNAL BUS REGISTER DESCRIPTION EXTRACT

REGISTER 8-1: MEMCON: EXTERNAL MEMORY BUS CONTROL REGISTER

R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0
EBDIS	—	WAIT1	WAIT0	—	—	WM1	WM0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7 **EBDIS**: External Bus Disable bit
1 = External bus is enabled when the microcontroller accesses external memory; otherwise, all external bus drivers are mapped as I/O ports
0 = External bus is always enabled, I/O ports are disabled

Following a similar process for the parallel master port, we see that this port is controlled by the PMPEN bit. It has a POR value of 0, which implies this peripheral is already disabled after POR. Thus, no additional code is required in our program for this peripheral.[1]

Step 4 repeats this process for any other peripheral listed in the table from step 1.

Through repetition of these steps, the test program for the PIC18F87J11 can now be expanded as follows[2].

```
#include <xc.h>

// your configuration bit settings go here
// configuration code (indicated earlier) omitted for brevity

int main(void)
{
    // intialization code for your device replaces the following
    WDTCONbits.ADSHR = 1;   // enable alternate access to MEMCON
    MEMCONbits.EBDIS = 1;   // turn off external memory bus

    // code to access your port replaces the following
    TRISD = 0x0;    // set all port D bits to be output
    LATD = 0x55;    // write a value to the port latch

    return 0;
}
```

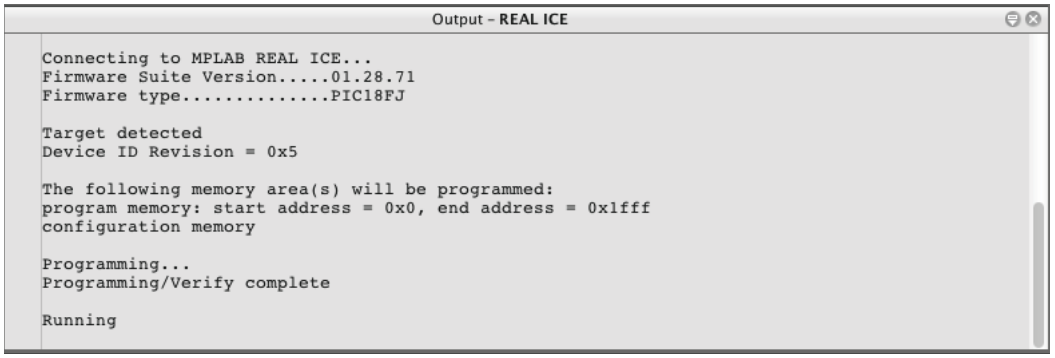
1. There is no harm in explicitly disabling this peripheral. If your program performs a soft Reset, then the value of this bit may no longer be 0.
2. The ADSHR bit must be set to access the MEMCON register on this device.

DOWNLOADING AND RUNNING YOUR CODE

The source code listed at the end of the previous section works for the PIC18F87J11 device. Confirm that the code you have written for your device works as expected. In this section, you can build, download and run the binary image.

Click the Build and Run project button, as shown in Figure 1-12. Or, click Build and Debug project if you want to explore your hardware debugger's features. Either of these buttons will ensure that the compiled binary image of your code is up-to-date, and will download and execute your code. If you click the Build and Debug project button, it will start the debug executive in your device, as well. You can watch your code being built in the Output window, similar to what is shown in Figure 1-20.

FIGURE 1-20: EXECUTION OF THE CODE



With your code running, you should see the LEDs illuminate as you have specified in your code, or be able to measure the voltage on the pins assigned to the port you are using. A value of 0x55 assigned to the port will illuminate every second LED connected to the port's pins.

However, if you are using the simulator, you should stop execution of your code and confirm the value contained in the port. And, that will only prove that you are correctly writing to the port, not that the port would be connected to the pins on an actual device.

Consult the MPLAB X IDE documentation if you wish to explore stepping, breakpoints or other debug features.

IMPLEMENTING A MAIN LOOP

As it stands, the simple test program we ran in the previous section executes a couple of statements then terminates. After execution reaches the end of `main()`, code added by the compiler jumps back to the Reset vector. The device then executes the runtime startup code and the `main()` function again. These soft Resets are not desirable. In this section, other features of the compiler are introduced. You will learn how to change the program so that it writes different values to the LEDs, and how to adjust `main()` so that it never terminates.

In the following code, we prevent `main()` from terminating by adding an infinite loop. Inside this loop we assign the value of a counter (`portValue`) to the port latch and increment this counter so that the port value changes with time. A delay routine is also added so that the individual states of the LEDs can be seen.

```
#include <xc.h>

// your configuration bit settings go here
// configuration code (indicated earlier) omitted for brevity

unsigned char portValue;

int main(void)
{
    // intialization code for your device replaces the following
    WDTCONbits.ADSHR = 1;   // enable alternate access to MEMCON
    MEMCONbits.EBDIS = 1;   // turn off external memory bus

    // code to access your port replaces the following
    TRISD = 0x0;    // set all port D bits to be output

    while(1) {
        LATD = portValue++;
        _delay(40000);
    }

    return 0;   // we should never reach this
}
```

Build and run this code. If you are using hardware, ensure the LEDs connected to the port count up the binary values from 0 to 0xFF.

Note that the port itself was not incremented. Using a port register in such expressions may trigger read-modify-write issues. Always use a variable to hold the value you want the port to assume. Modify this variable as required, then assign the value of this variable to the port or port latch.

The delay routine used here (note the leading underscore character) is actually a compiler built-in function, but you can find help for this and compiler library functions in the appendixes of the *MPLAB XC8 C Compiler User's Guide*. Without the delay, the LEDs would flash so fast that they would appear to be just dimly lit. You may need to adjust the delay length to suit your device's clock frequency.

USING INTERRUPTS

In this section, the code presented in the previous section is converted to use interrupts. This can be done entirely in the C language. For 8-bit devices, the compiler produces code that switches context and this is automatically linked to the interrupt vector.

Below is code that is functionally identical to the code we saw in the previous section. Instead of using a delay, it uses timer 0 to generate an interrupt. The code associated with the interrupt increments the counter variable. The `while()` loop in `main()` writes the counter value to the port (LEDs), as it did previously.

```
#include <xc.h>

// your configuration bit settings go here
// configuration code (indicated earlier) omitted for brevity

unsigned char portValue;    // our counter variable

void interrupt myIsr(void)
{
    // only process timer-triggered interrupts
    if(INTCONbits.TMR0IE && INTCONbits.TMR0IF) {
        portValue++;
        INTCONbits.TMR0IF = 0;  // clear this interrupt condition
    }
}

int main(void)
{
    WDTCONbits.ADSHR = 1;   // enable alternate access to MEMCON
    MEMCONbits.EBDIS = 1;   // turn off external memory bus

    TRISD = 0x0;

    T0CON = 0b10001000;     // enable the timer as 16 bit...
                            // internal clock, no prescaler
    INTCONbits.TMR0IE = 1;  // enable interrupts for timer 0
    ei();                   // enable all interrupts

    while(1) {
        LATD = portValue;
    }

    return 0;
}
```

After building and running this code, you should see the LEDs toggle as they did in the previous section. To adjust the LED's rate of change, you can, for example, enable the timer's prescaler to slow its clock.

Notice that the `interrupt` specifier was used to turn the function `myIsr()` into an interrupt function. As this one interrupt function may have to deal with multiple interrupt sources, code was added to ensure that the counter is only incremented if it was the timer that generated the interrupt. It is good practice to place as little code as possible inside the interrupt function.

In `main()`, note that a binary constant (using the `0b` prefix) is used so that the bits within T0CON can be seen easily. Remember that if you are using a different device, you need to consult the data sheet for the device to find the register names and bits within those registers that must be set for correct timer operation.

The compiler macro, `ei()`, was used to enable the interrupts, but you can explicitly set the GIE bit in the INTCON register, if you prefer.

CONCLUSION

Using the basic concepts and methods presented in this guide, you will be able to write quite advanced programs for a 8-bit PIC device. You will be able to configure the device correctly, determine the names for all the SFRs and the SFR bits used by your device. Plus, you can make the device peripherals trigger interrupts and have your code respond to these events.

It is important to be familiar with the C89 ANSI Standard C language implemented by this compiler. The *MPLAB XC8 C Compiler User's Guide* (DS50002053) has more detailed information on the operation of the compiler and non-standard syntax. This document can be found in the DOCS directory of the compiler's installation directory. It can also be accessed by clicking the Compiler Help button (the blue "?" button at the bottom of the vertical row of buttons in the dashboard, as shown in Figure 1-11) in the project Dashboard.

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

Printed on recycled paper.

ISBN: 978-1-62077-271-3

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═

Microchip received ISO/TS-16949:2009 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario, Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

Japan - Tokyo
Tel: 81-3-6880- 3770
Fax: 81-3-6880-3771

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7828
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

11/29/12