PICKIT™ 2 PIC18F46K20 Starter Kit C18 LESSONS

1 Introduction

The following series of lessons covers the basics of developing applications for the Microchip PIC18 series of microcontrollers. Working with the MPLAB IDE, MPLAB C18 compiler, and the PICkit 2 Development Programmer/Debugger is introduced in a series of lessons that cover fundamental microcontroller operations, from simply turning on an LED to creating interrupt service routines.

All lessons can be completed with the freely available MPLAB C18 Student Edition compiler in the freely available Microchip MPLAB Integrated Development Environment. The lesson files may be installed from the included CDROM.

Please note that these lessons are not intended to teach the C programming language itself, and prior familiarity with the C language is a prerequisite for these lessons.

PIC18F46K20 Starter Kit C18 Lessons

- Lesson 1: Hello LED (Turn on LED)
- Lesson 2: Blink LED
- Lesson 3: Rotate LED (Turn on in sequence)
- Lesson 4: Switch Input
- Lesson 5: Using Timer0
- Lesson 6: Using PICkit 2 Debug Express
- Lesson 7: Analog-to-Digital Converter (ADC)
- Lesson 8: Interrupts
- Lesson 9: Internal Oscillator
- Lesson 10: Using Internal EEPROM
- Lesson 11: Program Memory Operations
- Lesson 12: Using the CPP Module PWM

Appendix A contains the PIC18F46K20 Starter Kit Demo Board Schematic diagram.

1.1 Before Beginning the Lessons

Please ensure the following files and software has been installed on your PC before beginning:

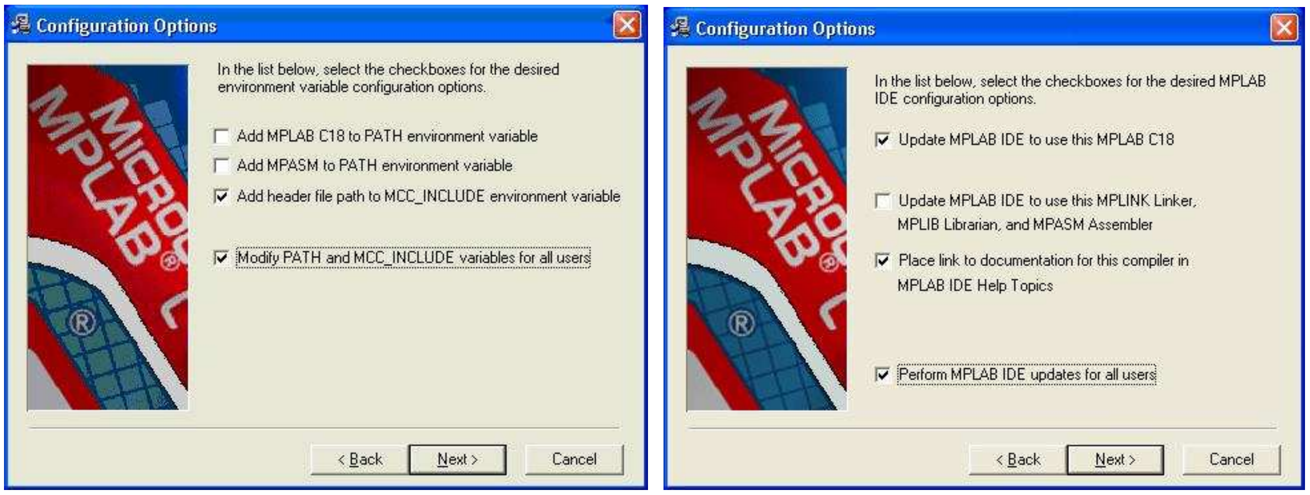
1. MPLAB IDE version 8.01 or later.

2. MPLAB C18 compiler v3.13 or later. The Student Edition may be used.

   When Installing MPLAB C18, please be sure to select the following options, as shown in Figure 1-1.
   Add header file path to MCC_INCLUDE environment variable
   Update MPLAB IDE to use this MPLAB C18
   Place Link to documentation for this compiler in MPLAB IDE Help Topics

3. The PIC18F46K20 Starter Kit Demo Board C18 Lessons files.

FIGURE 1-1: MPLAB C18 INSTALLATION CONFIGURATION OPTIONS



In the list below, select the checkboxes for the desired environment variable configuration options.

- ☐ Add MPLAB C18 to PATH environment variable
- ☐ Add MPASM to PATH environment variable
- ☑ Add header file path to MCC_INCLUDE environment variable
- ☑ Modify PATH and MCC_INCLUDE variables for all users

In the list below, select the checkboxes for the desired MPLAB IDE configuration options.

- ☑ Update MPLAB IDE to use this MPLAB C18
- ☐ Update MPLAB IDE to use this MPLINK Linker, MPLIB Librarian, and MPASM Assembler
- ☑ Place link to documentation for this compiler in MPLAB IDE Help Topics
- ☑ Perform MPLAB IDE updates for all users

2 PIC18FXXXX Microcontroller Architectural Overview

This section provides a simple overview of the PIC18FXXXX microcontroller architecture.

2.1 Memory Organization

The PIC18FXXXX microcontrollers are “Harvard Architecture” microprocessors, meaning that program memory and data memory are in separate spaces. This allows faster execution as the program and data busses are separate and dedicated, so one bus does not have to be used for both memory types. The return address stack also has its own dedicated memory.

2.1.1 Program Memory

The program memory space is addressed by a 12-bit Program Counter, allowing a 2 Mb program memory space. Typically, PIC18FXXXX microcontrollers have on-chip program memory in the range of 4K to 128K bytes. Some devices allow external memory expansion.

At Reset, the Program Counter is set to zero and the first instruction is fetched. Interrupt vectors are at locations 0x000008 and 0x000018, so a `GOTO` instruction is usually placed at address zero to jump over the interrupt vectors.

Most instructions are 16 bits, but some are double word 32-bit instructions. Instructions cannot be executed on odd numbered bytes.

These are some important characteristics of the PIC18C architecture and MPLAB C18 capabilities with reference to program memory:

MPLAB C18 Implementation
Refer to the *MPLAB C18 C Compiler User’s Guide* for more information on these features.

- Instructions are typically stored in program memory with the section attribute `code`.
- Data can be stored in program memory with the section attribute `romdata` in conjunction with the `rom` keyword.
- MPLAB C18 can be configured to generate code for two memory models, small and large. When using the small memory model, pointers to program memory use16 bits. The large model uses 24-bit pointers.

PIC18 Architecture

- In some PIC18XXXX devices, program memory or portions of program memory can be code-protected. Code will execute properly but it cannot be read out or copied.
- Program memory can be read using table read instructions, and can be written through a special code sequence using the table write instruction.

2.1.2 Data Memory

Data memory is called “file register” memory in the PIC18XXXX family. It consists of up to 4096 bytes of 8-bit RAM. Upon power-up, the values in data memory are random. Data is organized in banks of 256 bytes, requiring that a bank (the upper 4 bits of the register address) be selected with the Bank Select Register (BSR). Special areas in Bank 0 and in Bank 15 can be accessed directly without concern for banking. These special data areas are called Access RAM. The high Access RAM area is where most of the Special Function Registers are located.

When using MPLAB C18, this banking is usually transparent, but the use of the `#pragma varlocate` directive tells the compiler where variables are stored, resulting in more efficient code.

Uninitialized data memory variables, arrays and structures are usually stored in memory with the section attribute, `udata`. Initialized data can be defined in MPLAB C18 so that variables will have correct values when the compiler initialization executes. This means that the values are stored in program memory, then moved to data memory on start-up. Depending upon how much initialized memory is required for the application, the use of initialized data (rather than simply setting the data values at run time) may adversely affect the efficient use of program memory. Since file registers are 8 bits, when using variables consideration should be made on what is the best datatype to define them as. For example, when a variable value is not expected to exceed 255, defining it as a `char` instead of an `int` will result in smaller, faster code.

2.1.3 Special Function Registers

Special Function Registers (SFRs) are CPU core registers (such as the Stack Pointer, STATUS register and Program Counter) and include the registers for the peripheral modules on the microprocessor. The peripherals include such things as input and output pins, timers, USARTs and registers to read and write the EEDATA areas of the device. MPLAB C18 can access these registers by name, and they can be read and written like a variable defined in the application. Use caution, though, because some of the Special Function Registers have characteristics different from variables. Some have only certain bits available, some are read-only and some may affect other registers or device operation when accessed. These registers are mapped to addresses in Bank 15 of the data memory.

2.1.4 Return Address Stack

`CALL` and `RETURN` instructions push and pop the Program Counter on the return address stack. The return stack is a separate area of memory, allowing 31 levels of subroutines.

The `CALL/RETURN` stack is distinct from the software stack maintained by MPLAB C18. The software stack is used for automatic parameters and local variables and resides in file register memory as defined in the linker script.

3 PIC18F46K20 Starter Kit Demo Board Lessons

Connect the PICkit 2 Programmer/Debugger to a PC USB port, and connect the Demo Board to the PICkit via header P1 labeled ICSP.

3.1 Lesson 1: Hello LED

This first lesson shows how to create a C18 project in the MPLAB IDE and turn on a demo board LED using the PIC18F46K20.

Key Concepts

- Use the MPLAB IDE Project Wizard to create a new project for a microcontroller.
- The TRISx Special Function Registers (SFRs) are used to set microcontroller port I/O pin directions as inputs or outputs.
- The LATx SFRs are used to set microcontroller port Output pins to a high or low state.

3.1.1 Creating the Lesson 1 Project in the MPLAB IDE

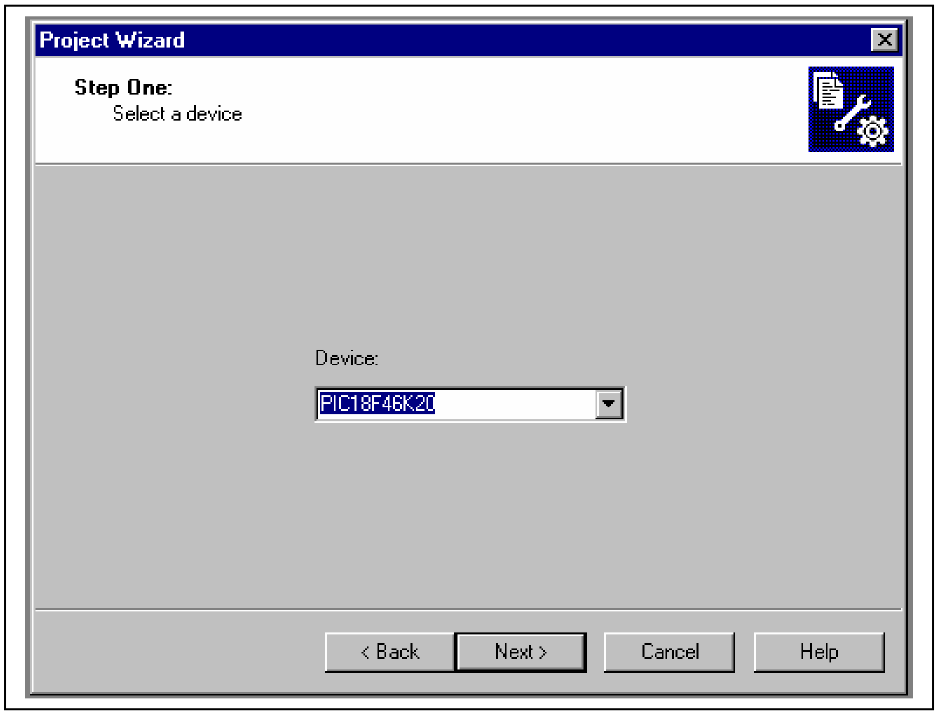
Begin by opening the MPLAB IDE from the desktop shortcut icon:

MPLAB IDE
v8.01

To create project, use the MPLAB IDE Project Wizard by selecting the menu *Project > Project Wizard…*. The Project Wizard "Welcome!" dialog is shown. Click **Next** to continue.

Step One: Select a device: In the Project Wizard dialog, select the <PIC18F46K20> as the target device in the dropdown box as shown in Figure 3-2 and click **Next** to continue.

FIGURE 3-1: WIZARD STEP ONE: SELECT PIC18F46K20 DEVICE



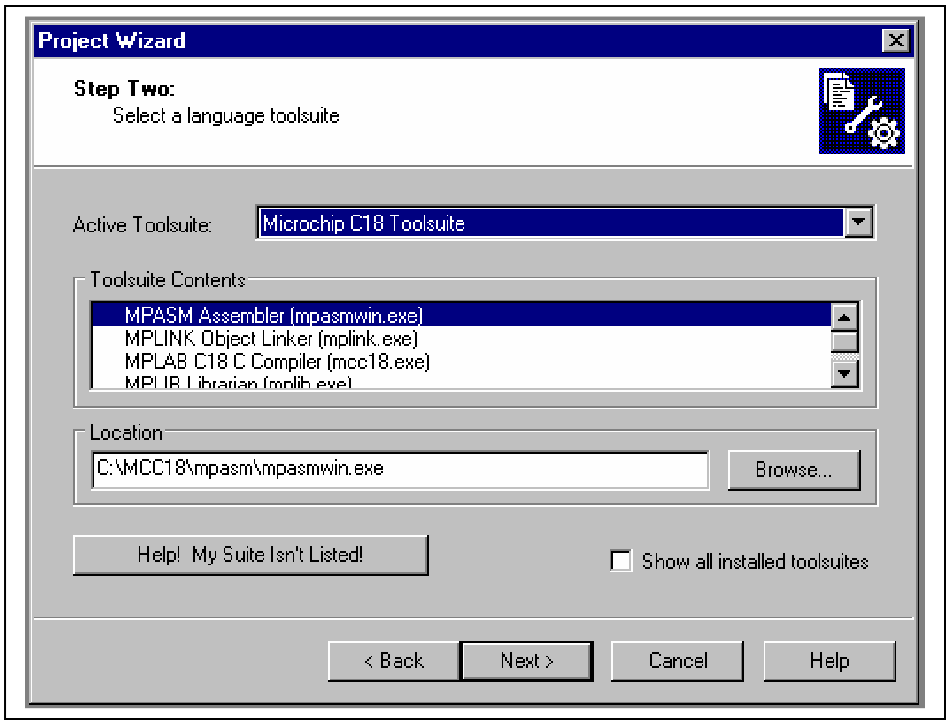
Project Wizard

Step One:
Select a device

Device: PIC18F46K20

Step Two: Select a language toolsuite: This PIC18F microcontroller project will be in C, so select the <Microchip C18 Toolsuite> from the "Active Toolsuite:" dropdown box, as shown in Figure 3-2. Click **Next** to continue.

FIGURE 3-2: WIZARD STEP TWO: SELECT TOOLSUITE



Project Wizard

Step Two:
Select a language toolsuite

Active Toolsuite: Microchip C18 Toolsuite

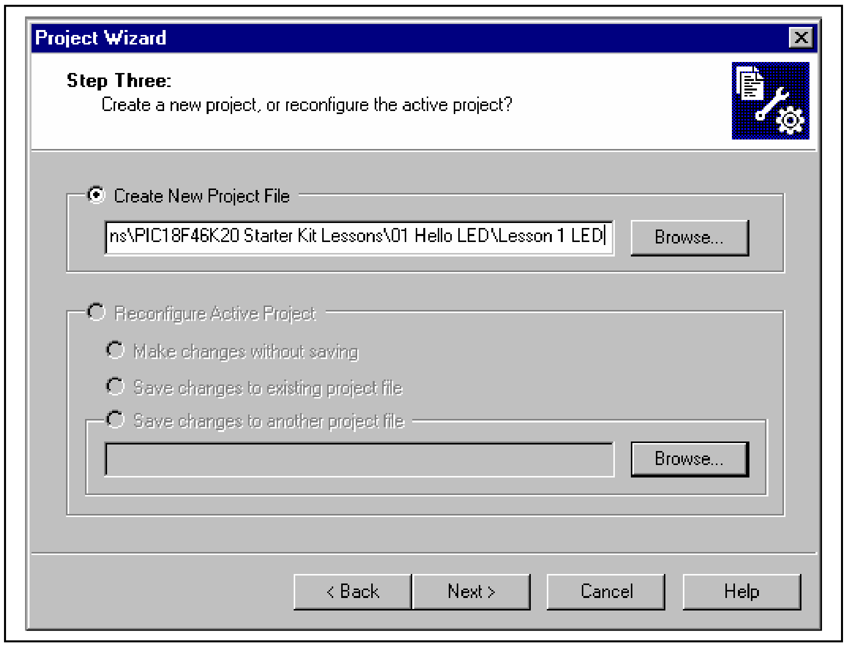
Toolsuite Contents
MPASM Assembler (mpasmwin.exe)
MPLINK Object Linker (mplink.exe)
MPLAB C18 C Compiler (mcc18.exe)
MPLIB Librarian (mplib.exe)

Location
C:\MCC18\mpasm\mpasmwin.exe

☐ Show all installed toolsuites

Step Three: Create a new project: Create the project file in the existing directory for lesson 1. **Browse** to the directory folder `C:\Lessons\PIC18F46K20 Starter Kit Lessons\01 Hello LED` and name the project `Lesson 1 LED`. **Save** the project and then click **Next** to continue.

FIGURE 3-3: WIZARD STEP THREE: CREATE A NEW PROJECT



Project Wizard

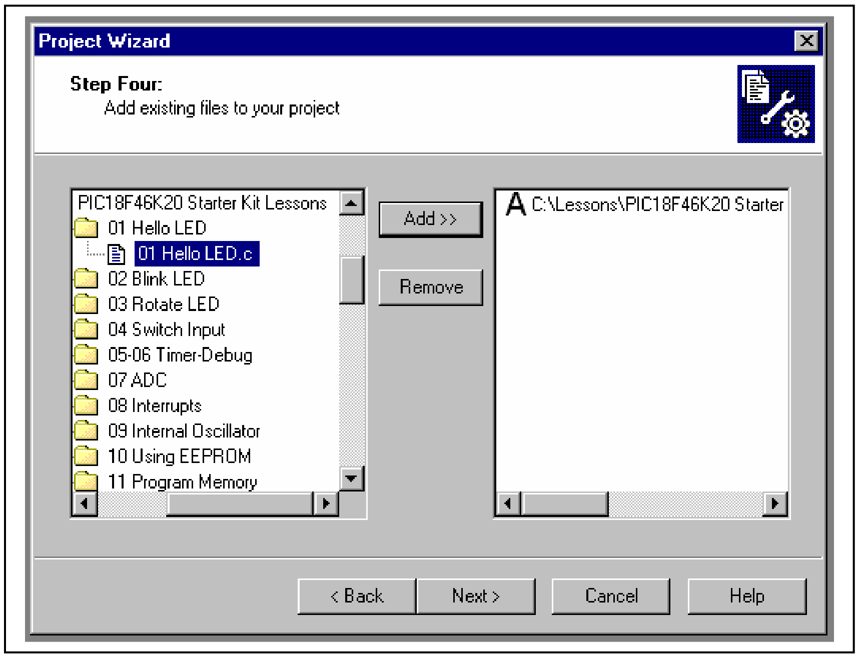
Step Three:
Create a new project, or reconfigure the active project?

(●) Create New Project File
ns\PIC18F46K20 Starter Kit Lessons\01 Hello LED\Lesson 1 LED

( ) Reconfigure Active Project
( ) Make changes without saving
( ) Save changes to existing project file
( ) Save changes to another project file

Step Four: Add existing files to your project: This dialog allows any existing source or other files to be added to the project. Note it is also possible to add new files to project after it has been created. In the left pane, select the `01 Hello LED.c` file in the project directory from Step Three and click **Add>>**. The file will now show up the right pane of the dialog as show in Figure 3-4. Click **Next** to continue.

FIGURE 3-4: WIZARD STEP FOUR: ADD EXISTING FILES



Project Wizard

Step Four:
Add existing files to your project

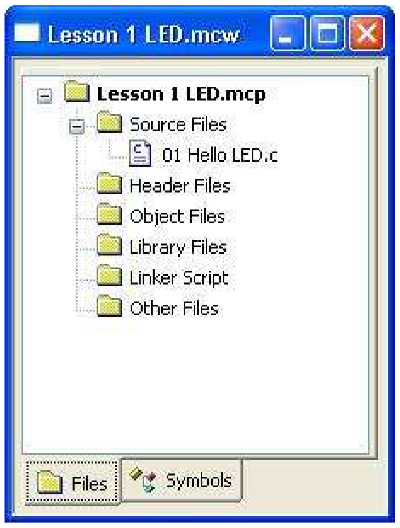
PIC18F46K20 Starter Kit Lessons
01 Hello LED
01 Hello LED.c
02 Blink LED
03 Rotate LED
04 Switch Input
05-06 Timer-Debug
07 ADC
08 Interrupts
09 Internal Oscillator
10 Using EEPROM
11 Program Memory

A C:\Lessons\PIC18F46K20 Starter

Summary: In the final wizard dialog, verify the Project Parameters and click **Finish.** To view the Project Window in the MPLAB IDE, select menu View > Project.

The Project Window (see Figure 3-5) shows the workspace file name (`Lesson 1 LED.mcw`) in the title bar, and the project file (`Lesson 1 LED.mcp`) at the top of the file tree view. A ***workspace*** file keeps track of what files and windows are open, where the windows are located in the MPLAB IDE workspace, what programmer or debugger tools are selected and how they are configured, and other information on how the MPLAB IDE environment is set up. A ***project*** file keeps track of all the necessary files to build a project , including source and header files, library files, linker scripts, and other files. As shown in Figure 3-5, the Lesson 1 LED project currently only contains one source file, `01 Hello LED.c`, which was added in the Project Wizard.

FIGURE 3-5: PROJECT WINDOW



To complete the project setup, we will add a linker script and microcontroller header file to the project. A linker script is required to build the project. It is a command file for the linker, and defines options that describe the available memories on the target microcontroller. There are four example linker files for the microcontroller:

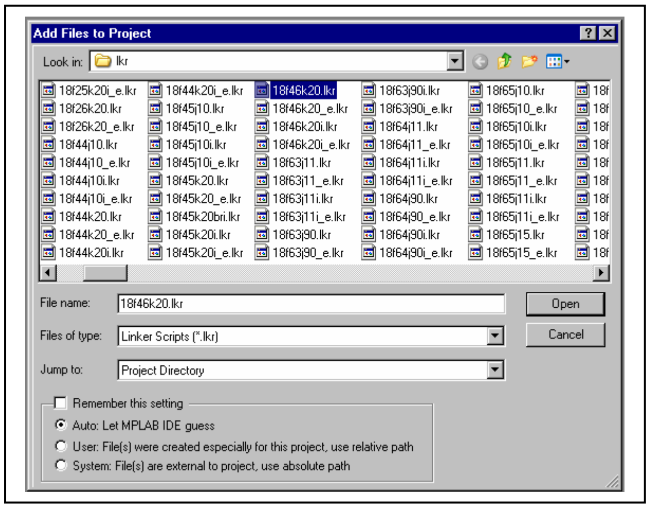
`18f46k20.lkr`	Basic linker script file for compiling a memory image in non-extended processor mode. (More on the extended mode in a later lesson.)
`18f46k20_e.lkr`	Linker script file for compiling using extended mode.
`18f46k20i.lkr`	Linker script file for use when debugging. These linker scripts prevent application code from the using the small areas of memory reserved for the debugger.
`18f46k20i_e.lkr`	Linker script file for debugging in extended mode.

Add the linker script by selecting menu Project > Add files to project…. In the "Files of type" dropdown box, select "Linker Scripts (*.lkr)" as shown in Figure 3-6. Browse to the linker scripts directory `C:\MCC18\lkr` and open the `18f46k20i.lkr` file as the debugger will be used in later lessons.

Files can also be added by right-clicking in the Project Window. Right-click on the "Header Files" folder and select Add Files… from the pop-up menu. Browse to the MPLAB C18 header file directory `C:\MCC18\h` and open the p18f46k20.h header file. The project window now looks like Figure 3-7.

It is important to note that the file selected in the directory it resides in will be added to be project, so modifying it will modify the original file. If this is not desired, open the file and use File > Save As… to save a new copy in the current project directory and then add the new file to the project.

FIGURE 3-6: ADD FILES TO PROJECT



Look in: lkr

```
18f25k20i_e.lkr  18f44k20i_e.lkr  18f46k20.lkr     18f63j90i.lkr    18f65j10.lkr     18f
18f26k20.lkr     18f45j10.lkr     18f46k20_e.lkr   18f63j90i_e.lkr  18f65j10_e.lkr   18f
18f26k20_e.lkr   18f45j10_e.lkr   18f46k20i.lkr    18f64j11.lkr     18f65j10i.lkr    18f
18f44j10.lkr     18f45j10i.lkr    18f46k20i_e.lkr  18f64j11_e.lkr   18f65j10i_e.lkr  18f
18f44j10_e.lkr   18f45j10i_e.lkr  18f63j11.lkr     18f64j11i.lkr    18f65j11.lkr     18f
18f44j10i.lkr    18f45k20.lkr     18f63j11_e.lkr   18f64j11i_e.lkr  18f65j11_e.lkr   18f
18f44j10i_e.lkr  18f45k20_e.lkr   18f63j11i.lkr    18f64j90.lkr     18f65j11i.lkr    18f
18f44k20.lkr     18f45k20bri.lkr  18f63j11i_e.lkr  18f64j90_e.lkr   18f65j11i_e.lkr  18f
18f44k20_e.lkr   18f45k20i.lkr    18f63j90.lkr     18f64j90i.lkr    18f65j15.lkr     18f
18f44k20i.lkr    18f45k20i_e.lkr  18f63j90_e.lkr   18f64j90i_e.lkr  18f65j15_e.lkr   18f
```

File name: 18f46k20.lkr

Files of type: Linker Scripts (*.lkr)

Jump to: Project Directory

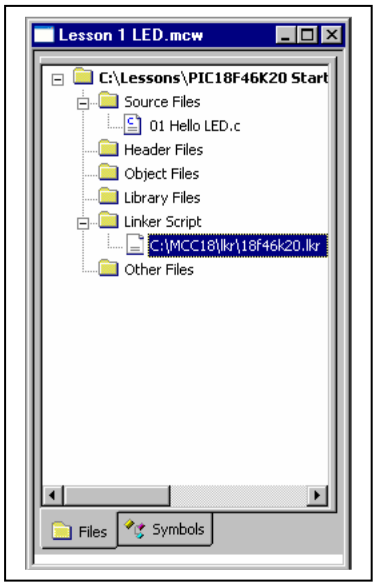
☐ Remember this setting

(●) Auto: Let MPLAB IDE guess

( ) User: File(s) were created especially for this project, use relative path

( ) System: File(s) are external to project, use absolute path

FIGURE 3-7: NEW PROJECT FILES



```
C:\Lessons\PIC18F46K20 Start
  Source Files
    01 Hello LED.c
  Header Files
  Object Files
  Library Files
  Linker Script
    C:\MCC18\lkr\18f46k20.lkr
  Other Files
```

Select *Project > Save Project* to save the new project configuration.

3.1.2 Exploring the Lesson 1 Source Code

Double-click the `01 Hello LED.c` source file name to open the lesson source code file in an MPLAB IDE editor window.

FIGURE 3-8: LESSON 1 "HELLO LED" SOURCE CODE

```
/** C O N F I G U R A T I O N   B I T S ******************************/

#pragma config FOSC = INTIO67
#pragma config WDTEN = OFF, LVP = OFF


/** I N C L U D E S **************************************************/
#include "p18f46K20.h"


/** D E C L A R A T I O N S *******************************************/


void main (void)
{

       TRISD = 0b01111111;        // PORTD bit 7 to output (0); bits 6:0 are inputs (1)

       LATDbits.LATD7 = 1;        // Set LAT register bit 7 to turn on LED

       while (1)
       ;

}
```

When this code is built, programmed into the PIC18F46K20 microcontroller, and executed it will turn on the LED connected to I/O pin RD7 by driving the pin high. Let’s discuss the elements of the code that makes this happen:

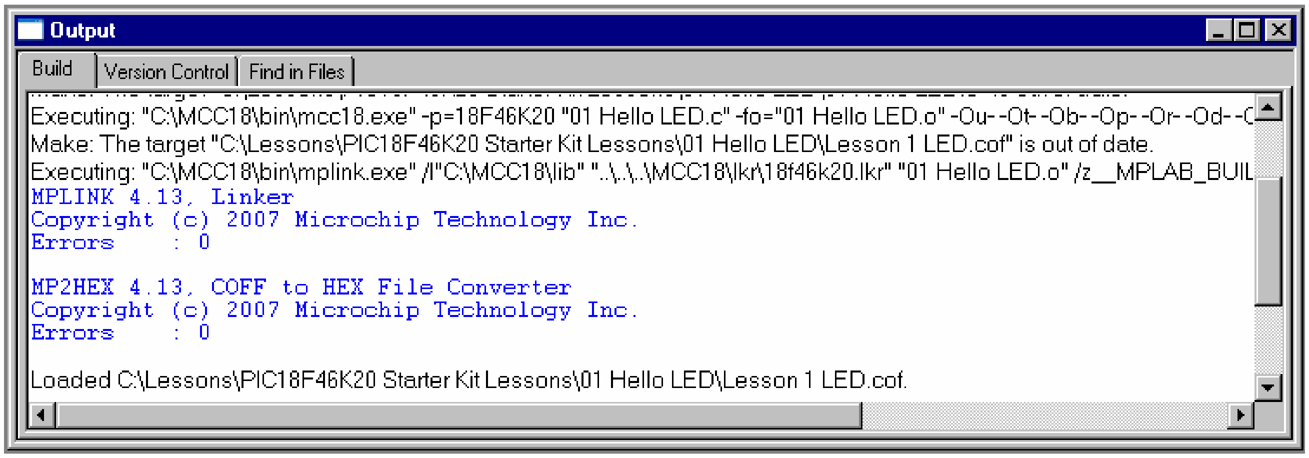
`#pragma config`	Pragma is a directive that has meaning for a specific compiler. It is used in MPLAB C18 with attributes to convey implementation-dependent information to the compiler. Here it is used with the `config` directive, which defines the states of the PIC18FXXXX Configuration bits. This will be discussed in more detail in Lesson 2.
`#include`	The `“p18f46k20.h”` file is included as this device-specific header file contains definitions for the variables used to access the Special Function Registers (SFRs) of the microcontroller. Some useful macros such as Nop() and ClrWdt() are also defined in this header.
`TRISD`	This variable is used to access the SFR of the same name, and is defined in the included microcontroller header file `p18f46k20.h`. The TRIS (tri-state) registers are used to set the directions of the pins in the associated I/O port, in this case pins RD0 to RD7. A TRISD bit value of `‘0’` sets the pin to an output. A value of `‘1’` sets a pin to be an input. With the binary value of `0b01111111` we set RD7 to an output and RD6-RD0 to inputs.
`LATDbits.LATD7`	The `LATDbits` struct is also defined in `p18f46k20.h`, and gives access to the individual bits in the LATD SFR. (There is also a `TRISDbits` struct, for accessing bits of TRISD, and a `LATD` variable defined to access the entire byte-wide register.) The LATD (latch) register is used to set the output state of the RD7-RD0 pins. A bit value of `‘1’` sets an output pin to a high state. Bits for pins defined in the TRIS register as inputs do not have an effect. Setting `LATDbits.LATD7 = 1` will output a high level on RD7, turning on LED 7 on the demo board.
`while(1)`	In this case of code running on an embedded microcontroller, there is no operating system to return to when the code finished executing. Therefore an infinite C `while` loop is used to keep the microcontroller running and prevent it from exiting `main()` and trying to execute undefined memory locations.

3.1.3 Building and Programming the Lesson 1 Code

Build the lesson code in an executable memory image by selecting ***Project > Build All*** in the MPLAB IDE. The memory image is stored in a `.hex` file in the project directory.

The results of the build will be shown in the Output Window in the MPLAB IDE workspace under the “Build” tab. The calls to the MCC18 compiler and Linker are shown, along with any errors that may occur. If the build is successful, the Output Window will show BUILD SUCCEEDED as in Figure 3-9.

FIGURE 3-9: MPLAB IDE OUTPUT WINDOW BUILD RESULTS



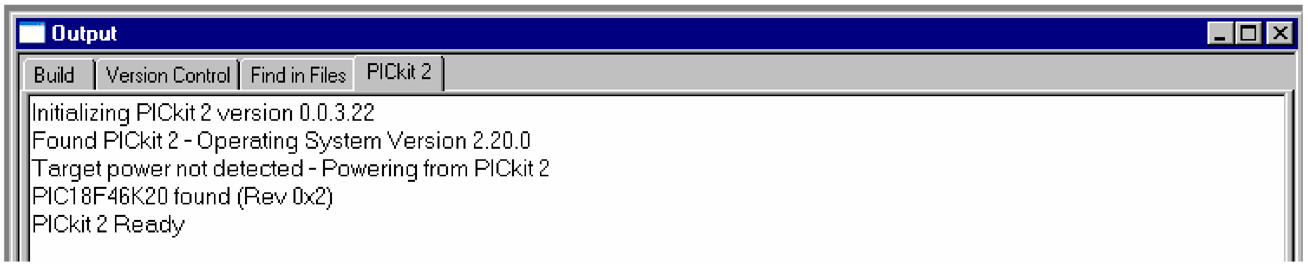
```
Executing: "C:\MCC18\bin\mcc18.exe" -p=18F46K20 "01 Hello LED.c" -fo="01 Hello LED.o" -Ou- -Ot- -Ob- -Op- -Or- -Od- -C
Make: The target "C:\Lessons\PIC18F46K20 Starter Kit Lessons\01 Hello LED\Lesson 1 LED.cof" is out of date.
Executing: "C:\MCC18\bin\mplink.exe" /l"C:\MCC18\lib" "..\..\MCC18\lkr\18f46k20.lkr" "01 Hello LED.o" /z__MPLAB_BUIL
MPLINK 4.13, Linker
Copyright (c) 2007 Microchip Technology Inc.
Errors    : 0

MP2HEX 4.13, COFF to HEX File Converter
Copyright (c) 2007 Microchip Technology Inc.
Errors    : 0

Loaded C:\Lessons\PIC18F46K20 Starter Kit Lessons\01 Hello LED\Lesson 1 LED.cof.
```

Note: If an error that the include file “`p18f46k20.h`” cannot be found is generated, this usually means that MPLAB C18 was installed without checking the *Add header file path to MCC_INCLUDE environment variable* option during setup. It is recommended to re-install MPLAB C18 with this option checked.

To program the code into the PIC18F46K20 microcontroller, the PICkit 2 Programmer/Debugger is used. Select the PICkit 2 as a programmer in the MPLAB IDE with *Programmer > Select Programmer > 4 PICkit 2*.

This will create a new tab in the Output Window for the PICkit 2 programmer, where messages from the programmer are displayed. The PICkit 2 will be initialized and should report finding the PIC18F46K20 microcontroller on the demo board as shown in Figure 3-10.

FIGURE 3-10: OUTPUT WINDOW PICKIT 2 PROGRAMMER



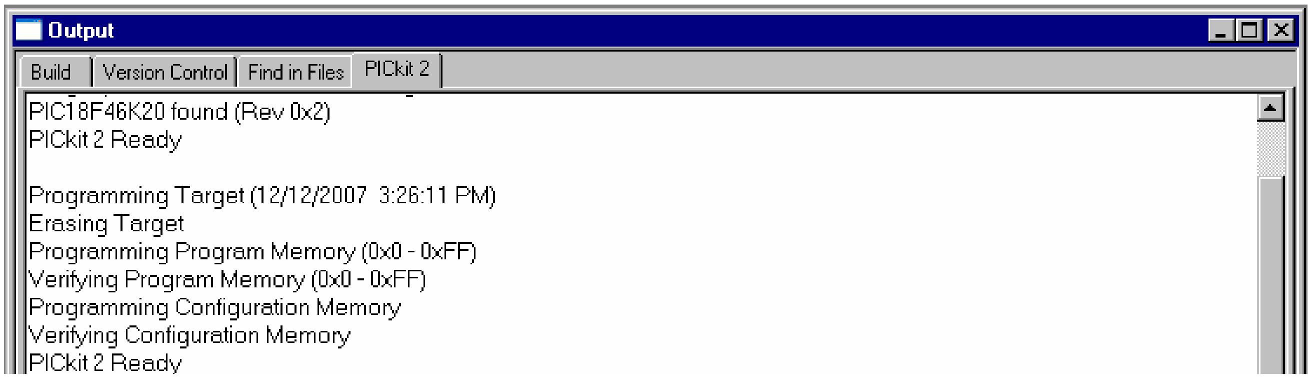
```
Initializing PICkit 2 version 0.0.3.22
Found PICkit 2 - Operating System Version 2.20.0
Target power not detected - Powering from PICkit 2
PIC18F46K20 found (Rev 0x2)
PICkit 2 Ready
```

Program the built code into the PIC microcontroller by selecting menu *Programmer > Program*. The results of the programming operation will appear in the Output Window as shown in Figure 3-11.

To allow the code to run, the PICkit 2 must release the microcontroller /MCLR pin. The device is held in reset after programming. This means that the /MCLR pin of the microcontroller is left asserted (low) by the programmer after programming. Select *Programmer > Release from Reset*. The project code will now execute and light LED 7 on the demo board.

Congratulations! You have created, built, programmed, and executed your first Microchip PIC18F project!

FIGURE 3-11: OUTPUT WINDOW PICKIT 2 PROGRAMMING RESULTS



```
PIC18F46K20 found (Rev 0x2)
PICkit 2 Ready

Programming Target (12/12/2007  3:26:11 PM)
Erasing Target
Programming Program Memory (0x0 - 0xFF)
Verifying Program Memory (0x0 - 0xFF)
Programming Configuration Memory
Verifying Configuration Memory
PICkit 2 Ready
```

Note: If an error occurs during programming, consult the PICkit 2 help file in the MPLAB IDE. Select *Help > Topics…* then under the "Programmers" heading select "PICkit 2 Programmer" and click **OK**. On the "Contents" tab, select the "Troubleshooting" section for information.

3.2 Lesson 2: Blink LED

This lesson discusses the Configuration bits of the PIC18FXXXX microcontrollers, and how to set them in an MPLAB C18 source file. It also presents using a library function and shows how delays can be used to blink an LED on the demo board.

Key Concepts

- Open existing project workspaces by selecting *File > Open Workspace…* in the MPLAB IDE
- Configuration bits are special purpose fuse bits that set PIC microcontroller modes of operation and enable or disable microcontroller features.
- A number of libraries are included with the MPLAB C18 compiler with predefined and compiled functions. The *MPLAB C18 C Compiler Libraries* document (DS51297) provides detailed information on all included libraries.
- Delays can be created to time events by using software loops.

3.2.1 Opening the Lesson 2 Project & Workspace in the MPLAB IDE

This and the remaining lessons already have a project and workspace defined. To open the workspace for Lesson 2, select menu *File > Open Workspace…* in the MPLAB IDE. Browse to the directory `C:\Pk2 Lessons\PIC18F46K20 Demo\02 Blink LED` and open the `02 Blink LED.mcw` file.

Before opening the new workspace, the MPLAB IDE will prompt you to save the current workspace. It is generally a good idea to click **Yes.** Afterwards, the new workspace and project for Lesson 2 will open.

3.2.2 Defining Configuration Bit Settings in the Source Code

Configuration bits are fuses in the PIC18FXXXX microcontrollers that are programmed along with the application code to set up or "configure" different microcontroller operating modes and enabled or disable certain microcontroller features. For example, in the PIC18F46K20 the configuration bits select such features which oscillator option to use, whether the processor runs in traditional or extended mode, whether to use the Brown-Out-Reset circuit and which voltage to trip at, whether the Watchdog Timer is enabled or disabled and which options to use, and if the Flash memory Code Protect feature is enabled among many other options.

Note that some features, such as the Watchdog Timer, can be configured so that it may be enabled or disabled by software in the Special Function Registers while the application code is executing. For detailed descriptions and information on the PIC18F46K20 Configuration bits, see section 23.1 Configuration Bits in the datasheet, under the section heading 23.0 Special Features of the CPU.

In the Lesson 2 source code, all configuration bits are defined at the top of the `02 Blink LED.c` file, as shown in Figure 3-12.

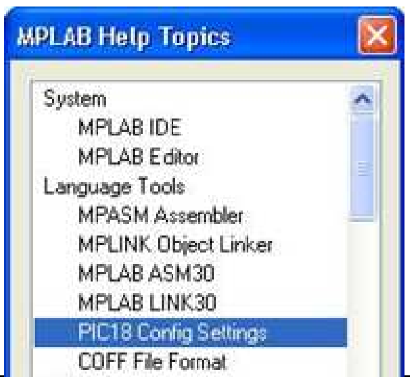
FIGURE 3-12: LESSON 2 “BLINK LED” CONFIGURATION BIT DEFINITIONS

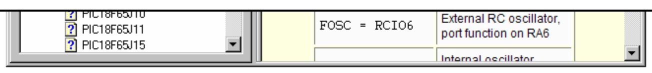
```
/** C O N F I G U R A T I O N   B I T S ******************************/

#pragma config FOSC = INTIO67, FCMEN = OFF, IESO = OFF                       // CONFIG1H
#pragma config PWRT = OFF, BOREN = SBORDIS, BORV = 30                        // CONFIG2L
#pragma config WDTEN = OFF, WDTPS = 32768                                    // CONFIG2H
#pragma config MCLRE = ON, LPT1OSC = OFF, PBADEN = ON, CCP2MX = PORTC        // CONFIG3H
#pragma config STVREN = ON, LVP = OFF, XINST = OFF                           // CONFIG4L
#pragma config CP0 = OFF, CP1 = OFF, CP2 = OFF, CP3 = OFF                    // CONFIG5L
#pragma config CPB = OFF, CPD = OFF                                          // CONFIG5H
#pragma config WRT0 = OFF, WRT1 = OFF, WRT2 = OFF, WRT3 = OFF                // CONFIG6L
#pragma config WRTB = OFF, WRTC = OFF, WRTD = OFF                            // CONFIG6H
#pragma config EBTR0 = OFF, EBTR1 = OFF, EBTR2 = OFF, EBTR3 = OFF            // CONFIG7L
#pragma config EBTRB = OFF                                                   // CONFIG7H
```

The Configuration bits are defined using the `#pragma config` directive for each configuration word. The MPLAB C18 attributes used to reference each bit or bit field setting (i.e. “`OSC = INTIO67`”) may differ from one PIC18FXXXX microcontroller to another, depending the features supported by a particular microcontroller. All the attributes available for a particular microcontroller may be found in the MPLAB IDE help. Let’s find the attributes for the PIC18F46K20:

1. Select MPLAB IDE menu _Help > Topics…_
2. In the “MPLAB Help Topics” dialog, find the “Language Tools” category and select the “PIC18 Config Settings” topic as shown in Figure 3-13. Click **OK.**
3. When the Help window opens, select the “Contents” tab, and expand the “Configuration Settings” section.
4. Select the PIC18F46K20 microcontroller to display all the configuration bit setting attributes that can be used with the `#pragma config` directive, as shown in Figure 3-14.

FIGURE 3-13: MPLAB HELP TOPICS **FIGURE 3-14: PIC18F46K20 CONFIGURATION**





FOSC = RCIO6 External RC oscillator, port function on RA6

Internal oscillator

The configuration bit settings that are important for this lesson project and are different from the default values are:

`FOSC = INTIO67`	This sets the PIC18F46K20 to run using the internal oscillator, so no crystal or external oscillator is needed. The default frequency is 1 MHz. The oscillator is covered in more detail in Lesson 9. It also sets OSC1 and OSC 2 pins to be used as the RA7 and RA7 I/O port pins as the OSC pin functions are not needed.
`WDTEN = OFF`	This turns off the Watchdog Timer, as it is not used in this lesson. When the Watchdog Timer is enabled, it must be cleared periodically in the code or it will reset the microcontroller.
`LVP = OFF`	This turns off Low-Voltage-Programming, and frees the PGM pin to be used as the RB5 I/O port pin. (LVP mode is not used by the PICkit 2 programmer.)

Even though all other bit settings are left as default, it is strongly recommended to define them all in the source as is done in the Lesson 2 source code. This ensures that the program memory image in the `.hex` file built by the compiler contains all the configuration settings intended for the target application. The one exception is the DEBUG bit, as this is defined by the MPLAB IDE environment depending on whether the target microcontroller is running in debug mode or not.

3.2.3 Exploring the Lesson 2 Source Code

Open the lesson 2 source code file `02 Blink LED.c` in an MPLAB IDE editor window if it is not open already.

FIGURE 3-15: LESSON 2 "BLINK LED" SOURCE CODE

```
/** I N C L U D E S **************************************************/
#include "p18f46k20.h"
#include "delays.h"

/** D E C L A R A T I O N S *******************************************/


void main (void)
{

      TRISD = 0b01111111;       // PORTD bit 7 to output (0) ; bits 6:0 are inputs (1)

      while (1)
      {
            LATDbits.LATD7 = ~LATDbits.LATD7; // toggle LATD

            Delay1KTCYx(50);    // Delay 50 x 1000 = 50,000 cycles; 200ms @ 1MHz
      }

}
```

This source code contains a couple of new lines of interest. The first is a new include file:

```
#include "delays.h"
```

This is header file for the MCC18 “delays” library, which provides functions used to create program delays of a certain number of processor cycles. The MPLAB C18 compiler comes with a number of useful libraries. These include the standard C libraries `stdio` & `stdlib`, and function libraries such as `ctype`, `delays`, `math`, & `string`. There are also libraries for using hardware peripheral functions such as `adc`, `i2c`, `pwm`, `spi`, `usart`, and `timers` as well as for software emulation of peripherals like `sw_i2c`, `sw_uart`, and `sw_spi`.

Headers for the libraries can be found in the MCC18 header directory `C:\MCC18\h`. The source code for most of the libraries can be found in `C:\MCC18\src`, and the libraries themselves are in `C:\MCC18\lib`. For more detailed information on the included library functions see the *MPLAB C18 C Compiler Libraries* document (DS51297).

The other new line of special interest is a function call to a function in the `delays` library:

```
Delay1KTCYx(50);
```

This function creates a time delay with a software of 1000 (1k) instruction cycles (TCY) times the argument value. In this case, the argument is 50 so this function will delay for 50 x 1000 = 50,000 instruction cycles. The instruction rate on PIC18FXXXX microcontrollers is equal to 1/4th the oscillator clock; in other words, it takes 4 clocks to execute an instruction. In this case the clock is the internal oscillator at 1MHz, so the instruction rate is 250kHz, or TCY = 4us per instruction. The total delay is 50,000 x 4us = 200ms, which is slow enough for the human eye to see the LED turning on and off.

The lesson 2 program runs this delay inside an indefinite `while` loop, which sets the RD7 I/O pin to the complement of its current value (the effect is to switch it back and forth between high and low) with a 200ms delay in between each RD7 output level change. This blinks the demo board LED 7.

3.2.4 Build and Program the Lesson 2 Code

In the MPLAB IDE, build the lesson 2 project and program the code into the demo board PIC18F46K20 using the PICkit 2 Programmer as we did in lesson 1. Don’t forget to release the microcontroller from reset!

The demo board LED 7 will blink continuously at 200ms on and 200ms off.

3.3 Lesson 3: Rotate LED

This lesson builds on the previous two lessons to introduce defining global variables and code sections, and to add rotation to the LED display. It will light up LED 0, then shift it to LED 1, then to LED 2 and on up to LED 7, and back to LED 0.

In this and following lessons, please open the lesson workspace in the MPLAB IDE upon starting the lesson.

Key Concepts

- The directives `#pragma udata` and `#pragma idata` are used to allocate memory for static variables in the file registers.
- The directive `#pragma code` is used to indicate a section of instructions to be compiled into the program memory of the PIC18FXXXX.
- The directive `#pragma romdata` is used for constant (read-only) data stored in the program memory. This is used with the keyword `rom`.
- Constant data can be stored in program memory so as not to use up file register RAM.

3.3.1 Allocating File Register Memory

In the source code file `03 Rotate LED.c` for lesson 3 the global variable, LED_Number, is declared as in Figure 3-16.

FIGURE 3-16 LESSON 3 GLOBAL VARIABLE DECLARATION

```
/** V A R I A B L E S *************************************************/
#pragma udata // declare statically allocated uninitialized variables
unsigned char LED_Number;  // 8-bit variable
```

The directive #pragma udata is used prior to declaring the variable LED_Number to indicate to the compiler that the following declarations are data variables that should be placed in the PIC18FXXXX file registers. This differs from PC compilers where instructions and variables share the same memory space due to the Harvard architecture of the PIC18FXXXX as discussed in section 2.1 of this document.

There are two directives for use with `#pragma` when defining variables:

`udata`	Uninitialized data. The following data is stored uninitialized in the file register space.
`idata`	Initialized data. The following data is stored in the file register space. The initialization values are stored in program memory, and then moved by the startup initialization code into file registers before program execution begins.

Data declarations can also be given a section name. The section name may be used with a linker script SECTION entry to place it in a particular area of memory. See section 2.9 of the *MPLAB C18 C Compiler User's Guide* for more information on using sections with linker scripts. Even without a linker script section, the #pragma udata directive may be used to specify the starting address of the data in the file registers. For example, to place `LED_Number` at the start of file register bank 3 declare the `udata` section as

```
#pragma udata mysection = 0x300
unsigned char LED_Number;  // 8-bit variable
unsigned int AnotherVariable;
```

Other variables declared in a `udata` or `idata` section will be placed at subsequent addresses. For instance, the 16-bit integer `AnotherVariable above would occupy address 0x301 and 0x302.`

Note that function local variables will be placed on the software stack.

For a list of data types supported by MPLAB C18, their sizes and limits, see section 2.1 of the *MPLAB C18 C Compiler User's Guide* (DS51288).

3.3.2 Allocating Program Memory

Program memory will most often be used for program instructions and constant data. The source code for lesson 3 includes examples of both, as shown in Figure 3-17.

FIGURE 3-17: LESSON 3 CONSTANT DATA AND PROGRAM CODE

```
/** D E C L A R A T I O N S *******************************************/
// declare constant data in program memory starting at address 0x180
#pragma romdata Lesson3_Table = 0x180
const rom unsigned char LED_LookupTable[8] = {0x01, 0x02, 0x04, 0x08,
                                              0x10, 0x20, 0x40, 0x80};

#pragma code    // declare executable instructions

void main (void)
{
```

There are two basic directives for defining program memory sections:

`code`	Program Memory Instructions. Compiles all subsequent instructions into the program memory space of the target PIC18FXXXX.
`romdata`	Data stored in program memory. Used in conjuction with the `rom` keyword, the following constant data is compiled into the program memory space.

In this lesson, we use a constant array `LED_LookupTable` to convert a number representing LEDs 0-7 to a bit pattern for setting the appropriate PORTD pin to turn on the corresponding LED. This constant is declared in a `romdata` section and uses the `rom` keyword so it will be placed in program memory. As the program never needs to change these array values, this saves file registers to be used for true variables.

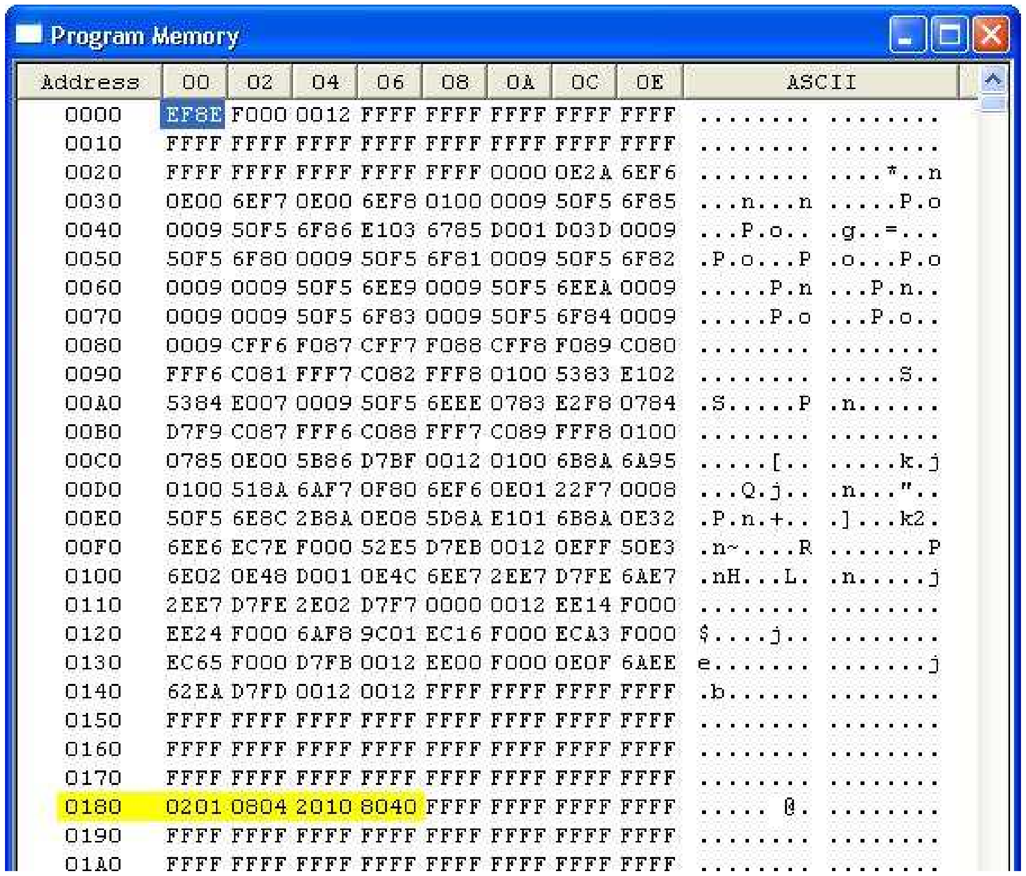
Note that the `romdata` section was also declared with a section name and absolute address:

```
#pragma romdata Lesson3_Table = 0x180
```

These optional attributes will force the compiler to place the 8 – byte char array at program memory address 0x0180. If an address is not specified, the `code` or `romdata` section may not always be placed at a deterministic address by the linker.

Select MPLAB IDE men *Project > Build All* to build the lesson 3 code, then select *View > Program Memory* to display the compiled contents of program memory. The instructions to execute the lesson program code are contained within addresses 0x0000 and 0x0146. Note that the array values have been compiled to program memory starting at the specified address of 0x180 through address 0x186 as shown in Figure 3-18.

FIGURE 3-18: PROGRAM MEMORY "LED_LOOKUPTABLE" ARRAY VALUES



```
Address   00   02   04   06   08   0A   0C   0E         ASCII
 0000    EF8E F000 0012 FFFF FFFF FFFF FFFF FFFF  ........ ........
 0010    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0020    FFFF FFFF FFFF FFFF FFFF 0000 0E2A 6EF6  ........ ....*..n
 0030    0E00 6EF7 0E00 6EF8 0100 0009 50F5 6F85  ...n...n .....P.o
 0040    0009 50F5 6F86 E103 6785 D001 D03D 0009  ...P.o.. .g..=...
 0050    50F5 6F80 0009 50F5 6F81 0009 50F5 6F82  .P.o...P .o...P.o
 0060    0009 0009 50F5 6EE9 0009 50F5 6EEA 0009  .....P.n ...P.n..
 0070    0009 0009 50F5 6F83 0009 50F5 6F84 0009  .....P.o ...P.o..
 0080    0009 CFF6 F087 CFF7 F088 CFF8 F089 C080  ........ ........
 0090    FFF6 C081 FFF7 C082 FFF8 0100 5383 E102  ........ .....S..
 00A0    5384 E007 0009 50F5 6EEE 0783 E2F8 0784  .S.....P .n......
 00B0    D7F9 C087 FFF6 C088 FFF7 C089 FFF8 0100  ........ ........
 00C0    0785 0E00 5B86 D7BF 0012 0100 6B8A 6A95  .....[.. .....k.j
 00D0    0100 518A 6AF7 0F80 6EF6 0E01 22F7 0008  ...Q.j.. .n..."..
 00E0    50F5 6E8C 2B8A 0E08 5D8A E101 6B8A 0E32  .P.n.+.. .]...k2.
 00F0    6EE6 EC7E F000 52E5 D7EB 0012 0EFF 50E3  .n~....R .......P
 0100    6E02 0E48 D001 0E4C 6EE7 2EE7 D7FE 6AE7  .nH...L. .n.....j
 0110    2EE7 D7FE 2E02 D7F7 0000 0012 EE14 F000  ........ ........
 0120    EE24 F000 6AF8 9C01 EC16 F000 ECA3 F000  $....j.. ........
 0130    EC65 F000 D7FB 0012 EE00 F000 0E0F 6AEE  e....... .......j
 0140    62EA D7FD 0012 0012 FFFF FFFF FFFF FFFF  .b...... ........
 0150    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0160    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0170    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0180    0201 0804 2010 8040 FFFF FFFF FFFF FFFF  ..... @. ........
 0190    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 01A0    FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
```

The directive `#pragma` code is then used to specify the following section, beginning with the `main ()` declaration, will be executable instructions to place in program memory. Since an optional section name and address are not specified, the code instructions will be placed at the first available address by the linker. As with data directives, a section name may used with a SECTION entry in the linker script to allocated a range of program memory for a section.

3.3.3 Exploring the Lesson 3 Source Code

Open the lesson source code file `03 Rotate LED.c` in an editor window if it is not open already.

FIGURE 3-19: LESSON 3 “ROTATE LED” SOURCE CODE

```
/** V A R I A B L E S *************************************************/
#pragma udata // declare statically allocated uninitialized variables
unsigned char LED_Number;  // 8-bit variable

/** D E C L A R A T I O N S *******************************************/
// declare constant data in program memory starting at address 0x180
#pragma romdata Lesson3_Table = 0x180
const rom unsigned char LED_LookupTable[8] = {0x01, 0x02, 0x04, 0x08,
                                              0x10, 0x20, 0x40, 0x80};

#pragma code    // declare executable instructions

void main (void)
{
    LED_Number = 0;            // initialize

    TRISD = 0b00000000;           // PORTD bits 7:0 are all outputs (0)

    while (1)
    {
            // use lookup table to output one LED on based on LED_Number value
        LATD = LED_LookupTable[LED_Number];

        LED_Number++;      // rotate display by 1

        if (LED_Number == 8)
            LED_Number = 0;    // go back to LED 0.

        Delay1KTCYx(50);       // Delay 50 x 1000 = 50,000 cycles; 200ms @ 1MHz
    }
}
```

Here is the basic flow of our Rotate LED program:

Initialize Variables & I/O Port

The global variable `LED_Number`, which holds the number of the LED we currently want on, is set to ‘0’ for the first LED.

The TRISD register bits are all set to ‘0’, so that all 8 port D pins RD0 – RD7 are outputs.

Loop Forever with the `while(1)` statement:

Set the I/O Port to turn on an LED.

The number of the LED to turn on, `LED_Number`, is used an index to the array `LED_LookupTable` which returns a value with a bit set corresponding to the LED to be turned on. This value is written to the LATD register to turn on the one LED.

Rotate the LED number

The LED number is incremented to the next LED. The `if` statement checks to see if it has been incremented past the last LED. If so, it is reset to the first LED, number 0.

Delay 200ms

As in Lesson 2, a “delays” library function is used to create a time delay.

(Loop End)

3.3.4 Build and Program the Lesson 3 Code

In the MPLAB IDE, build the lesson 3 project and program the code into the demo board using the PICkit 2 Programmer. Don't forget to release the microcontroller from reset!

The demo board LEDs will rotate from LED 0 up to LED 7 and then back to LED 0.

3.4 Lesson 4: Switch Input

The demo board switch is used in the lesson to rotate the LEDs once on each press.

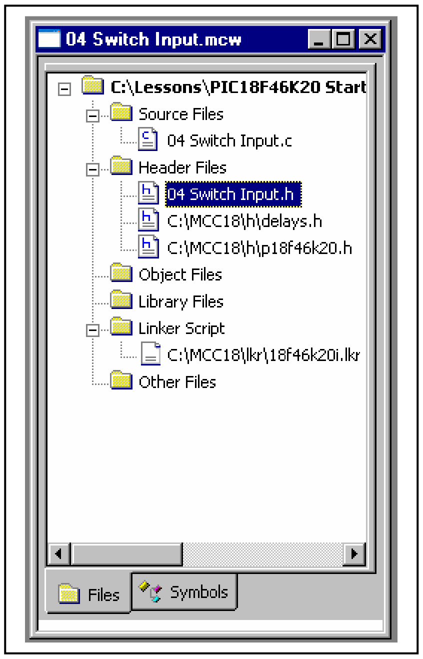
Key Concepts

- The directive #define can be used to give SFR registers and bits more meaningful names.
- I/O pins that share an analog input channel must be configured as digital pins if used as digital inputs using SFR ADCON1, or they will always read '0'.
- The PORTx SFRs are used to read the logic state on an input port pin.
- Mechanical switch debouncing can be handled in software to eliminate external components that may be otherwise required.

3.4.1 Header Files and the #define Directive

This lesson has added a header file to the project named `04 Switch Input.h` as shown in Figure 3-20.

FIGURE 3-20 HEADER FILE



```
04 Switch Input.mcw
C:\Lessons\PIC18F46K20 Start
  Source Files
    04 Switch Input.c
  Header Files
    04 Switch Input.h
    C:\MCC18\h\delays.h
    C:\MCC18\h\p18f46k20.h
  Object Files
  Library Files
  Linker Script
    C:\MCC18\lkr\18f46k20i.lkr
  Other Files
```

While it is assumed that the reader is familiar with C language header files, we'll note that in the `04 Switch Input.h` header file the `#define` directive has been used to give more meaningful names to the switch I/O pin variable and a constant value.

```
    #define Switch_Pin      PORTBbits.RB0
    #define DetectsInARow   5
```

As with other C compilers use of `#define`, MPLAB C18 will replace all instances of the text "Switch_Pin" with the text "PORTBbits.RB0" at compile time.

Remember, for the compiler to know about the `#define` definitions, the header file must be included in the C file, as is done in `04 Switch Input.c`:

```
    #include "04 Switch Input.h"  // header file
```

3.4.2 Switch Debouncing

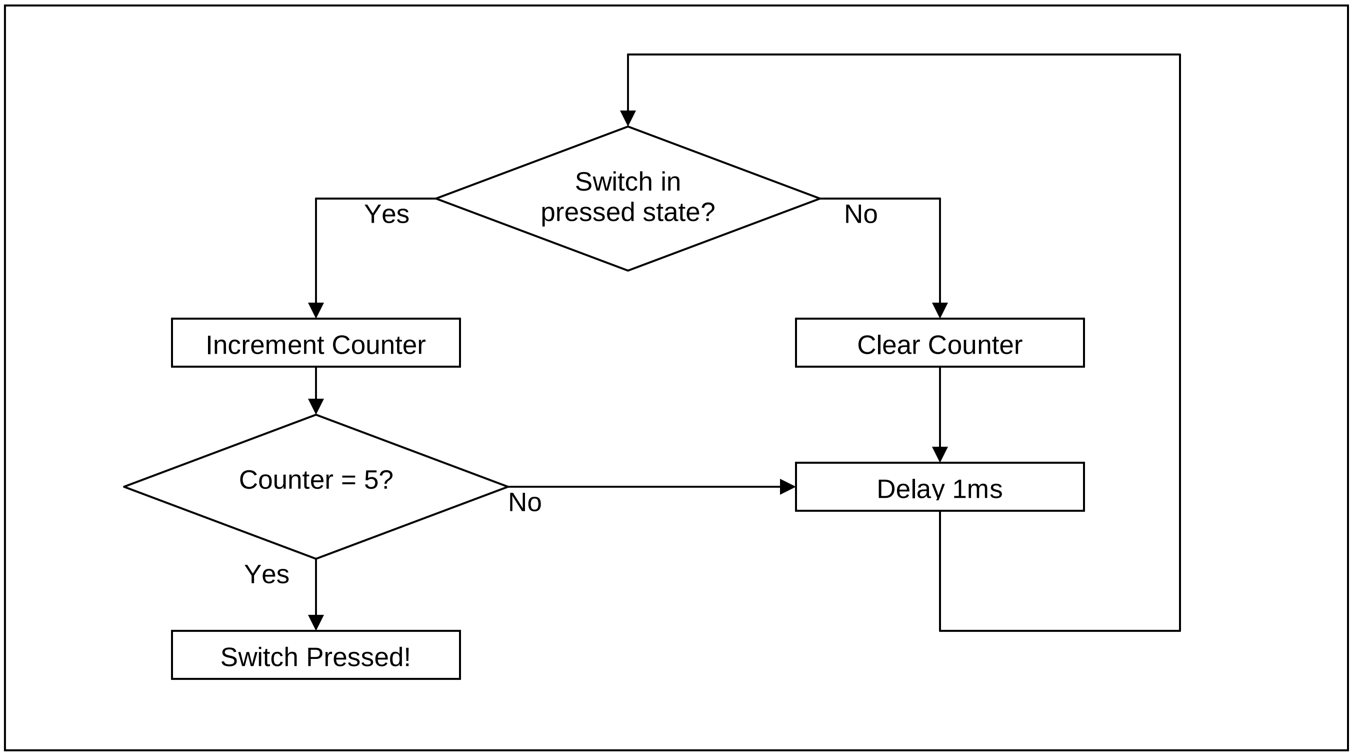
Mechanical switches are frequently encountered in embedded processor applications, and are inexpensive, simple, and reliable. However, such switches are also often very electrically noisy. This noise is known as switch bounce, whereby the connection between the switch contacts makes and breaks several, perhaps even hundreds, of times before settling to the final switch state. This can cause a single switch push to be detected as several distinct switch pushes by a fast device, especially with an edge-sensitive input. Think of advancing the TV channel, but instead of getting the next channel, the selection skips ahead two or three.

Classic solutions to switch bounce involved filtering out the fast switch bounce transitions with a resistor-capacitor circuit, or using re-settable logic shift registers. While effective, these methods add additional cost and increase circuit board real estate. Debouncing a switch in software eliminates these issues.

A simple way to debounce a switch is to sample the switch until the signal is stable. How long to sample requires some investigation of the switch characteristics, but usually 5ms is sufficiently long.

This lesson code demonstrates sampling the switch input every 1mS, waiting for 5 consecutive samples of the same value before determining that the switch was pressed. Note that the switch on the 44-Pin Demo Board doesn't bounce much, but it is good practice to debounce all system switches.

FIGURE 3-21: SWITCH DEBOUNCING PROGRAM FLOW



3.4.3 Exploring the Lesson 4 Source Code

Open the lesson source code file `04 Switch Input.c` in an editor window if it is not open already.

FIGURE 3-22: LESSON 4 “SWITCH INPUT” SOURCE CODE

```
/** V A R I A B L E S *************************************************/
#pragma udata   // declare statically allocated uinitialized variables
unsigned char LED_Display;  // 8-bit variable

/** D E C L A R A T I O N S *******************************************/
#pragma code    // declare executable instructions

void main (void)
{
    unsigned char Switch_Count = 0;

    LED_Display = 1;            // initialize

    TRISD = 0b00000000;          // PORTD bits 7:0 are all outputs (0)

    INTCON2bits.RBPU = 0;        // enable PORTB internal pullups
    WPUBbits.WPUB0 = 1;          // enable pull up on RB0
    ANSELH = 0x00;              // AN8-12 are digital inputs (AN12 on RB0)
    TRISBbits.TRISB0 = 1;       // PORTB bit 0 (connected to switch) is input (1)

    while (1)
    {
        LATD = LED_Display;     // output LED_Display value to PORTD LEDs

        LED_Display <<= 1;      // rotate display by 1

        if (LED_Display == 0)
            LED_Display = 1;    // rotated bit out, so set bit 0


        while (Switch_Pin != 1);// wait for switch to be released

        Switch_Count = 5;
        do
        { // monitor switch input for 5 lows in a row to debounce
            if (Switch_Pin == 0)
            { // pressed state detected
                Switch_Count++;
            }
            else
            {
                Switch_Count = 0;
            }
            Delay10TCYx(25);    // delay 250 cycles or 1ms.
        } while (Switch_Count < DetectsInARow);
```

Variables

This program has 2 declared variables, the global variable `LED_Display` and the local variable `Switch_Count`. A global variable will be placed in a dedicated location in the file register space as discussed in lesson 3. A local variable is placed on the software stack, and is created when a function is entered, and destroyed (removed from the stack) when the function exits.

Switch Input

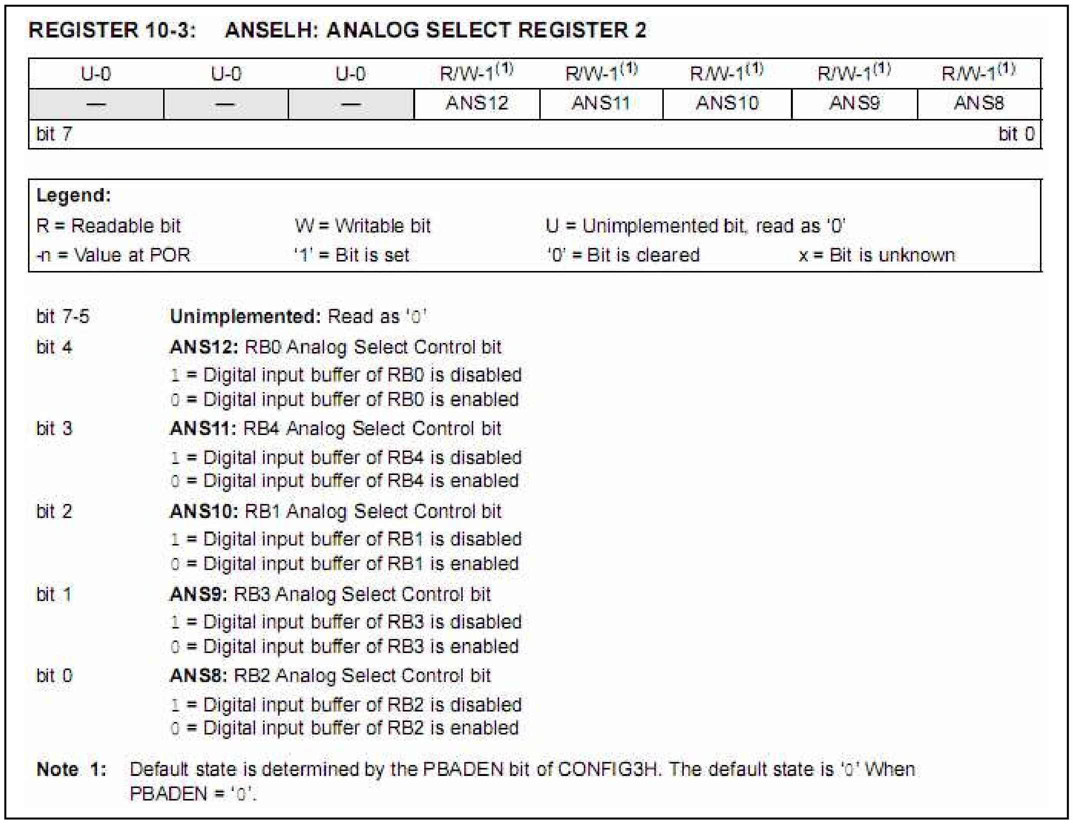
The demo board switch is connected to I/O pin RB0, which is normally pulled up to VDD internally. When the switch is pressed, it pulls RB0 to ground (low state).

The PORTx special function registers are used to read the state of an input pin. Therefore, reading PORTBbits.RB0 will give the value of the signal on the RB0 pin. Don't forget – in the header file, this was defined as `Switch_Pin`, which is what the code uses to read the pin value:

```
#define Switch_Pin      PORTBbits.RB0
```

In the PIC18F46K20, the RB0 pin is shared with analog input AN12. Such pins must be configured as either digital or analog inputs. This is important because RB0 will be used as a digital input pin to read the state of the switch in register PORTB. If RB0 is configured as an ***analog*** input, it will always read '0' and not the actual state of the switch. Pins are configured as analog or digital in the SFRs ANSEL and ANSELH.

FIGURE 3-23: ANSELH: ANALOG REGISTER 1



REGISTER 10-3: ANSELH: ANALOG SELECT REGISTER 2

U-0	U-0	U-0	R/W-1(1)	R/W-1(1)	R/W-1(1)	R/W-1(1)	R/W-1(1)
—	—	—	ANS12	ANS11	ANS10	ANS9	ANS8
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 7-5 **Unimplemented:** Read as '0'

bit 4 **ANS12:** RB0 Analog Select Control bit
1 = Digital input buffer of RB0 is disabled
0 = Digital input buffer of RB0 is enabled

bit 3 **ANS11:** RB4 Analog Select Control bit
1 = Digital input buffer of RB4 is disabled
0 = Digital input buffer of RB4 is enabled

bit 2 **ANS10:** RB1 Analog Select Control bit
1 = Digital input buffer of RB1 is disabled
0 = Digital input buffer of RB1 is enabled

bit 1 **ANS9:** RB3 Analog Select Control bit
1 = Digital input buffer of RB3 is disabled
0 = Digital input buffer of RB3 is enabled

bit 0 **ANS8:** RB2 Analog Select Control bit
1 = Digital input buffer of RB2 is disabled
0 = Digital input buffer of RB2 is enabled

Note 1: Default state is determined by the PBADEN bit of CONFIG3H. The default state is '0' When PBADEN = '0'.

We clear ANSELH to set all pins to digital functionality:

```
ANSELH = 0x00;
```

Now we can use RB0 as a digital input, so the TRISB bit is set to configure it as an input:

```
TRISBbits.TRISB0 = 1;
```

Rotating the LEDs

This program uses a simpler method of rotating the LEDs than lesson 3, which used the lookup table for demonstration purposes. 04 Switch Input.c uses a single set bit in the LED_Display variable which is written to LATD and shifted each time the display is updated. The bit will eventually be shifted out of the most significant bit of LED_Display, so the code checks for this, and sets LED_Display to ‘1’ again.

For more information on I/O port pins, see Section 10 I/O Ports of the PIC18F46K20 datasheet.

3.4.4 Build and Program the Lesson 4 Code

Build the lesson 4 project and program the code into the demo board using the PICkit 2 Programmer. Don’t forget to release the microcontroller from reset!

Press the demo board switch button to rotate the LEDs. The LEDs will advance once for each button press.

3.5 Lesson 5: Using Timer0

Timer0 is used to time delays while rotating the demo board LEDs, instead of using program loop delays. The demo board switch reverses the direction of the rotation.

Key Concepts

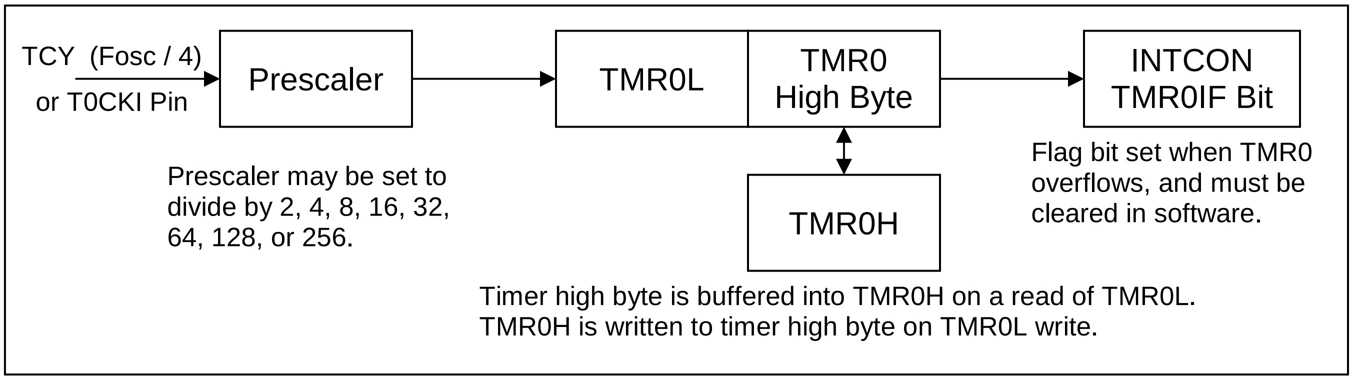
- Timer0 is hardware counter implemented in the microcontroller that can count clock cycles or external events.
- Using a timer instead of processor delay loops frees up the processor to do useful work instead of counting cycles.
- A timer "prescaler" sets the number of clock cycles or events required to increment the timer by 1, allowing it to be run faster or slower off the same frequency clock.

3.5.1 The PIC18F46K20 Timer0 Module

The Timer0 module is timer/counter peripheral of the PIC18FXXXX microcontroller that may be used to count oscillator clock cycles or external events on the T0CKI pin. It can be configured as an 8-bit or 16-bit timer, which means it can count from 0 to 255 or 0 to 65535. A bit flag is set when the counter rolls over from the maximum value back to zero.

The Timer0 module also includes an optional prescaler, which may be configured to divide the timer clock source before it reaches the timer/counter itself. For example, with a 1:1 prescaler, the timer would increment once every instruction clock cycle. (Remember that the instruction clock cycle TCY is the Fosc oscillator clock/4.) With a 1:8 prescaler, the timer would increment once every eight clock cycles. The prescaler is cleared on every write to the timer.

FIGURE 3-23: SIMPLIFIED 16-BIT TIMER0 BLOCK DIAGRAM

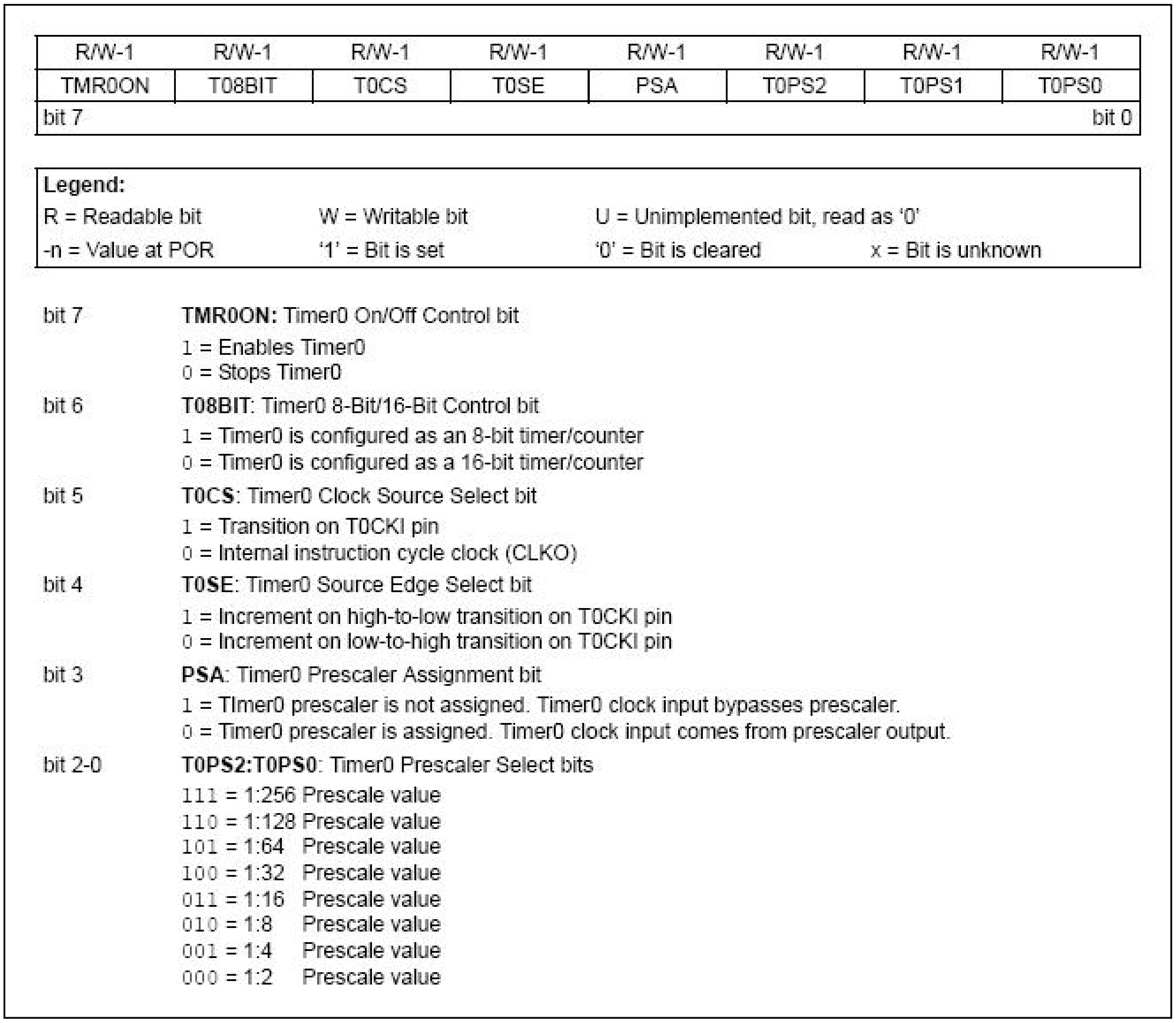

TCY (Fosc / 4)
or T0CKI Pin
Prescaler
TMR0L
TMR0
High Byte
INTCON
TMR0IF Bit
TMR0H
Prescaler may be set to divide by 2, 4, 8, 16, 32, 64, 128, or 256.
Flag bit set when TMR0 overflows, and must be cleared in software.
Timer high byte is buffered into TMR0H on a read of TMR0L.
TMR0H is written to timer high byte on TMR0L write.

When Timer0 is configured as a 16-bit timer, care must be taken when reading and writing the timer value. The lower byte of the timer is directly readable and writable as the SFR TMR0L. However, the high byte is not directly accessible. Instead, it is buffered through the SFR TMR0H. TMR0H is updated with the value of timer high byte when TMR0L is read. A write of TMR0L also writes the contents of TMR0H to the Timer0 high byte. This allows the entire 16-bit timer to be read or written at once.

Therefore, to read the timer, always read TMR0L first, then TMR0H. To write the timer, always write TMR0H first then TMR0L.

Timer0 operation is controlled by the T0CON SFR, shown in Figure 3-24.

FIGURE 3-24: T0CON: TIMER0 CONTROL REGISTER



R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
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	**TMR0ON:** Timer0 On/Off Control bit 1 = Enables Timer0 0 = Stops Timer0
bit 6	**T08BIT**: Timer0 8-Bit/16-Bit Control bit 1 = Timer0 is configured as an 8-bit timer/counter 0 = Timer0 is configured as a 16-bit timer/counter
bit 5	**T0CS**: Timer0 Clock Source Select bit 1 = Transition on T0CKI pin 0 = Internal instruction cycle clock (CLKO)
bit 4	**T0SE**: Timer0 Source Edge Select bit 1 = Increment on high-to-low transition on T0CKI pin 0 = Increment on low-to-high transition on T0CKI pin
bit 3	**PSA**: Timer0 Prescaler Assignment bit 1 = TImer0 prescaler is not assigned. Timer0 clock input bypasses prescaler. 0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.
bit 2-0	**T0PS2:T0PS0**: Timer0 Prescaler Select bits 111 = 1:256 Prescale value 110 = 1:128 Prescale value 101 = 1:64 Prescale value 100 = 1:32 Prescale value 011 = 1:16 Prescale value 010 = 1:8 Prescale value 001 = 1:4 Prescale value 000 = 1:2 Prescale value

To use Timer0 to replace the software delay `Delay1KTCYx(50)` it should be set up so it overflows about every 200 to 300ms. Let's go over the T0CON bit settings to make this happen:

T08BIT = 0

Timer0 is configured as a 16-bit timer/counter to illustrate the buffering of TMR0H.

T0CS = 0

Timer0 runs off the internal instruction clock. At Fosc = 1MHz, the instruction clock is 250kHz.

T0SE = 0

If Timer0 was running off the T0CKI pin, this bit would determine whether it incremented on the falling edge or rising edge of the T0CKI pin signal. Since we are running off the instruction clock, this bit is a “don’t care.” This means operation is not affected by either setting of this bit.

PSA = 1

The timer will overflow in 65536 counts. At the instruction clock rate of 250kHz, the timer overflow will occur every 65536 x (1 / 250,000) = 262ms. This is a time in the range we want, so the prescaler is **not** assigned to Timer0. It runs directly off the instruction clock.

T0PS2:T0PS0 = 000

Since the prescaler is not assigned, these bits are “don’t care.”

And finally:

TMR0ON = 0

This bits turns the timer and off. It’s set to zero now as the timer will be turned on once it is has been set up.

To configure Timer 0 with these settings, the binary value 0b0000100 is written to T0CON.

The PIC18F46K20 has 3 other configurable timers: Timer1, Timer2, and Timer3. More information on all four timer modules can be found in the PIC18F46K20 datasheeet sections 11 through 14.

3.5.2 Exploring the Lesson 5 Source Code

Open the lesson source code file `05 Timer.c` and header file `05 Timer.h` in editor windows if they are not open already.

Note that in `05 Timer.h` two custom enumerated variable types have been defined:

```
typedef enum { LEFT2RIGHT,
               RIGHT2LEFT} LEDDirections;

typedef enum {FALSE, TRUE} BOOL;
```

This allows us to declare variables using these types and initialize them in `main()`:

```
LEDDirections Direction = LEFT2RIGHT;
BOOL SwitchPressed = FALSE;
```

The `Direction` variable keeps track of which direction the LEDs are rotating in, and `SwitchPressed` remembers if the switch has been pressed or not, as the LED rotation direction should only be changed once when it is pressed.

The following code before the `while(1)` loop sets up the Timer0 module as discussed in previously.

```
// Init Timer
INTCONbits.TMR0IF = 0;    // line 1
T0CON = 0b00001000;       // line 2
// T0CON = 0b00000001;        (ignore commented line for now)
TMR0H = 0;                // line 3
TMR0L = 0;                // line 4
T0CONbits.TMR0ON = 1;     // line 5
```

Using the line numbers in the comments as references, let's discuss the function of each line in setting up the timer.

Line 1 clears the TMR0IF flag in the INTCON SFR. This bit flag is set whenever the timer overflows (rolls over), so the program will poll it to know when the LED rotation delay it up. However, the flag will not reset by hardware, it must be reset in software so the program makes sure it is clear before starting the timer.

Line 2 loads the settings into T0CON to configure the timer as discuss previously in this lesson.

Line 3 clears the TMR0H buffer. Remember that TMR0H only buffers the high byte of the timer. The '0' value will not actually be written to the timer upper byte until TMR0L is written.

Line 4 clears TMR0L, which also causes TMR0H to be written to the high byte of the timer. Thus, the entire 16-bit timer is loaded with the hex value 0x0000.

Line 5 sets bit 7, TMR0ON, of the T0CON register to turn on the timer so it begins incrementing. Using one of the SFR unions to access bits, like `T0CONbits.TMR0ON`, can change bits without affecting the other bits.

Note: Be aware that some cases using an SFR union to access a bit may affect other bits. What actually happens during this instruction execution is the register is read, the bit is modified, and the entire register is re-written. This operation is called Read-Modify-Write. If a bit reads a different value than what it was last set as, this operation may affect register bits other than the intended one. Check the SFR bit definitions carefully. In the case of T0CON, all bits are Read/Write and all are set by software only; the hardware will not affect any bit setting.

Moving on the rest of the lesson code: In the `while(1)` loop, the `LED_Display` global variable is updated to rotate the '1' bit based on the `Direction` variable value, and then LATD is updated.

The `do{…}while()` loop then polls the switch looking for a switch press while it waits for the timer to overflow and set the TMR0IF flag bit. This is a simplistic example of how using a timer allows the microcontroller to do work while waiting on a time delay, instead of wasting processing time counting cycles in an instruction loop.

Once the switch it pressed, the `Direction` variable value is reversed. Follow the `if – else if` logic flow in the `do{…}while()` loop to see how once the switch is pressed, the direction is reversed only once until it is released and pressed again.

Lastly, once Timer0 overflows and sets the TMR0IF flag the `do{…}while()` loop is exited. TMR0IF is then cleared in the software program so the next timer overflow can be detected.

3.5.3 Build and Program the Lesson 5 Code

Build and program the lesson 5 project. The LEDs will rotate, and pressing the demo board button will reverse them.

3.5.4 Assigning the Timer0 Prescaler

Now we'll go back to that commented-out line of code in the Timer0 setup statements. Comment out the T0CON assignment statement, and un-comment the other statement so the Timer0 setup code looks like this:

```
INTCONbits.TMR0IF = 0;
//T0CON = 0b00001000;
T0CON = 0b00000001;
TMR0H = 0;
TMR0L = 0;
T0CONbits.TMR0ON = 1;
```

Take a look at what this changes:

PSA = 0
The prescaler is now assigned to Timer0, and the values of T0PSx will set the prescaler clock divider ratio.

T0PS2:T0PS0 = 001
This value sets the prescale value to 1:4, which means Timer0 will now increment once every 4 instruction cycles, instead of once every instruction cycle. It now takes 4 times as long for it count up to 65536 – just over 1 second!

Rebuild and re-program the lesson 5 project with change in the source code. The LEDs will rotate more slowly, 4 times slower to be exact, than before.

3.6 Lesson 6: Using PICkit 2 Debug Express

This lesson covers using the PICkit 2 as an In-Circuit-Debugger (ICD). It uses the same MPLAB IDE workspace and project as lesson 5. Set T0CON assignment back to the "no prescale" statement if it was changed in the last lesson.

Key Concepts

- An In-Circuit-Debugger like PICkit 2 or MPLAB ICD 2 uses some on-chip resources to enabled debugging. These reserved file registers and program memory locations are marked 'R' in the MPLAB IDE views, and are not available for use by the user application.
- Debugging also reserves one level of the hardware return address stack and two I/O pins.
- Debugging allows the program to be run, halted, stepped-through statement by statement, and for breakpoints to be set on program statements.
- The number of available breakpoints depends on the PIC microcontroller being used.

Note: This lesson uses the project and source code from Lesson 5: Using Timer0.

3.6.1 Resources Reserved by the PICkit 2 Debug Express

Note that "PICkit 2 Debug Express" simply refers to using the PICkit 2 as a debugger.

The PICkit 2 Debug Express uses some on-chip resources to enable debugging. The resources are not available to the user application code.

General Resources

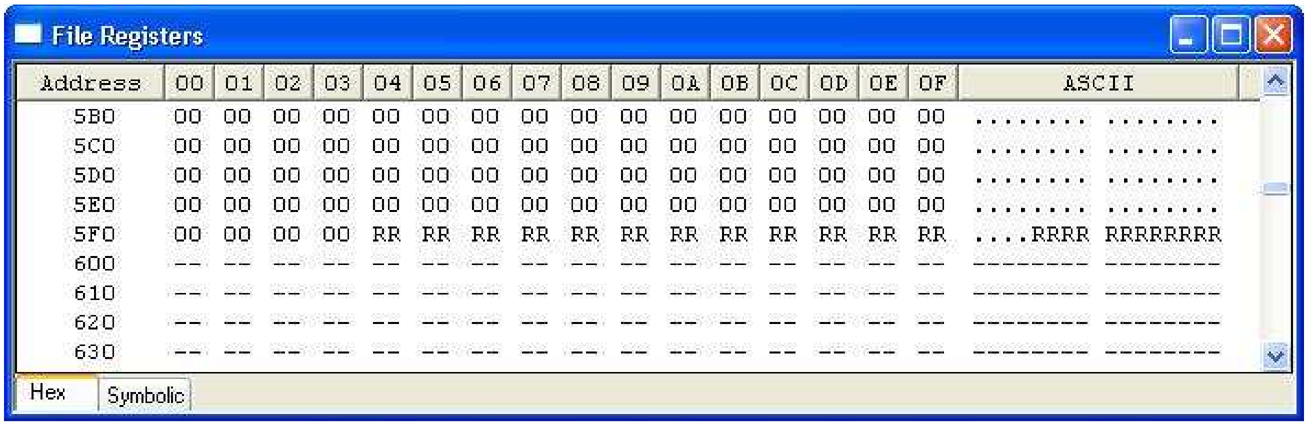
- MCLR pin reserved for debugging; this pin cannot be used as digital I/O while debugging.
- The PGD and PGC port pins are reserved for programming and in-circuit debugging. Therefore, other functions multiplexed on these pins will not be available during debug.
- One stack level is used by the debugger and not available.

Program and Data Memory Resources

The PICkit™ 2 Debug Express uses program memory and file register locations in the target device during debugging. These locations are not available for use by user code. In the MPLAB IDE, registers marked with an "R" in register displays represent reserved registers, as shown in Figure 3-25.

For device specific reserved locations, see MPLAB® IDE help for the MPLAB® ICD 2. In the MPLAB® IDE, select menu *Help > Topics…* . In the Help Topics dialog under "Debuggers", select "MPLAB® ICD 2" and click **OK**. In the MPLAB® ICD 2 Help dialog under the "Contents" tab, select "MPLAB® ICD 2 Overview" then "Resources Used By MPLAB® ICD 2". A list of device families will be presented. Select the device family of interest for more information on reserved device resources.

FIGURE 3-25: RESERVED ICD FILE REGISTER LOCATIONS IN THE PIC18F46K20



```
Address  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ASCII
    5B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ........
    5C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ........
    5D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ........
    5E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ........
    5F0  00 00 00 00 RR RR RR RR RR RR RR RR RR RR RR RR  ....RRRR RRRRRRRR
    600  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  -------- --------
    610  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  -------- --------
    620  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  -------- --------
    630  -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  -------- --------
```

Note: An ICD ‘i’ Linker Script must be used when debugging, as discussed in Section 3.1.1 of this document. The lesson projects already use the correct linker script, “18f46k20**i**.lkr”.

3.6.2 Selecting PICkit 2 as a debugger in the MPLAB IDE

The PICkit 2 cannot be used as a programmer and debugger at the same time, so if PICkit 2 is currently selected as a programmer, selecting it as a debugger will cause it to be disabled as a programmer.

To enable the PICkit 2 as a debugger in the MPLAB IDE select *Debugger > Select Tool > 6 PICkit 2*. the Output window will display the connection to the target microcontroller as in Figure 3-10.

To Begin Debugging

- Build the project: *Project > Build All*
- Program the target microcontroller: *Debugger > Program*
  After programming the target, the Output window will display “Debug mode entered, DE Version = 1.0.3” if debug mode is successfully entered.
- Select *Debugger > Run* to begin program execution.

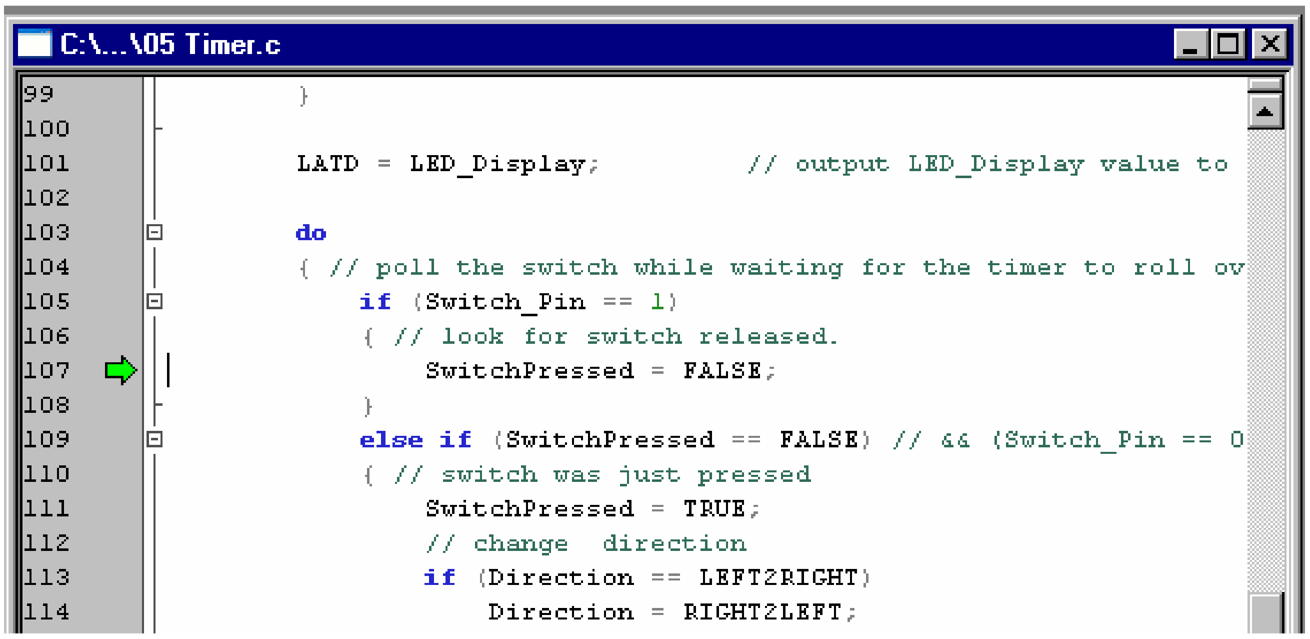
The lesson 5 code is now running in debug mode. The LEDs will rotate and the button may be pressed to reverse them, as the target microcontroller will operate in debug mode just as it normally would.

3.6.3 Basic Debug Operations

Halt

The PIC18F46K20 on the demo board is now running the lesson program code. Code execution can be halted (stopped) at any time by selecting *Debugger > Halt* <F5>. A green arrow on the left margin of the MPLAB IDE editor window will show the next statement to be executed. Your code will probably have stopped in a different place than that shown in Figure 3-26.

FIGURE 3-26: GREEN ARROW POINTS TO NEXT STATEMENT TO EXECUTE



```
        }

        LATD = LED_Display;          // output LED_Display value to

        do
        { // poll the switch while waiting for the timer to roll ov
            if (Switch_Pin == 1)
            { // look for switch released.
                SwitchPressed = FALSE;
            }
            else if (SwitchPressed == FALSE) // && (Switch_Pin == 0
            { // switch was just pressed
                SwitchPressed = TRUE;
                // change  direction
                if (Direction == LEFT2RIGHT)
                    Direction = RIGHT2LEFT;
```

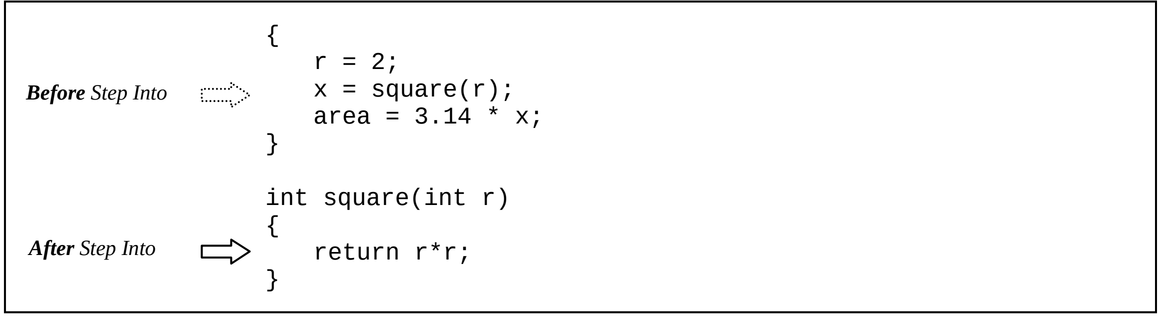
Step

Stepping, also known as single-stepping, allows the code to be executed one statement at a time. There are three step options:

Step Into

This will step through statements one at a time, until a function call is reached. When Step Into is selected on a function call, the debugger will step to the first statement in the called function. Shortcut key is <F7>

FIGURE 3-27: STEP INTO FUNCTION

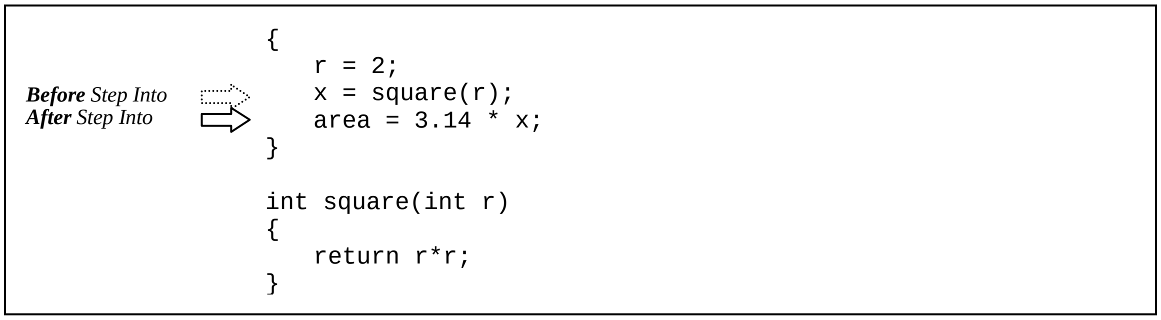


```
{
    r = 2;
    x = square(r);
    area = 3.14 * x;
}

int square(int r)
{
    return r*r;
}
```

Step Over

This will step through statements one at a time. When a statement includes a function call, the entire function will executed and the debugger will step to the next statement after the function call. It will not step into the function. Shortcut key is <F8>

FIGURE 3-28: STEP OVER FUNCTION



```
{
   r = 2;
   x = square(r);
   area = 3.14 * x;
}

int square(int r)
{
   return r*r;
}
```

Step Out

This completes execution of the current function and steps to the next statement after the function call.

You can step through lesson code by using the shortcut key for *Debugger > Step Over*, `<F8>`.

Run

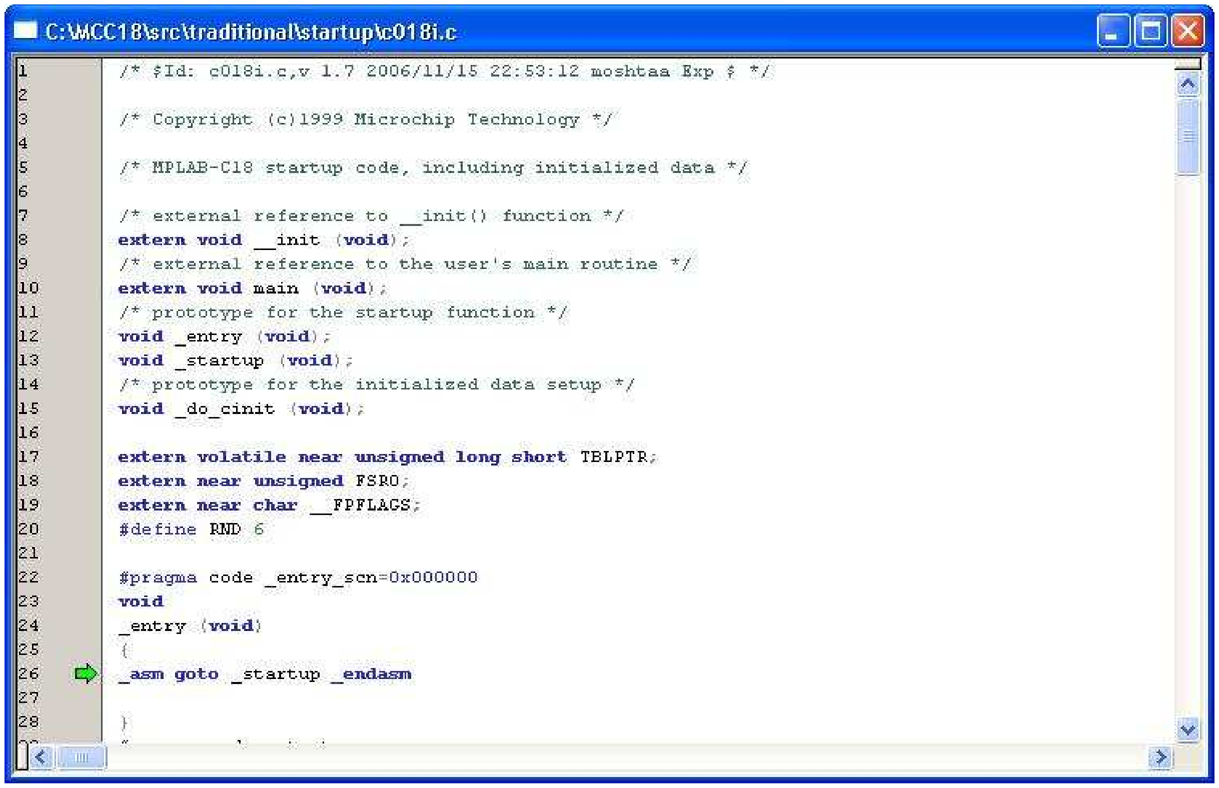
Debugger > Run <F9> will begin code execution until it is halted by the user or encounters a breakpoint.

Reset

Debugger > Reset > Processor Reset will perform a full reset of the target microcontroller, so execution can begin again from the start of the program code. This is only available when the target is halted.

Halt the demo board PIC18F46K20 if it is currently running, and select *Debugger > Reset > Processor Reset* <F6> This will open up a new file in the MPLAB IDE called `c018i.c`. This is the Start-Up Code, part of the MPLAB C18 library. This library code initializes the C software stack, assigns appropriate data values to any initialized data variables, and jumps to the start of the application function `main()`.

FIGURE 3-29: C018 START-UP LIBRARY CODE



C:\MCC18\src\traditional\startup\c018i.c

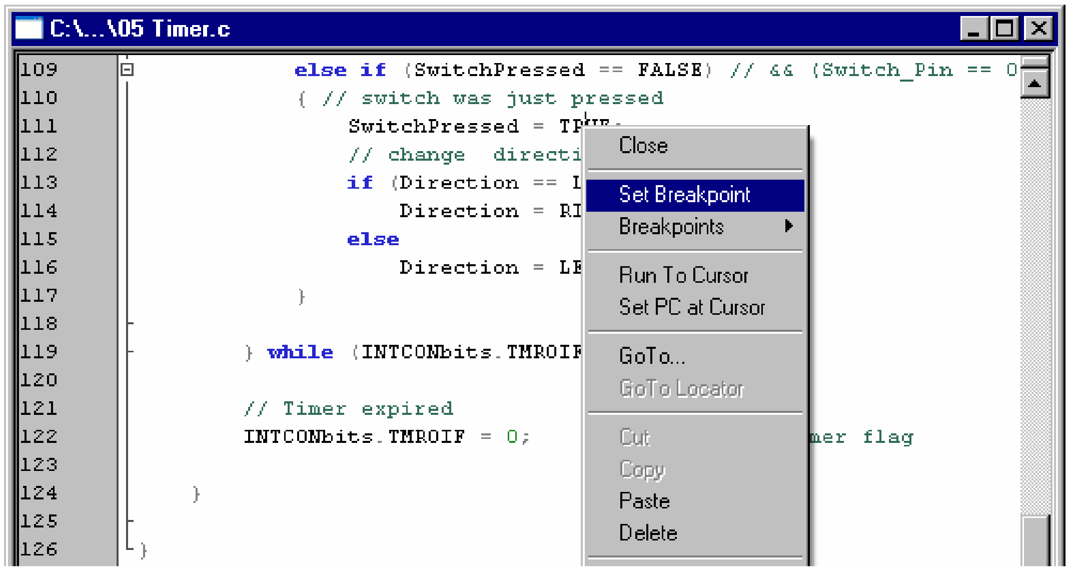
```
/* $Id: c018i.c,v 1.7 2006/11/15 22:53:12 moshtaa Exp $ */

/* Copyright (c)1999 Microchip Technology */

/* MPLAB-C18 startup code, including initialized data */

/* external reference to __init() function */
extern void __init (void);
/* external reference to the user's main routine */
extern void main (void);
/* prototype for the startup function */
void _entry (void);
void _startup (void);
/* prototype for the initialized data setup */
void _do_cinit (void);

extern volatile near unsigned long short TBLPTR;
extern near unsigned FSR0;
extern near char __FPFLAGS;
#define RND 6

#pragma code _entry_scn=0x000000
void
_entry (void)
{
_asm goto _startup _endasm

}
```

3.6.4 Using Breakpoints

When debugging code, a “breakpoint” can be added to a program statement. When running the program, the debugger will halt the target upon reaching the breakpoint statement.

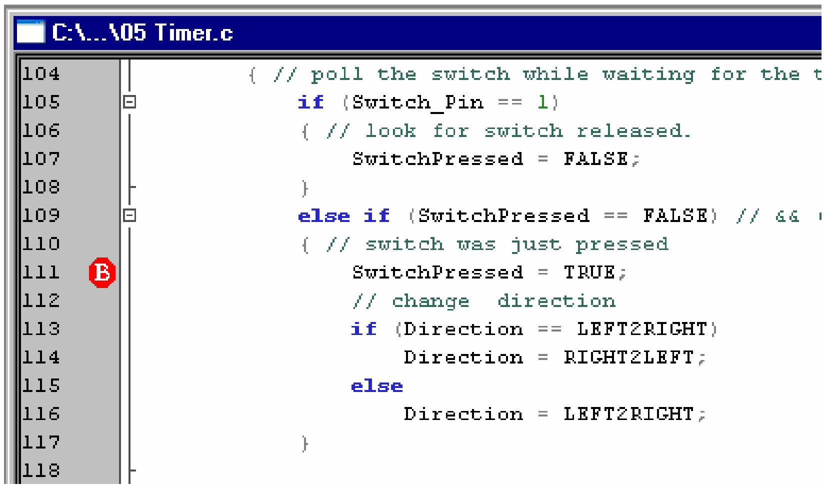
In the MPLAB IDE `05 Timer.c` source code, place the editor cursor on line 111, `SwitchPressed = TRUE;`, and right-click to open the contextual menu. Select *Set Breakpoint* as shown in Figure 3-30. A red octagon with the letter 'B' will appear in the editor margin to indicate a breakpoint has been set on that line.

FIGURE 3-30: SET BREAKPOINT ON LINE 111



```
109              else if (SwitchPressed == FALSE) // && (Switch_Pin == 0
110              { // switch was just pressed
111                  SwitchPressed = TRUE;
112                  // change  directi
113                  if (Direction == L
114                      Direction = RI
115                  else
116                      Direction = LE
117              }
118
119          } while (INTCONbits.TMROIF
120
121          // Timer expired
122          INTCONbits.TMROIF = 0;                       mer flag
123
124      }
125
126  }
```

FIGURE 3-30: BREAKPOINT SET

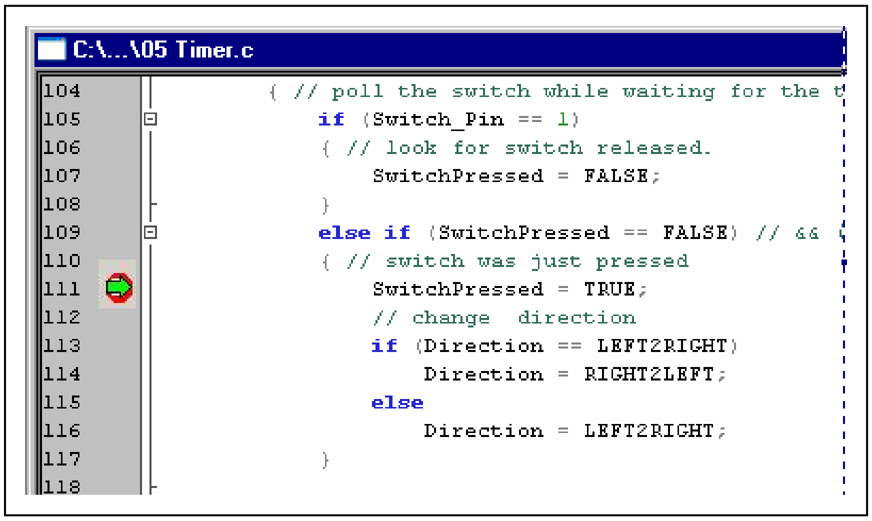


```
104          { // poll the switch while waiting for the t
105              if (Switch_Pin == 1)
106              { // look for switch released.
107                  SwitchPressed = FALSE;
108              }
109              else if (SwitchPressed == FALSE) // &&
110              { // switch was just pressed
111                  SwitchPressed = TRUE;
112                  // change  direction
113                  if (Direction == LEFT2RIGHT)
114                      Direction = RIGHT2LEFT;
115                  else
116                      Direction = LEFT2RIGHT;
117              }
118
```

The statement we've placed the breakpoint on will be executed when the demo board switch button is pressed. Select *Debugger > Run* to begin program execution. The demo board LEDs will rotate as the code runs since the breakpoint statement has been executed yet.

Press the demo board switch button. The program will halt on the breakpoint statement, as shown in Figure 3-31. `<F8>` can now be used to step through the code.

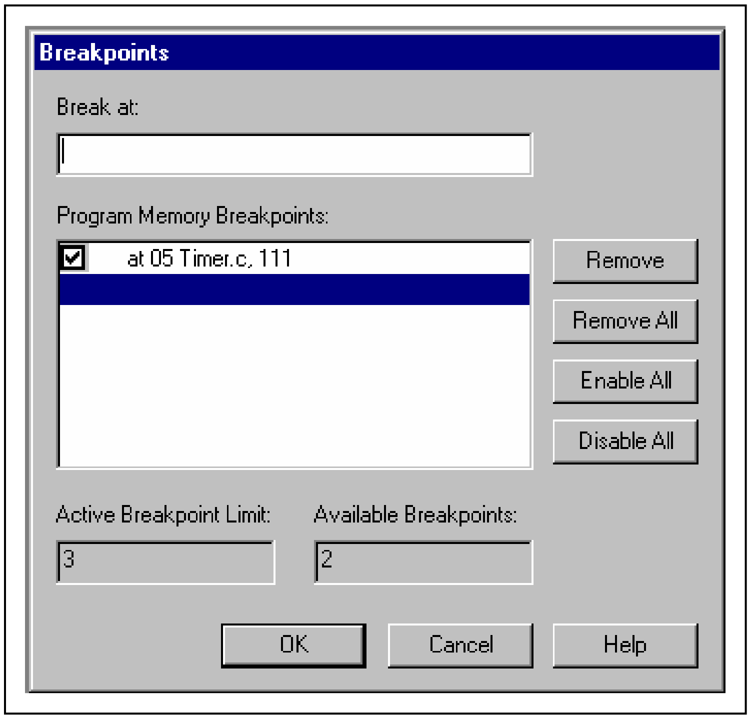
FIGURE 3-31: BREAKPOINT HALT



```
104             { // poll the switch while waiting for the t
105                 if (Switch_Pin == 1)
106                 { // look for switch released.
107                     SwitchPressed = FALSE;
108                 }
109                 else if (SwitchPressed == FALSE) // && (
110                 { // switch was just pressed
111                     SwitchPressed = TRUE;
112                     // change  direction
113                     if (Direction == LEFT2RIGHT)
114                         Direction = RIGHT2LEFT;
115                     else
116                         Direction = LEFT2RIGHT;
117                 }
118
```

The number of breakpoints that can be set at once in a program depends on the PIC18FXXXX device being debugged. Select menu _Debugger > Breakpoints…_. This will open a dialogue box to show the currently set breakpoints, the total number available in "Active Breakpoint Limit:" and the number of unused breakpoints that are still available as "Available Breakpoints:". The PIC18F46K20 can have up to 3 breakpoints set at once, and has 2 currently available since one is already set on line 111 of `05 Timer.c`.

FIGURE 3-32: BREAKPOINTS DIALOGUE



- [✓] at 05 Timer.c, 111

Active Breakpoint Limit: 3

Available Breakpoints: 2

Note: The number of active breakpoints can affect using the *Step Into* and *Step Over* functions. When these functions are used, a breakpoint is set at the next statement to step to. If all breakpoints are currently used and none are available, the MPLAB IDE is not able to set a breakpoint on the next C statement. Instead, it must step through each assembly instruction until the next statement is reached. If using *Step Over*, it may take some time to step over all the assembly functions in the compiled function. Free up a breakpoint to avoid this issue.

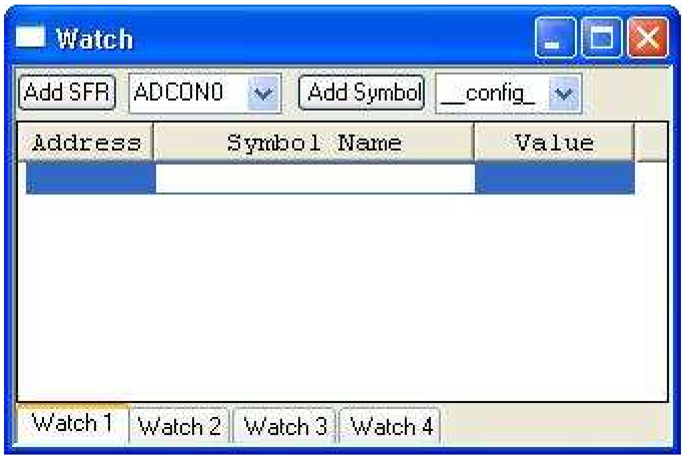
3.6.5 Watching Variables and Special Function Registers.

All the values in the File Registers can be seen by opening *View > File Registers*, and the values in the Special Function Registers can be seen by opening *View > Special Function Registers*. However, keeping these windows open is not recommend. This is because the entire file memory and all SFRs must be read from the target device whenever it is Run, Halted, and on each Step. Reading all of this data over the ICD bus can take a significant amount of time. The actual time it takes depends on how much memory the target PIC18FXXXX has, and how fast the target oscillator is. The slower the target oscillator, the longer it will take as the oscillator speed directly affects the ICD bus speed.

If you have opened either of these windows, please close them now.

The best way to watch variables and SFRs is to use a Watch Window. This way, only the variables and registers that are of interest are updated. To open a Watch Window, select *View > Watch*.

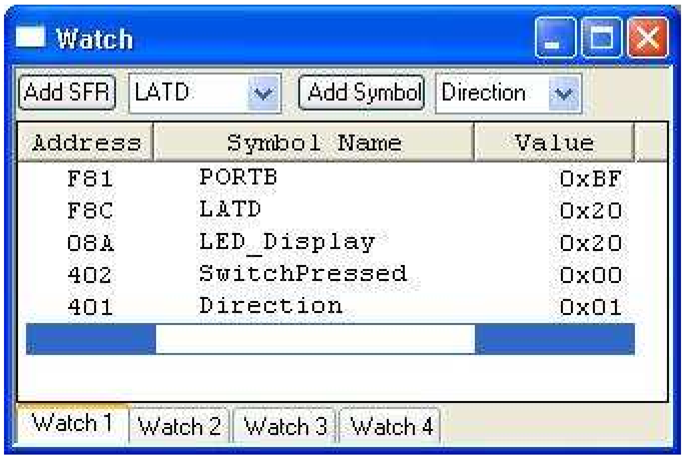
FIGURE 3-33: WATCH WINDOW



SFRs may be added to the watch window by selecting them in the dropdown box on the upper left, and clicking the **Add SFR** button. Go ahead and add PORTB, which used to read the switch state, and LATD, which our program uses to set the LEDs.

User variables are added using the dropdown on the upper right, and clicking the **Add Symbol** button. Add the `LED_Display`, `SwitchPressed`, and `Direction` variables now.

FIGURE 3-34: WATCH VARIABLES



Address	Symbol Name	Value
F81	PORTB	0xBF
F8C	LATD	0x20
08A	LED_Display	0x20
402	SwitchPressed	0x00
401	Direction	0x01

Note: The "Value" fields in the Watch Window, File Register Window, and Special Function Register windows may not be valid immediately after first being opened. Step the code once to update the values.

For each watch variable, the Watch Window displays the File Register Address, the Symbol Name (variable name), and current Value. The value display format can be changed by right-clicking on a value and selecting *Properties* from the pop-up menu. Note that our two enumerated type variables, `SwitchPressed` and `Direction` will display the enumeration value, and not the mnemonic.

The Watch Window can also be used to edit variable values. Select the LATD value by clicking on it, and type in the hex value 'AA'. Press enter to set the value. Look at the demo board; note that every other LED is now turned on. This is because through the Watch Window, we just directly wrote to the LATD register the value 0xAA, which is binary 0b10101010!

Select the PORTB symbol, right-click and select *Properties*. In the properties dialogue, go to the dropdown box for "Format:" and select "Binary". Click **OK** to close the dialogue. The PORTB value is now displayed in a binary format, with bit 7 on the left.

Step through the code once using `<F8>`. Note the value for PORTB bit 4, which is pin RB4 and connected to the demo board switch. The bit value should now be set ('1'). While pressing down the demo board button, step again with `<F8>`. Note that PORTB bit 4 is now low since the switch is pressed!

Take some time to play with the lesson code, stepping through it and watching variables and the demo board LEDs. You can also press the button and step through the switch detection statements. Set different breakpoints to experiment using them.

Add TMR0L and TMR0H SFRs to the watch window, and observe them counting while you step through the code. Note that they don't increment once per step, as each C statement may be compiled into more than one assembly instruction and Timer0 is incremented once per assembly (machine) instruction.

3.7 Lesson 7: Analog-to-Digital Converter (ADC)

Lesson 7 builds on the previous lesson by using the on-chip ADC to read the demo potentiometer voltage. The result is used to vary the LED rotation time delay so that the potentiometer controls the LED rotation speed.

Key Concepts

- An Analog-to-Digital Converter is used to convert an analog voltage level into a digital number representing the voltage.
- The ANSEL, ANSELH, ADCON0, ADCON1, & ADCON2 SRFs configure and control the on-chip ADC.
- A timer register(s) can be written to set the amount of time until it overflows without changing the

3.7.1 PIC18F46K20 ADC Basics

Simply put, an ADC takes the ratio of an input voltage to a reference voltage and represents it as a number. This number is dependent on the bits of resolution of the ADC. For example, the 10-bit resolution of the PIC18F46K20 ADC means that 1024 numbers from 0 – 1023 are available to represent the voltage ratio. In mathematical terms,

ADC Value = (Vin / Vref) * 1023

If Vin = 2.5Volts, and Vref = 5.0Volts, then the ADC Value is (2.5/5)*1023 = 511. This makes sense in that Vin is half of Vref, so the ADC value is half of 1023.

Knowing the reference voltage and solving the equation for Vin allows the ADC Value to be converted back into a voltage:

Vin = (ADC Value / 1023) * Vref

The PIC18F46K20 ADC may be referenced to the device VDD voltage or an external voltage reference. In this lesson, the ADC is referenced to the PIC18F46K20 Starter Kit Demo Board VDD, which is supplied by PICkit 2. This voltage is typically around 3.3V for this device.

The ADC can convert the voltage from any one of 13 channels on the PIC18F46K20. These analog input channels, numbered AN0 up to AN12, are shared with digital microcontroller pins and must be configured as analog inputs to be used with the ADC.

The ADC is configured and controlled by 5 Special Function Registers: ANSEL, ANSELH, ADCON0, ADCON1, and ADCON2. These are covered in detail in the next section.

3.7.2 ADC Configuration and Operation

Looking at the schematic of the PIC18F46K20 Starter Kit Demo board in the Appendix, the potentiometer (RP1) output is connected to the RE0/AN5 pin of the PIC18F46K20.

The basic steps needed to convert the ADC voltage on this pin are:

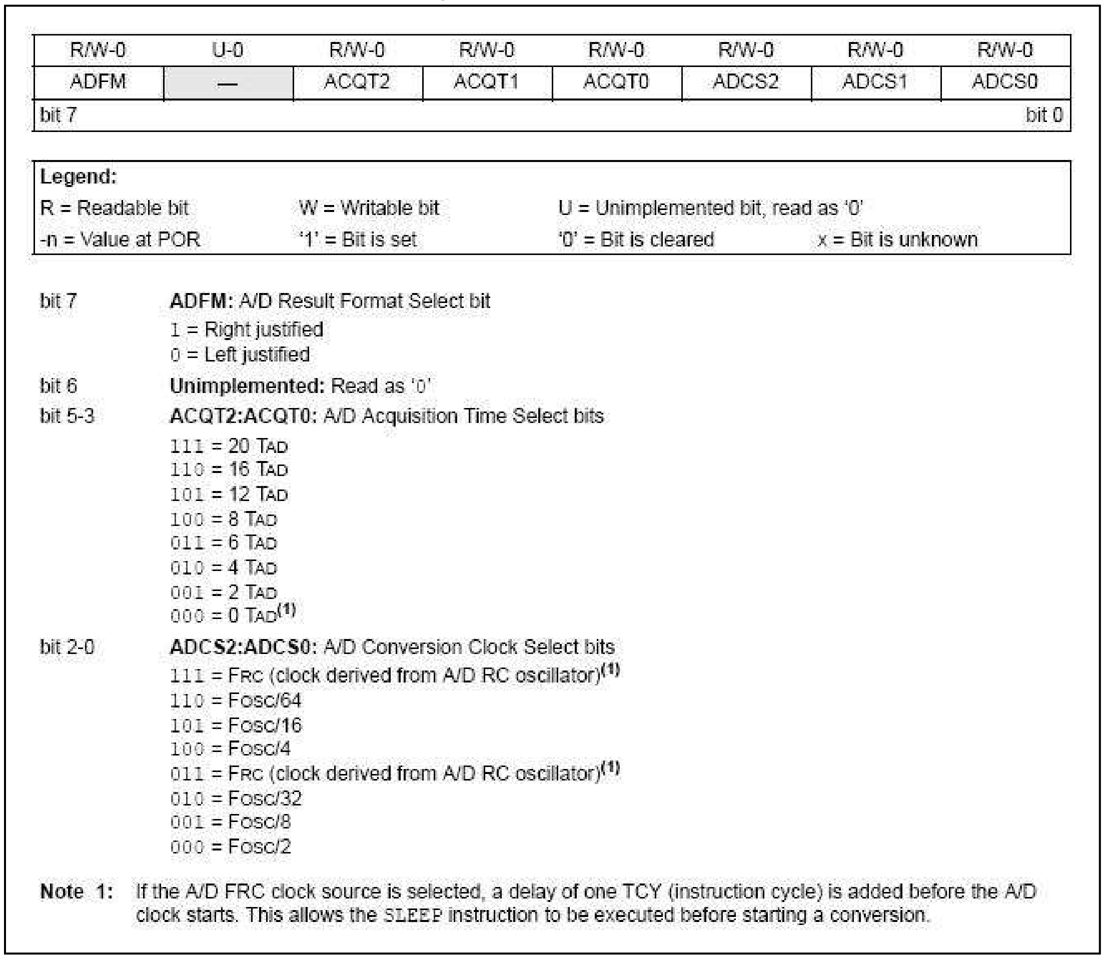
1. Configure the RE0/AN5 pin as an analog input in ANSEL.
2. Set the ADC voltage references in ADCON1.
3. Set the result justification, ADC clock source, and acquisition time in ADCON2.
4. Select the channel and turn on the ADC in ADCON0.
5. Start the conversion in ADCON0.

#1: To use a pin as an analog input, it must not be used by other peripheral functions multiplexed on the same pin. The pin TRIS bit must be set to '1' (input) and the ANSEL bit associated with RE0 should be set to '1' (analog input). However, we still want RB0/AN12 configured as a Digital input to for the switch. Therefore, we will clear '0' the AN12 bit in ANSELH.

#2: The VCFGx bits in ADCON1 can select the ADC voltage references to use the AN2 and AN3 pins, VDD and VSS, or some combination. Since the demo board does not have voltage references connected to AN2 and AN3, the ADC will be referenced to VDD and VSS. This means an ADC result of '0' corresponds to 0 Volts, or VSS. A result of '1023' corresponds to about 3.3 Volts, or VDD. Including the values from #1, the ADCON1 setting for this lesson is

```
ADCON1 = 0;
```

FIGURE 3-35: ADCON2: A/D CONTROL REGISTER 2



R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
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 **ADFM:** A/D Result Format Select bit
1 = Right justified
0 = Left justified

bit 6 **Unimplemented:** Read as '0'

bit 5-3 **ACQT2:ACQT0:** A/D Acquisition Time Select bits
111 = 20 TAD
110 = 16 TAD
101 = 12 TAD
100 = 8 TAD
011 = 6 TAD
010 = 4 TAD
001 = 2 TAD
000 = 0 TAD(1)

bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
111 = FRC (clock derived from A/D RC oscillator)(1)
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = FRC (clock derived from A/D RC oscillator)(1)
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2

Note 1: If the A/D FRC clock source is selected, a delay of one TCY (instruction cycle) is added before the A/D clock starts. This allows the SLEEP instruction to be executed before starting a conversion.

#3: The ADC clock should be set as short as possible but still greater than the minimum period "TAD" time, datasheet parameter 130. The minimum TAD time for the PIC1846K20 (as of this writing) is

1.4us. At a 1 MHz oscillator Fosc, selecting bits ADCS = Fosc/2 gives a 500kHz ADC clock. One clock period 1/ 500kHz = 2us, which is greater than the minimum TAD = 1.4us. Thus ADCSx = ‘000’.

The ACTQx bits determine the acquisition time, and should take into account the internal acquisition time Tacq of the ADC, datasheet parameter 132, and the settling time of the application circuit connected to the ADC pin. From the datasheet, the internal acquisition time Tacq = 1.4us over temperature. The application circuit is an RC network formed by the potentiometer and capacitor C3, which has a very long settling time. For this demo lesson, we’ll simply set ACQTx to the largest value, 20TAD or ‘111’. 20 TAD is 20 times the ADC Clock period, or 20 * 2us = 40us.

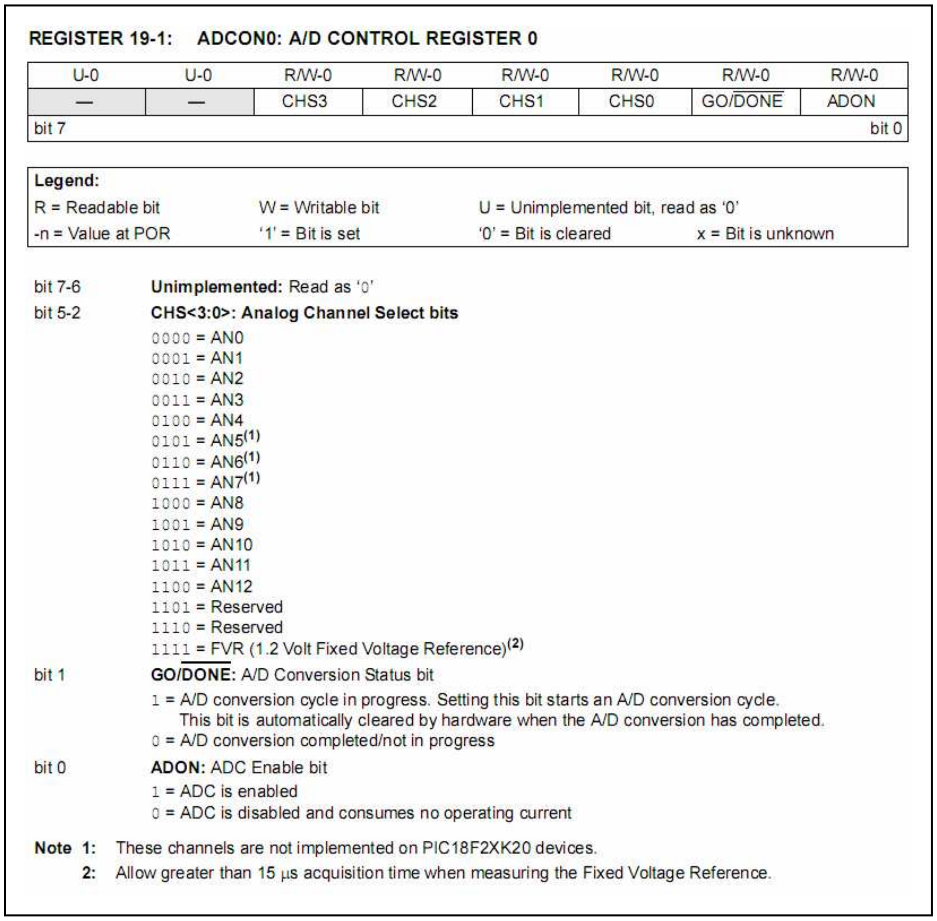
For result justification, we choose bit ADFM = 0 to the result is left-justified. This makes it easy to get the 8 most significant bits of the result from ADRESH. Thus the ADCON2 configuration value is

ADCON2 = 0b00111000

#4: The demo board potentiometer is connected to AN5, so Channel 5 is selected in ADCON0. Bit ADON is set to ‘1’ to turn on the ADC peripheral. The GO/DONE bit is left clear as we don’t wish to start a conversion yet.

ADCON0 = 0b00010101

FIGURE 3-36: ADCON0: A/D CONTROL REGISTER 0



REGISTER 19-1: ADCON0: A/D CONTROL REGISTER 0

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
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 7-6 **Unimplemented:** Read as ‘0’

bit 5-2 **CHS<3:0>: Analog Channel Select bits**

0000 = AN0
0001 = AN1
0010 = AN2
0011 = AN3
0100 = AN4
0101 = AN5(1)
0110 = AN6(1)
0111 = AN7(1)
1000 = AN8
1001 = AN9
1010 = AN10
1011 = AN11
1100 = AN12
1101 = Reserved
1110 = Reserved
1111 = FVR (1.2 Volt Fixed Voltage Reference)(2)

bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit

1 = A/D conversion cycle in progress. Setting this bit starts an A/D conversion cycle.
This bit is automatically cleared by hardware when the A/D conversion has completed.
0 = A/D conversion completed/not in progress

bit 0 **ADON:** ADC Enable bit

1 = ADC is enabled
0 = ADC is disabled and consumes no operating current

Note 1: These channels are not implemented on PIC18F2XK20 devices.
2: Allow greater than 15 μs acquisition time when measuring the Fixed Voltage Reference.

#5: To begin an ADC conversion, set bit 1 of ADCON0, the GO/DONE bit. When the conversion is done the hardware will clear that bit, so the GO/DONE may then be polled to wait for the conversion to complete. Once the conversion is complete and GO/DONE = 0, the ADC conversion result may be read from ADRESH and ADRESL.

3.7.3 Exploring the Lesson 7 Source Code

Open the lesson source files `07 ADC.c` and `07 ADC.h` in an MPLAB editor window if they are not already open.

Of note is that the Timer0 setup code has been moved into a function and replaced with a function call. Two new functions were added to support the ADC.

```
void Timer0_Init(void)
void ADC_Init(void)
unsigned char ADC_Convert(void)
```

The function prototypes have also been added to the header file, `07 ADC.h`.

In `main()` before getting to the `while(1)` loop, the program makes two function calls to set up the Timer0 and ADC peripherals using `Timer0_Init()` and `ADC_Init()` respectively.

To change the LED rotation speed based on the potentiometer, the ADC conversion value is used to set Timer0 just after it overflows. The higher the value written to Timer0, the less time it takes to overflow again, as the timer counts up from the written value. This is accomplished with two new statements at the bottom of the `while(1)` loop:

```
TMR0H = ADC_Convert();      // MSB from ADC
TMR0L = 0;                  // LSB = 0
```

The TMR0H buffer is written with the 8 most significant bits of the ADC conversion, and then is written with Timer0 with a ‘0’ in the low byte on the TMR0L assignment statement. Recall from lesson 5 that since TMR0H is actually a buffer and not the upper byte of the timer, and is written to the timer when TMR0L is written. Thus, it must be written first as it is here.

We can calculate the amount of delay for a given ADC value, knowing that `Timer0_Init()` sets the TMR0 prescaler to 1:4, and our Oscillator is 1MHz. Timer 0 will count at 4 * the instruction rate, or 4 * 1/(Fosc/4) = 4 * 1/(1MHz/4) = 4 * 1/250kHz = 16us. The number of counts until overflow occurs is 0x10000 – (start count) where (start count) is the value written to TMR0 – The ADC result in the upper byte and 0x00 in the lower. The total delay is then the number of counts times the count rate. For an ADC result of 0x81, the delay is (0x10000 – 0x8100) * 16 us = 0x7F00 * 16us = 32512 * 16us = 0.52 seconds.

3.7.4 Build and Run the Lesson 7 Code with PICkit 2 Debug Express

Build and program the lesson 7 project, then Run the application in the debugger. Turning the demo board potentiometer will affect the rotation speed of the LEDs. The switch may be pressed to reverse the rotation.

Halt the lesson 7 program. Note that several SFRs and variables have already been added to a Watch Window. Use Breakpoints and Step commands to explore the code. Observe how the ADC result in ADRESH is affected by the potentiometer voltage, and how this result is copied into TMR0.

See section 19.0 10-Bit Analog-to-Digital Converter (A/D) Module in the PIC18F46K20 for more information on the ADC peripheral.

Note: If TMR0L is added to the Watch Window, it will cause incorrect operation when stepping through the following 2 lines of code:

```
TMR0H = ADC_Convert();
TMR0L = 0;
```

This is caused by the buffered nature of TMR0H. When “Stepping Over” the TMR0H assignment statement, the MPLAB IDE will read the TMR0L register to update the value in the Watch Window. When TMR0L is read, the upper byte of TMR0 is loaded into the TMR0H buffer, wiping out the value written in the previous TMR0H assignment statement.

One workaround to be able to add TMR0L to the Watch Window is to make sure not to step from the TMR0H to the TMR0L statement. Set a breakpoint on the TMR0L assignment statement, and Run from the TMR0H assignment statement.

3.8 Lesson 8: Interrupts

This lesson changes the lesson 7 code to use interrupts to act on the switch press and Timer0 events instead of polling them. The switch uses the RB0/INT0 external interrupt capability.

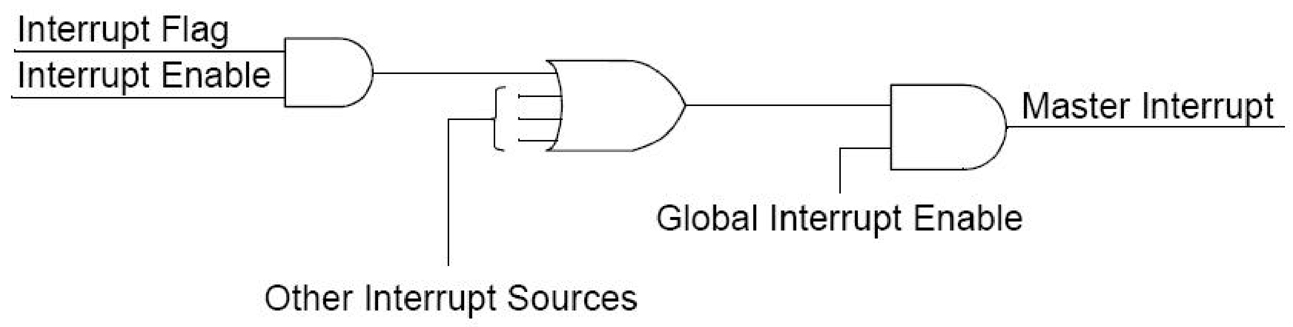
Key Concepts

- An interrupt is a hardware based event that "interrupts" the program code to execute a special function. When the interrupt function exits, program execution returns to where it left off.
- The PIC18FXXXX supports a single interrupt priority or two levels of interrupt priority.
- A Low Priority interrupt can interrupt the main program. A High Priority interrupt can interrupt the main program or a low priority interrupt.
- The directives `#pragma interruptlow` and `#pragma interrupt` are used to define the interrupt functions.

3.8.1 PIC18FXXXX Interrupt Architecture

When a peripheral requires attention or an event occurs, it sets an interrupt flag. Each flag has an interrupt enable bit that determines whether it will generate an interrupt to the microcontroller or not. In the previous lessons, interrupt flags such as TMR0IF were polled, but did not create an interrupt as the enable bit was not set. The enable bits allow only selected events to cause in interrupt. All interrupts are ORed together, and then ANDed with a global interrupt enable.

FIGURE 3-37: SIMPLIFIED INTERRUPT LOGIC



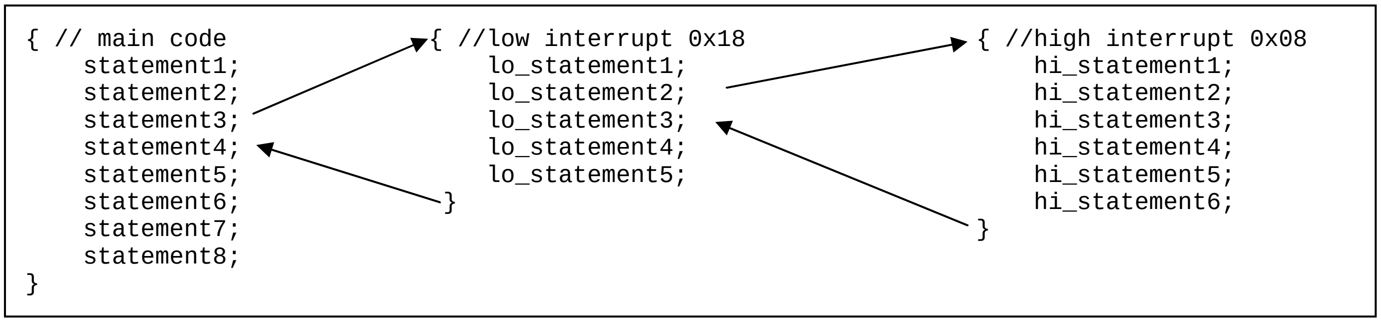
When an interrupt occurs and the Master Interrupt signal is asserted, the PIC microcontroller finishes executing the current instruction, stores the next address on the Return Address Stack, and then jumps to an interrupt vector. At the interrupt vector it begins executing a function designated as the interrupt service routine. When this function exits, program execution returns to the address stored on the Return Address Stack.

Interrupts allow hardware events to be acted upon very quickly and regardless of the state of the main program because they cause the immediate execution of dedicated code.

The PIC18FXXXX architecture supports up to two levels of interrupt priority, each of which have a logic structure like that in Figure 3-37. Most interrupts have a Priority bit associated with the interrupt flag and enable that assigns it to one of the two priority levels. Using priority levels is optional, and the PIC18FXXXX may be configured to use only one level priority.

When two levels of interrupt priority are used, an interrupt of either priority level may interrupt the main program. However, only a High Priority Interrupt may interrupt a Low Priority Interrupt, and nothing may interrupt a High priority Interrupt. As shown in Figure 3-38, when a low priority interrupt event occurs during execution of `statement3` in the main code, the program jumps to begin executing the Low Priority Interrupt function. During execution of the `lo_statement2`, a high priority interrupt event occurs, causing program execution to jump to the High Priority Interrupt function. When the high priority function completes and exits, execution is returned to where it left off in the low priority function. Similarly, when the low priority function completes and exits, program execution returns to where it left off in the main code, at `statement4`.

FIGURE 3-38: PRIORITY INTERRUPT EXECUTION FLOW



```
{ // main code              { //low interrupt 0x18          { //high interrupt 0x08
    statement1;                 lo_statement1;                  hi_statement1;
    statement2;                 lo_statement2;                  hi_statement2;
    statement3;                 lo_statement3;                  hi_statement3;
    statement4;                 lo_statement4;                  hi_statement4;
    statement5;                 lo_statement5;                  hi_statement5;
    statement6;               }                                 hi_statement6;
    statement7;                                               }
    statement8;
}
```

The High Priority Interrupt Vector is at Program Memory address 0x0008. The Low Priority Interrupt Vector is at Program Memory address 0x0018. If interrupt priorities are not used, all interrupts jump to the high priority vector at 0x0008.

3.8.2 Exploring the Lesson 8 Source Code

The first thing to note is that the `Directions` variable is now global, so it may be accessed in the interrupt service routine functions.

When using interrupts, the interrupt vectors must be defined and placed at the appropriate vector addresses using the `#pragma code` directives. An inline assembly `GOTO` statement redirects program execution to the interrupt functions, whose name serves as the `GOTO` argument.

FIGURE 3-39: DEFINE INTERRUPT VECTORS

```
/** I N T E R R U P T S ***********************************************/

//----------------------------------------------------------------------------
// High priority interrupt vector

#pragma code InterruptVectorHigh = 0x08
void InterruptVectorHigh (void)
{
  _asm
    goto InterruptServiceHigh //jump to interrupt routine
  _endasm
}

//----------------------------------------------------------------------------
// Low priority interrupt vector

#pragma code InterruptVectorLow = 0x18
void InterruptVectorLow (void)
{
  _asm
    goto InterruptServiceLow //jump to interrupt routine
  _endasm
}
```

The interrupt service routine functions themselves are then declared with the `#pragma interrupt` directive for the high priority vector, and `#pragma interruptlow` for the low priority. Note the names must match between the vector GOTO argument, the `#pragma` attribute, and the function declaration name. The interrupt functions may call other functions defined elsewhere in the source, though the lesson source code does not do this.

FIGURE 3-40: INTERRUPT SERVICE FUNCTIONS

```
// -------------------- Iterrupt Service Routines --------------------------
#pragma interrupt InterruptServiceHigh    // "interrupt" pragma for high priority
void InterruptServiceHigh(void)
{
    // function statements

}  // return from high-priority interrupt

#pragma interruptlow InterruptServiceLow  // "interruptlow" pragma for low priority
void InterruptServiceLow(void)
{
    // function statements

}  // return from low-priority interrupt
```

As all interrupts of the same priority vector to the same function, it is necessary in the function to examine which of the enabled interrupt flags caused the interrupt. Once the flag is found so that peripheral or event may be serviced, the software must clear the interrupt flag bit to reset the interrupt. In the lesson source code, the high priority interrupt routine looks for the INT0 pin interrupt INT0IF flag bit. Examples are shown in the source code of how it might check for other enabled interrupts, such as Timer1 TMR1IF and the ADC ADIF although neither of these interrupts are enabled in the lesson code. Similarly, the low priority vector checks for the Timer0 flag TMR0IF.

Setting Up Interrupts

Now that the source code has defined the interrupt vectors, and has functions to deal with the interrupts, it must properly setup and configure the interrupting logic and enable the individual interrupts it wants to use.

Timer0 and external pin interrupts are set up using the INTCONx special function registers. Other interrupts are setup through a number set of peripheral interrupt SFRs: PIRx, PIEx, and IPRx. The PIRx registers contain the interrupt flags. The associated interrupt enable bits are in the PIEx registers, and the IPRx register bits set the interrupt priority as low or high. For detailed information the bits in these registers, see Section 9.0 Interrupts of the PIC18F46K20 Datasheet.

FIGURE 3-41: LESSON 8 INTERRUPT INITIALIZATIONS

```
    // Set up switch interrupt on INT0
    INTCON2bits.INTEDG0 = 0;    // interrupt on falling edge of INT0 (switch pressed)
    INTCONbits.INT0IF = 0;      // ensure flag is cleared
    INTCONbits.INT0IE = 1;      // enable INT0 interrupt
    // NOTE: INT0 is ALWAYS a high priority interrupt

    // Set up global interrupts
    RCONbits.IPEN = 1;          // Enable priority levels on interrupts
    INTCONbits.GIEL = 1;        // Low priority interrupts allowed
    INTCONbits.GIEH = 1;        // Interrupting enabled.

void Timer0_Init(void)
{
    // Set up Interrupts for timer
    INTCONbits.TMR0IF = 0;          // clear roll-over interrupt flag
    INTCON2bits.TMR0IP = 0;         // Timer0 is low priority interrupt
    INTCONbits.TMR0IE = 1;          // enable the Timer0 interrupt.
```

An interrupt is desired when the demo board button is pressed. Therefore, the program utilizes the INT0 functionality of the RB0 pin to use it as an external interrupt input pin. The interrupt is edge triggered, and we want it to interrupt on the falling edge so the initial switch press is detected. The edge direction is set with `INTCON2bits.INTEDG0`. INT0 is always a high priority interrupt. The flag INT0IF in INTCON is cleared before enabling the interrupt with INT0IE. Switch debouncing is ignored for the sake of simplicity here, but would be recommended in a product application.

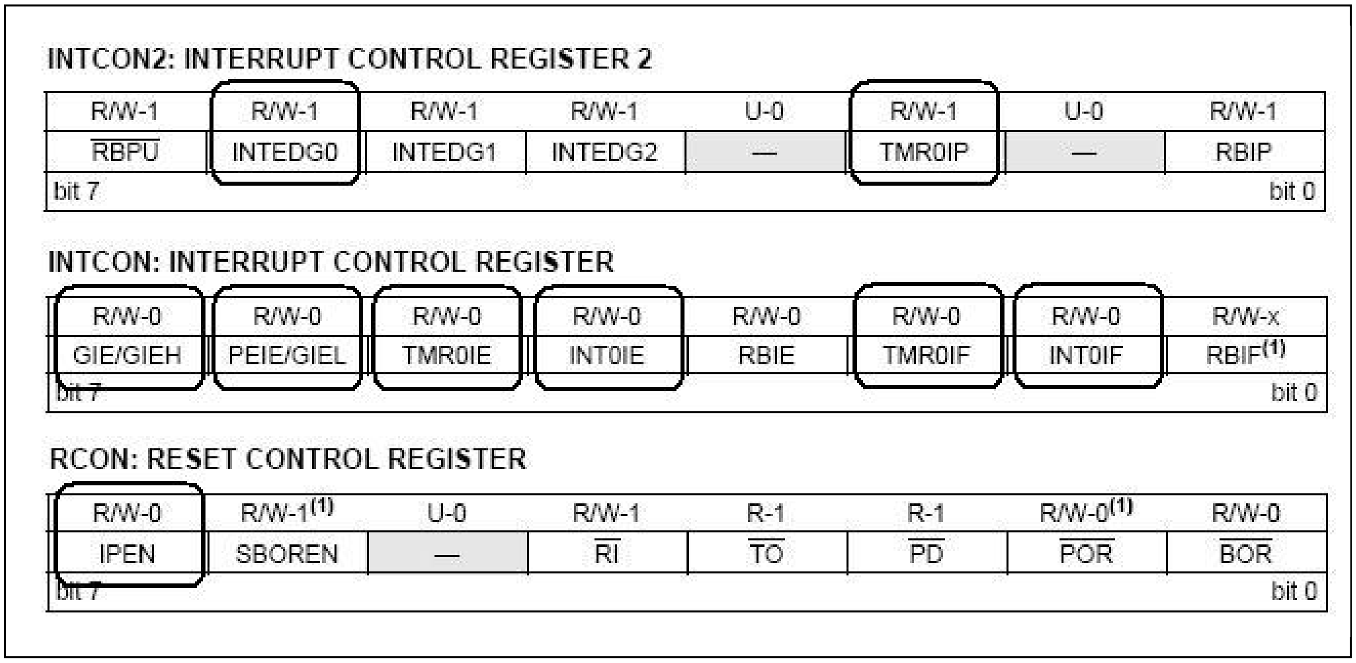
The interrupt configuration for Timer0 has been added to the `Timer0_Init()` function. First, we make sure the flag TMR0IF is cleared, set the priority to low (0) with TMR0IP, and then enable the interrupt with TMR0IE.

Enabling the individual interrupts has no effect until interrupts are enabled at the global level. First, the IPEN bit in RCON is used to enable or disable priority interrupts. In lesson 8 it is set to enable priority interrupts. Low priority interrupts are enabled with GIEL, and microcontroller interrupting is enabled with GIEH. Note that high and low priority interrupts aren't individually enabled with the two bits, as GIEH shuts off both when it is off:

`INTCONbits.GIEH`	`INTCONbits.GIEL`	Interrupt Functions
0	0	No Interrupts; all interrupts disabled.
0	1	No Interrupts; all interrupts disabled.
1	0	High priority interrupts only enabled.
1	1	Both priority level interrupts enabled

In this way, all interrupts may disabled with a single bit, GIEH in INTCON.

FIGURE 3-42: LESSON 8 INTERRUPT SFRS



INTCON2: INTERRUPT CONTROL REGISTER 2

R/W-1	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
$\overline{RBPU}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP
bit 7							bit 0

INTCON: INTERRUPT CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF(1)
bit 7							bit 0

RCON: RESET CONTROL REGISTER

R/W-0	R/W-1(1)	U-0	R/W-1	R-1	R-1	R/W-0(1)	R/W-0
IPEN	SBOREN	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

In the lesson 8 source code, all the statements to change the rotation direction are in the INT0 switch interrupt function, and the statements to rotate the LED display are in the TMR0 interrupt function. All that remains in the main program is a `while()` loop that updates the PORTD register with LED_Display. This statement could have also been placed in the TMR0 interrupt function, but is left in the main program to illustrate how the main program runs continuously and interacts with the interrupts.

Single Priority Interrupts
If only a single level of interrupts were used (RCON bit IPEN = 0), then it is only necessary to define the interrupt vector at 0x0008, and a single interrupt service routine function with `#pragma interrupt`. All priority bit settings are ignored. The function of the INTCON bits GIEH and GIEL become GIE and PEIE respectively, with the following functions:

`INTCONbits.GIE`	`INTCONbits.PIEIE`	Interrupt Functions
0	0	No Interrupts; all interrupts disabled.
0	1	No Interrupts; all interrupts disabled.
1	0	Only interrupts enabled in INTCONx enabled. All PIEx interrupts remain disabled.
1	1	All interrupts, including those enabled in PIEx registers, are enabled.

3.8.3 Build and Run the Lesson 8 Code with PICkit 2 Debug Express

Build and program the lesson 8 project, then Run the application in the debugger. Turning the demo board potentiometer will affect the rotation speed of the LEDs. The switch may be pressed to reverse the rotation. Use breakpoints to explore the interrupting functions.

3.9 Lesson 9: Internal Oscillator

Using the on-chip internal oscillator and PLL (Phase Locked Loop) of the PIC18F46K20 is discussed. Clocks from 31 kHz up to 64 MHz can be generated without requiring external oscillator components.

Key Concepts

- To use the internal oscillator block, set the OSC configuration bits to INTIO67 or INTIO7. The latter outputs the clock signal CLKO on the RA6 pin.
- The OSCCON Special Function Register is used to set the base internal oscillator frequency from 31 kHZ up to 16 MHz.
- The OSCTUNE register allows the internal oscillator frequency to be adjusted on a fine scale, and enables or disables the PLL.
- The 4x PLL may only be used when base frequencies of 8 MHz or 16 MHz are selected in OSCCON. Enabling the PLL multiplies the base frequency by 4, providing clocks at 32 MHz and 64 MHz, respectively.

3.9.1 The Internal Oscillator Block

The internal oscillator block of the PIC18F46K20 generates two different clock signals. The main output, INTOSC, is a factory calibrated 16 MHz clock source with postscaler that can provide a range of clock frequencies down to 31 kHz.

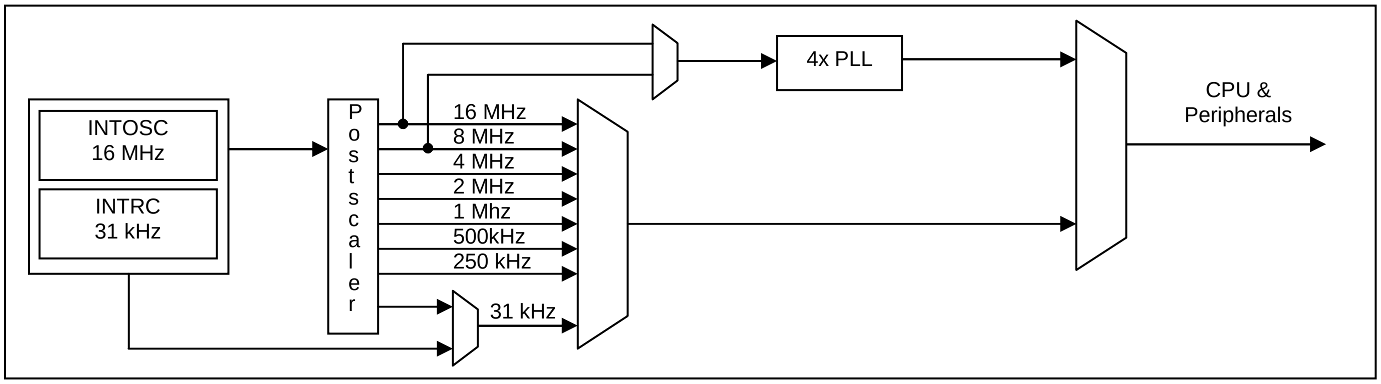
The other output, INTRC, is a nominal 31 kHz clock source that drives peripherals such as the Power-up Timer, the Fail-Safe Clock Monitor, the Watchdog Timer, and the Two-Speed Startup feature.

When the oscillator block is set to provide a 31 kHz clock to the microcontroller, it can be selected as a postscaled output of INTOSC, which has the benefit of calibrated accuracy, or INTRC, which has the benefit of lower power consumption.

The oscillator block also contains a 4x PLL (Phase Locked Loop) frequency multiplier that can increase the microcontroller clock source up to 32 MHz. The PLL is only available when the internal oscillator block selected output is 8 MHz or 16 MHz. It will multiply the base 4 MHz signal by 4 to 32 MHz, and the 8 MHz base clock to 64 MHz.

This allows the internal oscillator block to provide a range of 10 different, software selecteable frequencies of 31 kHz, 250 kHz, 500 kHz, 1 MHz, 2 MHz, 4MHz, 8 MHz, 16 MHz and (with the PLL) 32 MHz and 64 MHz. Recall from previous lessons that the default frequency on a reset is 1 MHz.

FIGURE 3-43: SIMPLIFIED INTERNAL OSCILLATOR BLOCK DIAGRAM

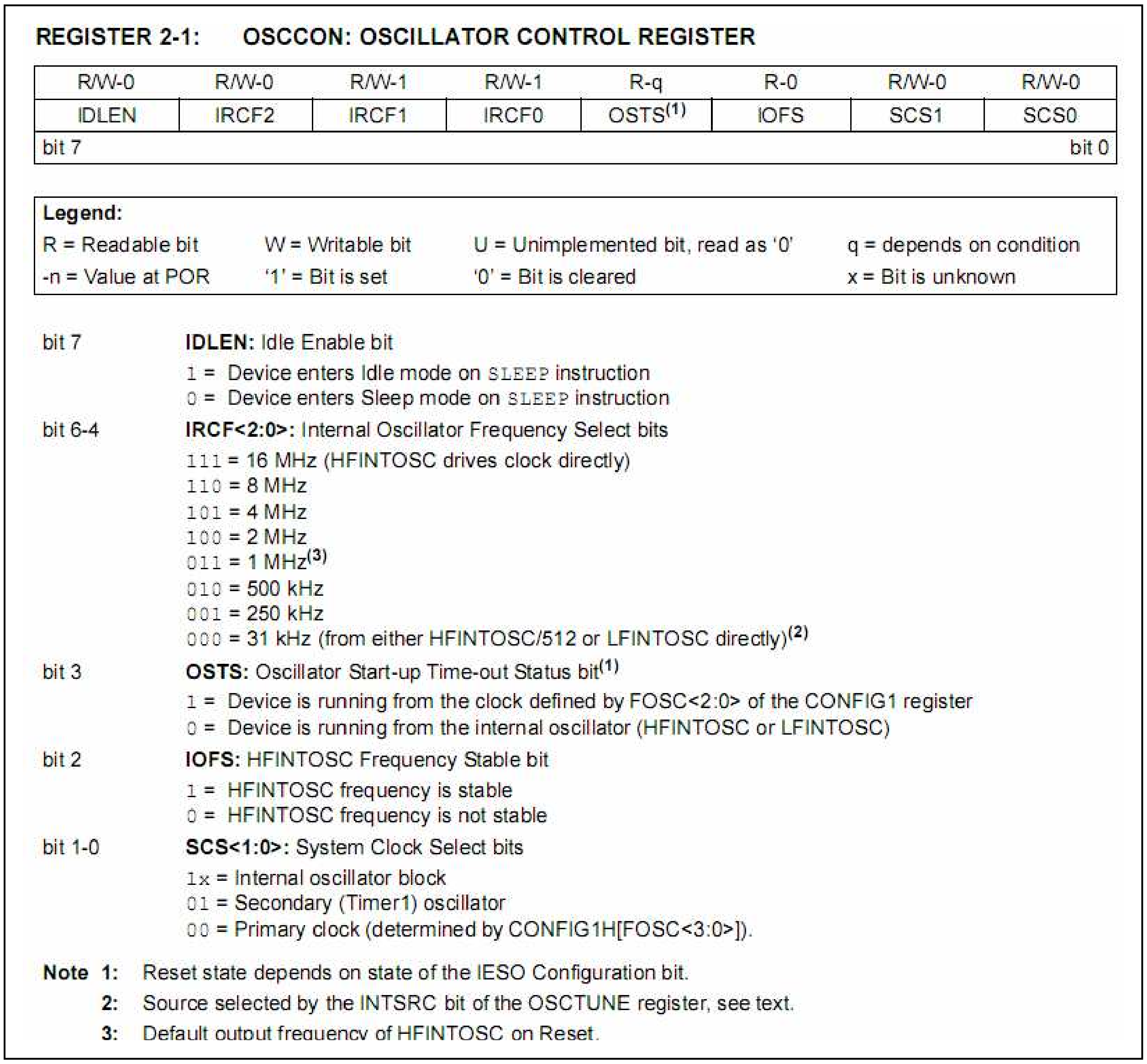


3.9.2 Configuring the Internal Oscillator

The internal oscillator block is selected as the primary oscillator in the Configuration bits. The OSC bits in the CONFIG1H configuration word are set to either INTIO67 or INTIO7. When INTIO67 is selected, the internal oscillator is the primary oscillator with the external oscillator pins OSC2 & OSC1 available as RA6 & RA7 IO. OSC = INTIO7 differs only in that RA6 is not available; instead the internal instruction clock is output as CLKO on that pin.

The two Special Function Registers that control the internal oscillator block in software are OSCCON and OSCTUNE, shown in figures 3-44 and 3-45.

FIGURE 3-44: OSCCON: OSCILLATOR CONTROL REGISTER



REGISTER 2-1: OSCCON: OSCILLATOR CONTROL REGISTER

R/W-0	R/W-0	R/W-1	R/W-1	R-q	R-0	R/W-0	R/W-0
IDLEN	IRCF2	IRCF1	IRCF0	OSTS[1]	IOFS	SCS1	SCS0
bit 7							bit 0

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

bit 7 **IDLEN:** Idle Enable bit
1 = Device enters Idle mode on `SLEEP` instruction
0 = Device enters Sleep mode on `SLEEP` instruction

bit 6-4 **IRCF<2:0>:** Internal Oscillator Frequency Select bits
111 = 16 MHz (HFINTOSC drives clock directly)
110 = 8 MHz
101 = 4 MHz
100 = 2 MHz
011 = 1 MHz[3]
010 = 500 kHz
001 = 250 kHz
000 = 31 kHz (from either HFINTOSC/512 or LFINTOSC directly)[2]

bit 3 **OSTS:** Oscillator Start-up Time-out Status bit[1]
1 = Device is running from the clock defined by FOSC<2:0> of the CONFIG1 register
0 = Device is running from the internal oscillator (HFINTOSC or LFINTOSC)

bit 2 **IOFS:** HFINTOSC Frequency Stable bit
1 = HFINTOSC frequency is stable
0 = HFINTOSC frequency is not stable

bit 1-0 **SCS<1:0>:** System Clock Select bits
1x = Internal oscillator block
01 = Secondary (Timer1) oscillator
00 = Primary clock (determined by CONFIG1H[FOSC<3:0>]).

Note 1: Reset state depends on state of the IESO Configuration bit.
2: Source selected by the INTSRC bit of the OSCTUNE register, see text.
3: Default output frequency of HFINTOSC on Reset.

The IDLEN bit in OSCCON affects how the oscillator behaves in power managed modes, and is not discussed further here.

The IRFCx bits determine the internal oscillator frequency. These are the outputs of the postscaler. As Note 2 in Figure 3-44 indicates, the 31 kHz clock can be selected as either a postscaled version of the

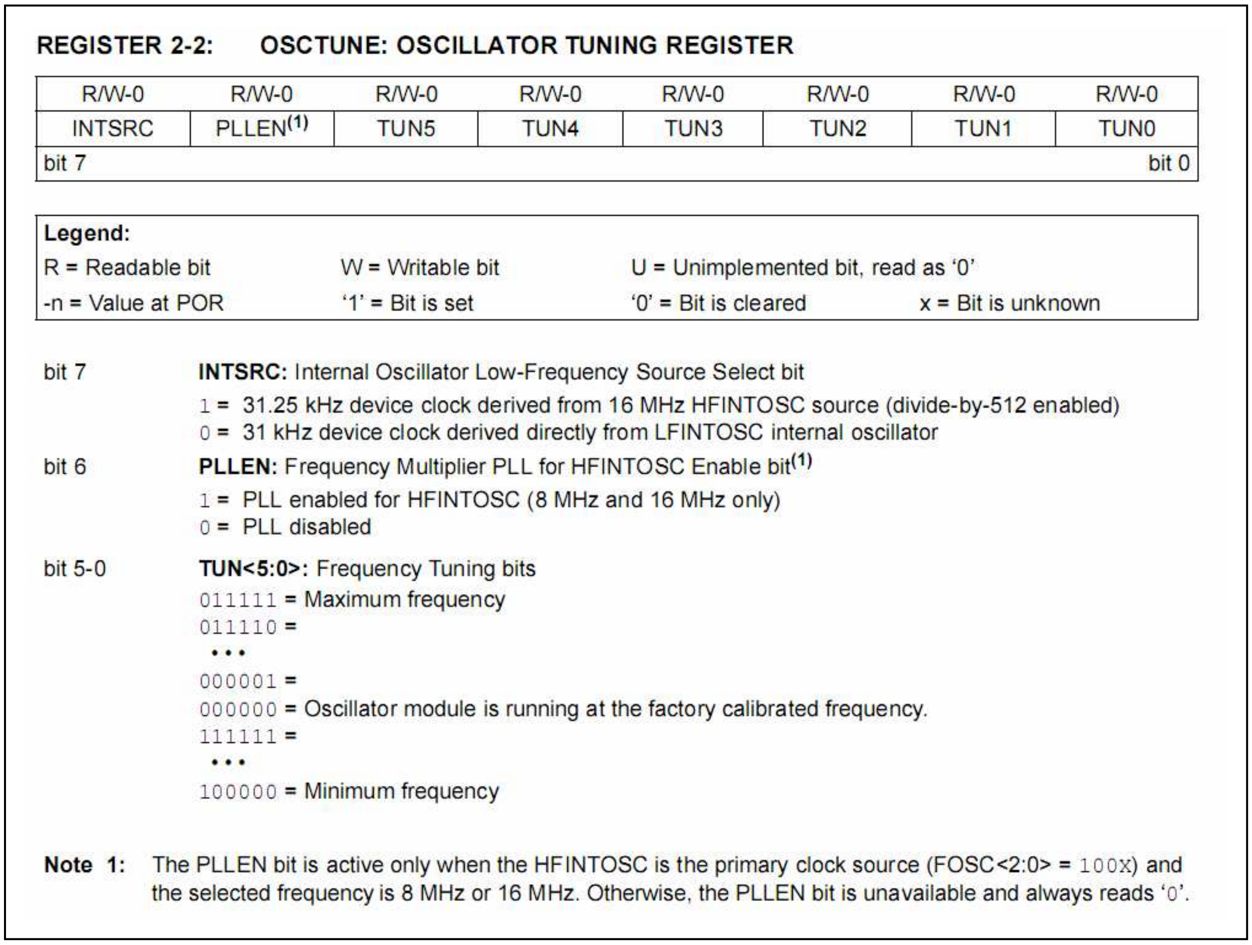
INTOSC 8 MHz oscillator, on which all other frequencies are based, or the INTRC low power 31 kHz oscillator as discussed in section 3.9.1. This selection is made with the INTSRC bit in the OSCTUNE register.

The IRFCx bits may be changed by software during program execution, allowing the program to "throttle" the microcontroller execution speed to current processing needs. This can save on power consumption when fast clock speeds aren't required.

The OSTS and IOFS bits are read-only status bits. The PIC18F46K20 has the option to startup running off the internal oscillator until an external oscillator circuit has stabilized. This allows faster startup of the microcontroller with external oscillators. OSTS is used to alert the software when the clock source has switched over to the external primary oscillator. This functionality is not covered further in this lesson.

The SCSx bits allow the software to switch the microcontroller clock source over to the internal oscillator block even when an external oscillator has been selected in the Configuration bits. The Secondary oscillator may also be selected, which is the low-speed low-power oscillator that is part of Timer1 and is usually run with a 32kHz crystal for real-time-clock applications. In this lesson, the internal oscillator has been selected as the primary oscillator in the Configuration bits, and SCS1:SCS0 = 00.

FIGURE 3-45: OSCTUNE: OSCILLATOR TUNING REGISTER



REGISTER 2-2: OSCTUNE: OSCILLATOR TUNING REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
INTSRC	PLLEN[1]	TUN5	TUN4	TUN3	TUN2	TUN1	TUN0
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 **INTSRC:** Internal Oscillator Low-Frequency Source Select bit
1 = 31.25 kHz device clock derived from 16 MHz HFINTOSC source (divide-by-512 enabled)
0 = 31 kHz device clock derived directly from LFINTOSC internal oscillator

bit 6 **PLLEN:** Frequency Multiplier PLL for HFINTOSC Enable bit[1]
1 = PLL enabled for HFINTOSC (8 MHz and 16 MHz only)
0 = PLL disabled

bit 5-0 **TUN<5:0>:** Frequency Tuning bits
011111 = Maximum frequency
011110 =
•••
000001 =
000000 = Oscillator module is running at the factory calibrated frequency.
111111 =
•••
100000 = Minimum frequency

Note 1: The PLLEN bit is active only when the HFINTOSC is the primary clock source (FOSC<2:0> = 100x) and the selected frequency is 8 MHz or 16 MHz. Otherwise, the PLLEN bit is unavailable and always reads '0'.

The 5 TUNx bits in OSCTUNE allow small adjustments in the INTOSC oscillator frequency. This can be used to calibrate the frequency more accurately than the factory calibration, and adjust for drift over Vdd and temperature changes.

The PLLEN bit enables the PLL, multiplying the INTOSC output by 4. Note that the PLL may only be enabled for INTOSC = 8 MHz or INTOSC = 16 MHz. Enabling the PLL with a 4 MHz base frequency gives a 16 MHz clock, and with a 16 MHz base frequency gives 64 MHz.

For further information on the internal oscillator block, see section 2.6 of the PIC18F46K20 Datasheet.

3.9.3 Exploring the Lesson 9 Source Code

The lesson 9 program code has a simple background loop in the `main()` function that displays a binary count on the demo board LEDs, as shown in Figure 3-46. Each count increment is delayed by 32,000 instruction cycles. As the clock frequency is changed, the instruction rate changes and so the total time in seconds of the delay gets shorter as the clock frequency increases. The effect is that the LED display will count faster as the clock speed is increased.

At the start of the program, the internal oscillator is running at 250 kHz. Each press of the demo board switch creates an interrupt that increases the clock frequency by a factor of 2 up through 64 MHz, after which it returns to 250 kHz.

FIGURE 3-46: SOURCE CODE BACKGROUND LOOP

```
while (1)
{ // delay and count on LEDs here.  Interrupt handles switch and freq changes

    LATD = LED_Count++;         // output count to PORTD LEDs
    Delay1KTCYx(32);            // delay 32,000 cycles or about 1 sec at 125kHz
}
```

A few other things of interest in the lesson 9 source code are:

- The interrupts are configured for only a single level of priority, where interrupt priorities are disabled. This differs from the lesson 8 source code where interrupt priorities were enabled.
- Instead of using ADCON1 to configure the switch input RB0 as a digital input as was done in previous lessons, the lesson 9 source sets the Configuration bit PBADEN = OFF. This causes all PORTB pins to default to digital, instead of analog, inputs on a reset.
- The lesson 9 interrupt service function `void InterruptService(void)` demonstrates calling another function `void SetIntOSC(IntOSCFreq *ClockSet)` from within the interrupt service code.

3.9.4 Build and Run the Lesson 9 Code with PICkit 2 Debug Express

Build and program the lesson 9 project, then Run the application in the debugger. Pressing the demo board switch causes the program to change the oscillator frequency during execution. As the oscillator frequency increases, the rate at which the LEDs count increases.

3.10 Lesson 10: Using Internal EEPROM

The PIC18F46K20 microcontroller includes 256 bytes of on-chip EEPROM for data storage. This lesson discusses reading and writing the internal EEPROM in software.

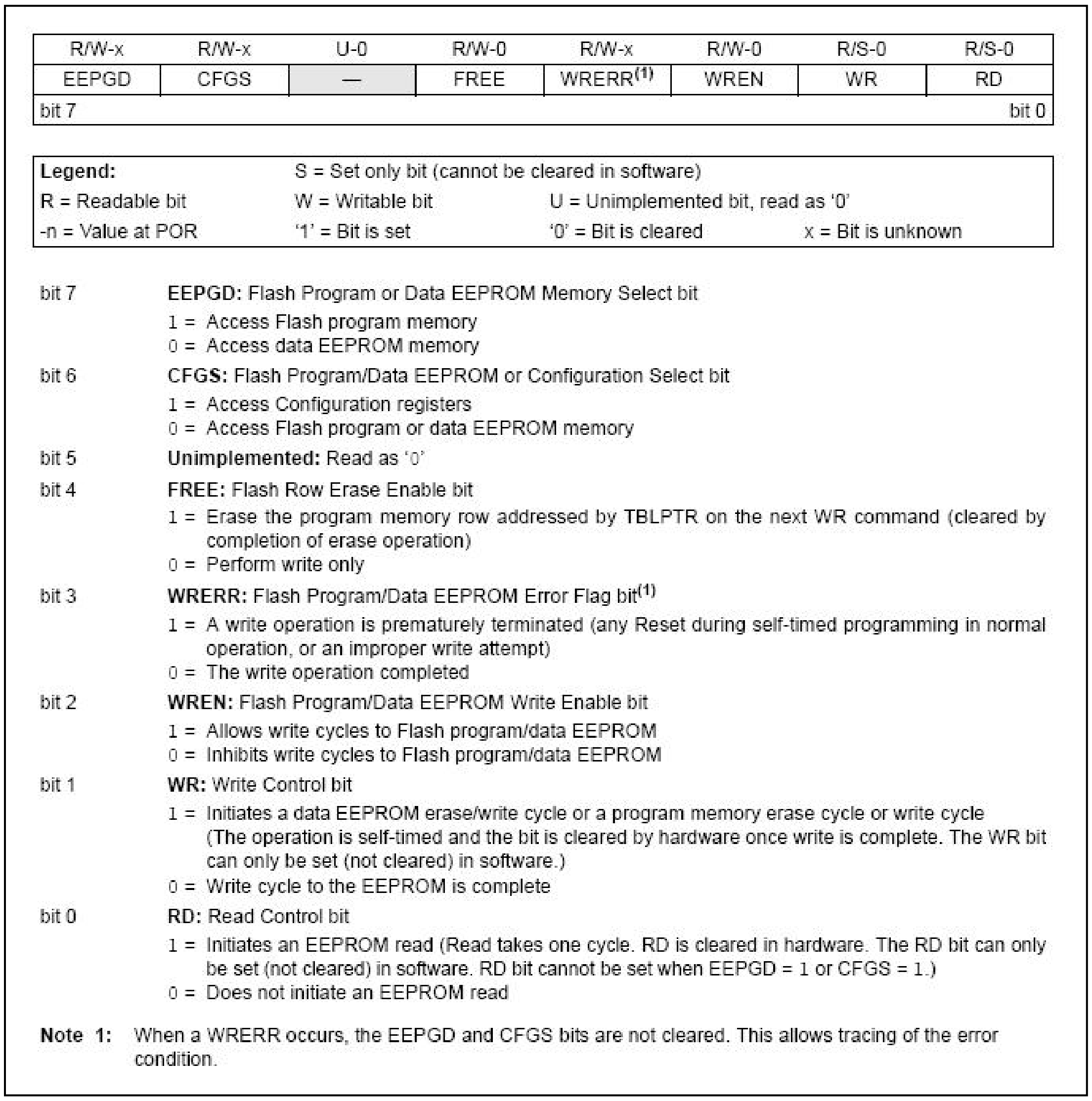
Key Concepts

- The 4 SFRs that control EEPROM operations are EECON1, EECON2, EEDATA, and EEADR.
- The internal EEPROM is written and read one byte at a time.
- To write EEPROM, a short code sequence must be written to EECON2 immediately before starting the write operation. This is to prevent inadvertent EEPROM writes.
- Writing a byte to EEPROM takes a period of time before the write cycle is complete. The microcontroller will continue to execute code during an EEPROM write cycle.

3.10.1 Reading a data byte from EEPROM

The EECON1 Special Function Register controls operations to both the internal EEPROM as well as the Program Memory flash array.

FIGURE 3-47: EECON1: EEPROM CONTROL REGISTER 1



R/W-x	R/W-x	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
EEPGD	CFGS	—	FREE	WRERR[(1)]	WREN	WR	RD
bit 7							bit 0

Legend:	S = Set only bit (cannot be cleared in software)		
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 **EEPGD:** Flash Program or Data EEPROM Memory Select bit
1 = Access Flash program memory
0 = Access data EEPROM memory

bit 6 **CFGS:** Flash Program/Data EEPROM or Configuration Select bit
1 = Access Configuration registers
0 = Access Flash program or data EEPROM memory

bit 5 **Unimplemented:** Read as '0'

bit 4 **FREE:** Flash Row Erase Enable bit
1 = Erase the program memory row addressed by TBLPTR on the next WR command (cleared by completion of erase operation)
0 = Perform write only

bit 3 **WRERR:** Flash Program/Data EEPROM Error Flag bit[(1)]
1 = A write operation is prematurely terminated (any Reset during self-timed programming in normal operation, or an improper write attempt)
0 = The write operation completed

bit 2 **WREN:** Flash Program/Data EEPROM Write Enable bit
1 = Allows write cycles to Flash program/data EEPROM
0 = Inhibits write cycles to Flash program/data EEPROM

bit 1 **WR:** Write Control bit
1 = Initiates a data EEPROM erase/write cycle or a program memory erase cycle or write cycle (The operation is self-timed and the bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.)
0 = Write cycle to the EEPROM is complete

bit 0 **RD:** Read Control bit
1 = Initiates an EEPROM read (Read takes one cycle. RD is cleared in hardware. The RD bit can only be set (not cleared) in software. RD bit cannot be set when EEPGD = 1 or CFGS = 1.)
0 = Does not initiate an EEPROM read

Note 1: When a WRERR occurs, the EEPGD and CFGS bits are not cleared. This allows tracing of the error condition.

A read of an EEPROM byte begins by clearing the EEPGD bit in EECON1. This selects the data EEPROM array for access. The CFGS bit should also be cleared during an EEPROM access; it is only set to access the Configuration bit locations.

The byte address of the data EEPROM location to be read is loaded into the EEADR register. The RD bit in EECON1 is then set to execute the read. On the next instruction cycle, the value of the read EEPROM location is available in the EEDATA register. Figure 3-48 shows a function that reads a byte of EEPROM.

FIGURE 3-48: DATA EEPROM READ

```
unsigned char EEPROM_Read(unsigned char address)
{ // reads and returns the EEPROM byte value at the address given
  // given in "address".

    EECON1bits.EEPGD = 0;   // Set to access EEPROM memory
    EECON1bits.CFGS = 0;    // Do not access Config registers

    EEADR = address;        // Load EEADR with address of location to write.

    // execute the read
    EECON1bits.RD = 1;      // Set the RD bit to execute the EEPROM read

    // The value read is ready the next instruction cycle in EEDATA.  No wait is
    // needed.

    return EEDATA;
}
```

3.10.2 Writing a data byte to EEPROM

Similar to a read, a write to the internal EEPROM must clear the EEPGD and CFGS bits in EECON1 to access the internal EEPROM array. The data value to be written is then written to the EEDATA register. The address of the byte to be written is loaded into EEADR.

Before a write can take place, the WREN bit in EECON1 must be set, or the write will not occur. It is also necessary to write a sequence of two bytes, values 0x55 and 0xAA to EECON2 immediately before beginning the write by setting the WR bit in EECON1. Both the WREN bit and the EECON2 sequence are to protect against inadvertent writes to EEPROM and ensure the integrity of EEPROM values.

The three step sequence of:

```
      EECON2 = 0x55;
      EECON2 = 0xAA;
      EECON1bits.WR = 1;
```

<u>must</u> be completed in this order, without other statements or interruptions or the write will not execute. Therefore, if interrupts are enabled, they should be disabled before the sequence and re-enabled after the WR bit is set.

EEPROM writes take some time to erase and program the byte in the array. This time is listed as parameter D122 in the datasheet section 26.0 Electrical Characteristics, and is usually several ms. During this time, the PIC18F46K20 microcontroller continues to execute program code. The program may determine when a write has completed by polling or by an interrupt generated by the EEPROM module.

In the example write function in Figure 3-49, the code waits for the EEPROM write to complete by polling the WR bit of EECON1. When the write is complete, this bit will be cleared. Alternatively, the program can be alerted that the write has been completed with an interrupt. The EEPROM module will set the EEIF bit in PIR2 when the write completes.

For more information on the data EEPROM memory see section 7.0 of the PIC18F46K20 datasheet.

FIGURE 3-49: DATA EEPROM WRITE

```
void EEPROM_Write(unsigned char address, unsigned char databyte)
{ // writes the "databyte" value to EEPROM at the address given
  // location in "address".
    EECON1bits.EEPGD = 0;   // Set to access EEPROM memory
    EECON1bits.CFGS = 0;    // Do not access Config registers

    EEDATA = databyte;      // Load EEDATA with byte to be written
    EEADR = address;        // Load EEADR with address of location to write.

    EECON1bits.WREN = 1;    // Enable writing

    INTCONbits.GIE = 0;     // Disable interrupts
    EECON2 = 0x55;          // Begin Write sequence
    EECON2 = 0xAA;
    EECON1bits.WR = 1;      // Set WR bit to begin EEPROM write
    INTCONbits.GIE = 1;     // re-enable interrupts

    while (EECON1bits.WR == 1)
    {   // wait for write to complete.
    };

    EECON1bits.WREN = 0;    // Disable writing as a precaution.
}
```

3.10.3 Exploring the Lesson 10 Source Code

The lesson 10 program writes all 256 bytes of the data EEPROM memory, writing each location with value = 255 – address. For example, the EEPROM byte at address 0x09 is written with value 0xF6 = 246.

Once all locations have been written, the program ends in an infinite `while(1)` loop.

3.10.4 Build and Run the Lesson 10 Code with PICkit 2 Debug Express

Build and program the lesson 10 project, then Run the application in the debugger. The EEPROM memory may be viewed in the MPLAB IDE by selecting *view > EEPROM*.

Note: The EEPROM window in the MPLAB IDE does **not** update with new EEPROM values during debugging.

As the EEPROM memory window does not update with changed EEPROM byte values during debugging, it is necessary to select *Debugger > Read EEDATA* to see the current contents of the data EEPROM memory. However, doing so will cause a program reset.

3.11 Lesson 11: Program Memory Operations

Topics covered in this include reading, writing, and erasing locations in the Flash Program Memory, protecting areas of program memory in the Configuration bits, and considerations for using C pointers to program memory.

Key Concepts

- Pointers declared with the `rom` keyword point to program memory locations.
- The EECON1 and EECON2 SFRs control program memory erase and write operations.
- Unlike Data EEPROM Memory, the Flash Program Memory must be explicitly erased before it may be written.
- The CPx (Code Protect) Configuration bits prevent programmers from reading ranges of a microcontroller's program memory.
- The WRTx Configuration bits prevent software write operations on ranges of program memory, and the EBTRx bits prevent software read operations on ranges of program memory.

3.11.1 ROM Pointers and Reading Flash Program Memory

The MPLAB C18 Compiler simplifies working with data stored in program memory by allowing pointers to program memory to be declared. The pointer address length is either 16 or 24 bits, depending on which "Code Model" is selected in the project settings. The "Small Code Model" will generate 16-bit pointers, while the "Large Code Model" generates 24-bit pointers. For the best microcontroller performance, the "Small Code Model" with 16-bit pointers should be used. The "Large Code Model" is necessary for devices that have more than 64 KB of Flash Program Memory to be able to point to locations above the first 64 KB of program memory. (The maximum of a 16-bit value is 65536 which is 64 x 1024 or 64 K).

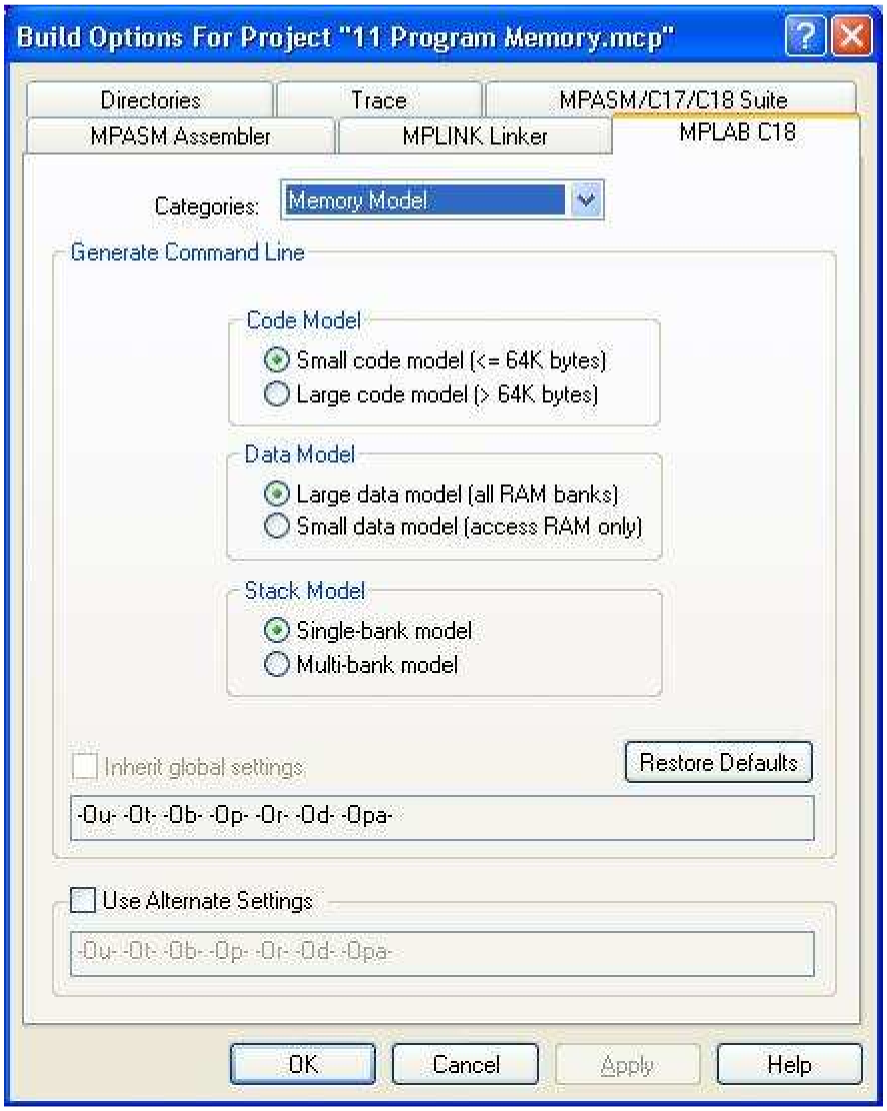
The Code Model settings may changed in the MPLAB IDE by selecting *Project > Build Options… > Project*. This brings up the Build Options dialog. Select the "MPLAB C18" tab and then "Memory Model" from the "Categories" drop-down box as shown in Figure 3-50.

An individual pointer declaration may also use the keywords `near` or `far` to explicitly specify the pointer address length. Use of either keyword overrides the code model settings.

```
      near rom char *rom_pointer;     // 16-bit pointer to program memory
      far rom char *rom_pointer;      // 24-bit pointer to program memory
```

For more information on project memory models, see Chapter 3 of the *MPLAB C18 C Compiler User's Guide*.

FIGURE 3-50: PROJECT CODE MODEL SETTINGS


Build Options For Project "11 Program Memory.mcp"

Tabs: Directories | Trace | MPASM/C17/C18 Suite | MPASM Assembler | MPLINK Linker | MPLAB C18

Categories: Memory Model

Generate Command Line

Code Model
- (●) Small code model (<= 64K bytes)
- ( ) Large code model (> 64K bytes)

Data Model
- (●) Large data model (all RAM banks)
- ( ) Small data model (access RAM only)

Stack Model
- (●) Single-bank model
- ( ) Multi-bank model

☐ Inherit global settings

Restore Defaults

-Ou- -Ot- -Ob- -Op- -Or- -Od- -Opa-

☐ Use Alternate Settings

-Ou- -Ot- -Ob- -Op- -Or- -Od- -Opa-

OK | Cancel | Apply | Help

Once a pointer to program memory has been declared, it can be pointed to a declared location in program memory, for example a `#pragma romdata` array, or an explicit address.

```
#pragma romdata mystrings = 0x100
rom char hello_str[] = "Hello!";

rom_pointer = hello_str;        // = &hello_str[0]
char letter = *rom_pointer
```

The first letter ‘H’ of the `hello_str[]` array in program memory is now pointed to by `rom_pointer`. The value of the variable `letter` is now ‘H’.

```
rom_pointer = (near rom char *)0x320;
```

Now, `rom_pointer` points to the program memory byte at address 0x320.

Reading Flash Program Memory then simply requires declaring a `rom` pointer and using an assignment statement to read the pointer value.

3.11.2 Erasing and Writing Flash Program Memory

Unlike writing Data EEPROM Memory, writing Flash Program Memory requires that the locations being written are erased first. When erased, a program memory location has all bits set to ‘1’. Thus an erased byte has the hex value 0xFF. Writing a program memory location sets the appropriate bits to ‘0’, but a write cannot set a bit ‘1’. Also different from EEPROM operations is that program memory erases and writes cannot operate on a single byte, but instead operation on “blocks” of a particular number of bytes.

The PIC18F46K20 erase block size is 64 bytes. This means it will always erase 64 sequential bytes at once, and the block must start at an address that is a multiple of 64. For example, we could erase the 64 bytes from address 128 through 191 at once, but not the 64 bytes from address 100 through 163.

To erase a 64 byte block of program memory, we use a `rom` pointer to set the address of the block to be erased, and use EECON1 to control the erase. Setting the pointer address puts the address in the TBLPTRx Special Function Registers. These 3 registers hold the address for program memory operations with `TBLRD` and `TBLWR` assembly instructions. The MPLAB C18 compiler handles these tasks for us. The EEPGD bit EECON1 is set to ‘1’, so the operation affects program memory and not data EEPROM. The CFGS bit is set to ‘0’, as we do not want to select the Configuration bits. To select an erase operation as opposed to a write operation, bit FREE of EECON1 is set to ‘1’. WREN is then set to ‘1’ to enable write/erase operations.

```
    // point to address 2176, which is a multiple of 64
    rom_pointer = (near rom char *)0x880;

    EECON1bits.EEPGD = 1;      // point to flash program memory
    EECON1bits.CFGS = 0;       // not configuration registers
    EECON1bits.FREE = 1;       // we're erasing
    EECON1bits.WREN = 1;       // enable write/erase operations
```

Next, the EECON2 sequence must be followed as with data EEPROM writes, and the WR bit of EECON1 is set to initiate the write.

```
    INTCONbits.GIE = 0;     // Disable interrupts
    EECON2 = 0x55;          // Begin Write sequence
    EECON2 = 0xAA;
    EECON1bits.WR = 1;      // Set WR bit to begin EEPROM write
    INTCONbits.GIE = 1;     // re-enable interrupts
```

As with a data EEPROM write, and erase or write to Flash Program Memory takes up to several ms to complete. While there is an active erase or a write operation to program memory, all microcontroller program execution is halted since it is possible the microcontroller might attempt to execute instructions from the locations being erased or written. This would be illegal, as the program memory location’s value is in an indeterminate state until the operation has completed.

The PIC18F46K20 write block size is 32 bytes. This requires that we write 32 sequential bytes at a time. As with erasing, the first byte must be at an address that is a multiple of the block size, 32.

The sequence for writing program memory is very similar to that for erasing. The differences are that a `rom` pointer is used to write the 32 locations, and that the EECON1 bit FREE is cleared to select a write operation. Don't forget that the locations to be written must be erased first!

When the 32 locations are written with the pointer, they are not actually written to program until the completion of the entire sequence. The pointer writes actually store the data in 32 temporary hardware registers. When the actual write sequence is executed, it is the contents of this 32 byte buffer that is written to the program memory array. For example, we might use a `for` loop to write the contents of a RAM array to these buffers using a `rom` pointer.

```
    for (i = 0; i < 32; i++)
    {
        *(rom_pointer + i) = ram_array[i];     // write to the holding registers
    }
```

This data is not actually in program memory yet, and won't be until the entire write sequence is completed as shown in Figure 3-51.

Note: The program memory block that is written to is determined by the address in the TBLPTRU:TBLPTRH:TBLPRTL Special Function Registers, excluding the 5 least significant bits. These bits are excluded to ensure the write block begins on a 32 byte boundary. **Therefore, it is critically important that the pointer address is not incremented past the last address in the block.** If this occurs, the 32 bytes will be written at the next block boundary instead of the intended one.

As an example for the above note, suppose using the following code we intended to write to the 32 block of program memory from address 0x100 to 0x11F. The data would actually be written to address 0x120 because the pointer is incremented to address 0x120 after the last write.

```
    rom_pointer = (near rom unsigned char *)0x100;

    for (i = 0; i < 32; i++)
    {
        *(rom_pointer++) = ram_array[i];     // write to the holding registers
    }
    // after the for loop, the rom_pointer address value is 0x120.
```

If the `rom_pointer` value were left at 0x11F, the data would be written as intended started at 0x100.

FIGURE 3-51: EXAMPLE PROGRAM MEMORY WRITE FUNCTION

```
unsigned char ProgMemWr32(unsigned int address, unsigned char *buffer_ptr)
{ // NOTE: program memory must also be erased first.
    near rom unsigned char *ptr;
    char i;

    ptr = (rom unsigned char *)(address & 0xFFE0);// ensure write starts on 32-byte boundary

    for (i = 0; i < 32; i++)
    {
        *(ptr + i) = buffer_ptr[i];     // write the data into the holding registers
    }

    EECON1bits.EEPGD = 1;               // write to flash program memory
    EECON1bits.CFGS = 0;                // not configuration registers
    EECON1bits.FREE = 0;                // we're not erasing now.
    EECON1bits.WREN = 1;                // enable write/erase operations

    // execute code sequence, which cannot be interrupted, then execute write32

    INTCONbits.GIE = 0;     // Disable interrupts
    EECON2 = 0x55;          // Begin Write sequence
    EECON2 = 0xAA;
    EECON1bits.WR = 1;      // Set WR bit to begin 32-byte write
    INTCONbits.GIE = 1;     // re-enable interrupts

    EECON1bits.WREN = 0;                // disable write/erase operations
}
```

3.11.3 Protecting Program Memory in the Configuration Bits.

The program is divided into sections that can individually be protected by setting the appropriate Configuration bits. The protections available are:

Code Protect – The CPx bits prevent microcontroller programmers such as the PICkit 2 from reading the contents of program memory in the address range associated with the particular CPx configuration bit. If a programmer attempts to read a code-protected section of memory, all locations will read as value 0x00. This prevents other parties from stealing proprietary program code.

Write Protect – When a WRTx configuration bit is ON, then program memory erase or write operations prohibited from working on the associated range of memory. This could be used to protect a bootloader from accidental corruption by inadvertent application program memory writes or erases.

Table Read Protect – The EBTRx bits, when asserted, prevent program memory locations being read from instructions executing in another program memory block. For example, if EBTR3 was asserted, then program memory locations from 0x6000 to 0x7FFF by any code executing from program memory locations 0x0000 to 0x5FFF. Locations in the block 0x6000 to 0x7FFF could still be read by code executing in that block. This could be used, for example, to prevent using a bootloader to read out sensitive code-protected data.

Once these protective Configuration bits have been asserted (set to ON), they cannot be turned off or changed without a programmer executing a Bulk Erase on the microcontroller, which erases all program memory and data EEPROM memory. It is possible to prevent other Configuration bits from being changed after the device is initially programmed using the WRTC Configuration bit.

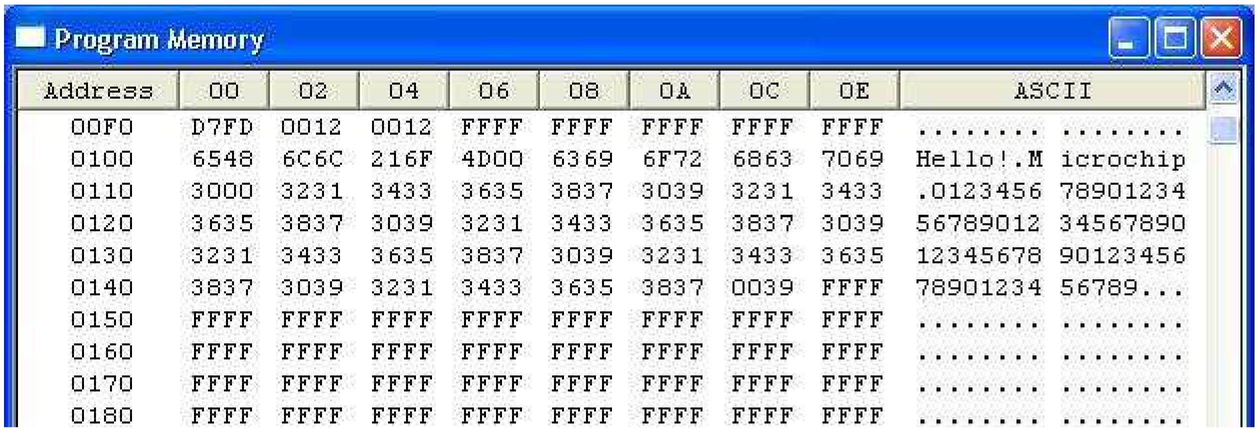
3.11.4 Exploring the Lesson 11 Source Code with PICkit 2 Debug Express

At compile time, when the project is built, the lesson 11 source code places three strings in Flash Program Memory at address 0x100:

```
#pragma romdata mystrings = 0x100
rom char hello_str[] = "Hello!";
rom char mchp_str[] = "Microchip";
rom char fill_60[] =
"012345678901234567890123456789012345678901234567890123456789";
```

After building the project, the strings can be seen in Program Memory by opening the Program Memory window in the MPLAB IDE using *View > Program Memory*.

FIGURE 3-52: STRINGS IN PROGRAM MEMORY

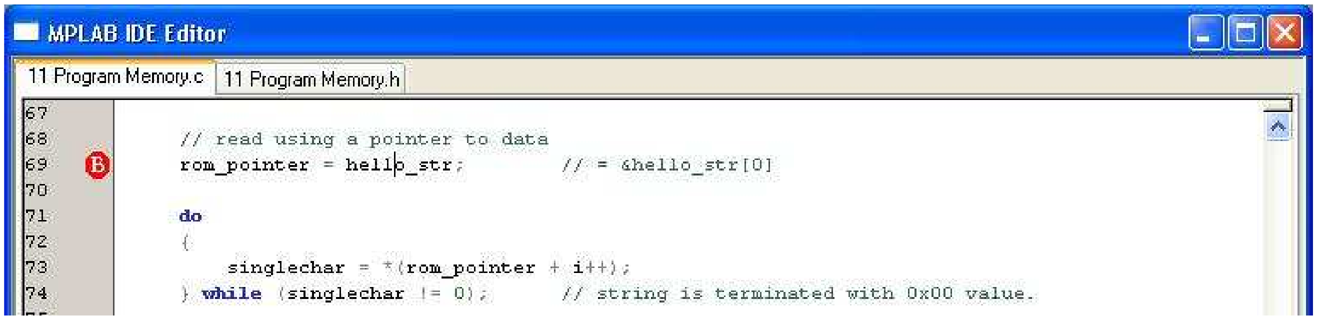


```
Address   00    02    04    06    08    0A    0C    0E           ASCII
 00F0    D7FD  0012  0012  FFFF  FFFF  FFFF  FFFF  FFFF   ........ ........
 0100    6548  6C6C  216F  4D00  6369  6F72  6863  7069   Hello!.M icrochip
 0110    3000  3231  3433  3635  3837  3039  3231  3433   .0123456 78901234
 0120    3635  3837  3039  3231  3433  3635  3837  3039   56789012 34567890
 0130    3231  3433  3635  3837  3039  3231  3433  3635   12345678 90123456
 0140    3837  3039  3231  3433  3635  3837  0039  FFFF   78901234 56789...
 0150    FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF   ........ ........
 0160    FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF   ........ ........
 0170    FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF   ........ ........
 0180    FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF  FFFF   ........ ........
```

The program code doesn't start until address 0x280.

Build and program the lesson 11 code and set a breakpoint on the first pointer assignment statement as shown in Figure 3-53.

FIGURE 3-52: BREAKPOINT ON POINTER ASSIGNMENT



```
    // read using a pointer to data
    rom_pointer = hello_str;        // = &hello_str[0]

    do
    {
        singlechar = *(rom_pointer + i++);
    } while (singlechar != 0);      // string is terminated with 0x00 value.
```

Run the program until is stops at the breakpoint. Step through the `do while` loop in Figure 3-53 and observe the characters of the `hello_str[]` string are read into the `singlechar` variable one at a time until the terminating '0' value of the string is reached.

The next statement demonstrates reading from an explicit program memory address using a function:

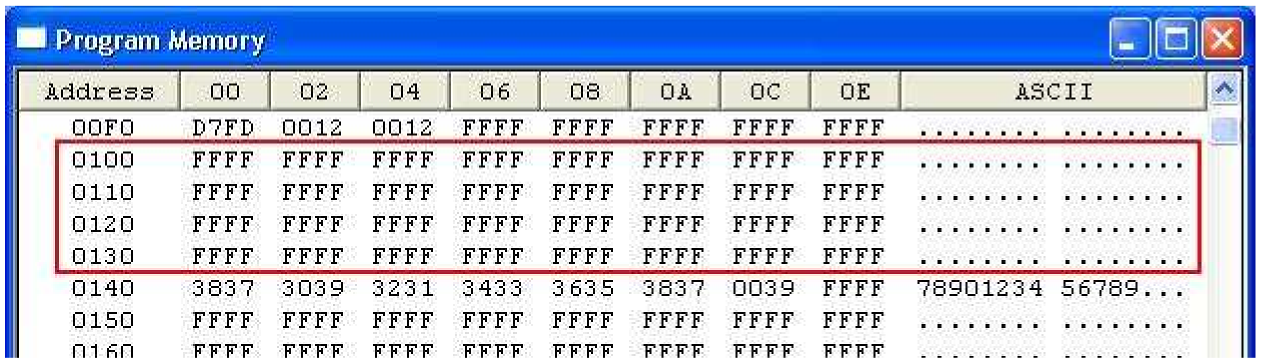
```
singlechar = ProgMemRdAddress(0x107);  // returns 'M' from "Microchip".
```

Step into the following statement and through the function, which erases a 64 byte block of memory that the strings are stored in.

```
// Erase the 64 bytes starting at 0x100
ProgMemErase64(0x100);
```

After completing the erase, select menu *Debugger > Read*. In the Program Memory window, the 64 bytes of program memory starting at address 0x0100 where the strings were stored have been erased, as shown in Figure 3-53.

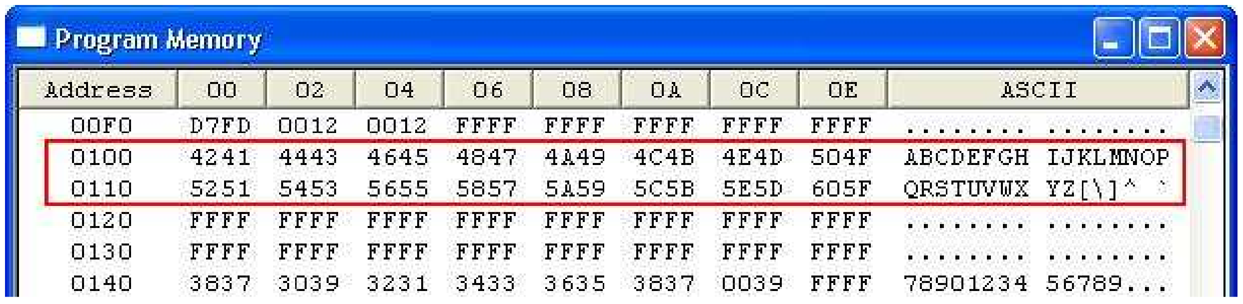
FIGURE 3-53: ERASED 0x0100 TO 0x013F



```
Address  00   02   04   06   08   0A   0C   0E    ASCII
 00F0   D7FD 0012 0012 FFFF FFFF FFFF FFFF FFFF  ........ ........
 0100   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0110   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0120   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0130   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0140   3837 3039 3231 3433 3635 3837 0039 FFFF  78901234 56789...
 0150   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0160   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
```

The remaining code creates a 32 byte buffer in RAM and fills it with the alphabet characters in uppercase, plus a few punctuation characters at the end. This buffer is then written to the 32 byte block of program memory starting at 0x0100 that was just erased. Since we read program memory, we'll have to reset the debugger. Select *Debugger > Reset > Processor Reset*. Right-click on the source code and select *Breakpoints > Remove All Breakpoints* from the pop-up menu to clear the breakpoint we set earlier. Run the program. After running for a few seconds, select *Debugger > Halt*. The program should be stopped at final `while(1)` loop. Select *Debugger > Read* again and we can see that the write to program memory was successful.

FIGURE 3-54: PROGRAM MEMORY WRITE RESULTS



```
Address  00   02   04   06   08   0A   0C   0E    ASCII
 00F0   D7FD 0012 0012 FFFF FFFF FFFF FFFF FFFF  ........ ........
 0100   4241 4443 4645 4847 4A49 4C4B 4E4D 504F  ABCDEFGH IJKLMNOP
 0110   5251 5453 5655 5857 5A59 5C5B 5E5D 605F  QRSTUVWX YZ[\]^_`
 0120   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0130   FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF  ........ ........
 0140   3837 3039 3231 3433 3635 3837 0039 FFFF  78901234 56789...
```

3.12 Lesson 12: Using the CCP Module PWM

This lesson gives a brief introduction to using the Pulse Width Modulation (PWM) functionality of the Capture/Compare/Pwm (CCP) peripheral of the PIC18F46K20.

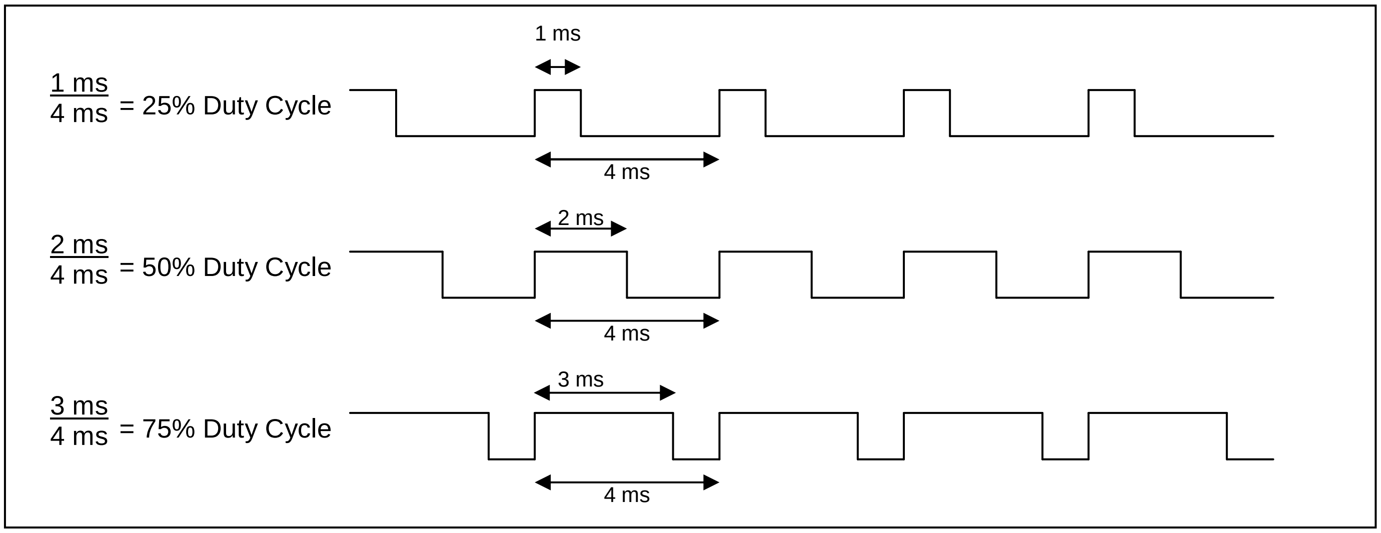
Key Concepts

- The PWM timebase (frequency) is determined by Timer2 and the PR2 Special Function Register.
- PWM operation of the CCP module is selected in the CCPxCON SFR.
- Up to 10 bits of resolution are possible, with the 8 MSb's of the duty cycle in CCPRxL, and the 2 LSBs in CCPxCON.
- The actual amount of duty cycle resolution depends on the value of the PR2 register.

3.12.1 PWM Overview

In short, Pulse Width Modulation is a square wave of a given frequency where the duty cycle of the period is varied. The duty cycle is a ratio of how long the signal is high to the total length of the period. For example, a waveform with a frequency of 250 Hz has a period of 4 ms. For a PWM signal with a 25% duty cycle, the waveform would be high for 1 ms and low for 3ms (and then repeat). A PWM signal with 50% duty is high for 2ms and low for 2ms, while a 75% duty cycle would be high for 3ms and low for 1 ms.

FIGURE 3-55: EXAMPLE PWM DUTY CYCLES


1 ms
1 ms / 4 ms = 25% Duty Cycle
4 ms
2 ms
2 ms / 4 ms = 50% Duty Cycle
4 ms
3 ms
3 ms / 4 ms = 75% Duty Cycle
4 ms

Pulse Width modulation is used in a variety of applications, including communications, motor control, audio and analog outputs, and lighting. In this lesson, the brightness of a demo board LED will be controlled with the output of the PWM. The LED is only on during the high portion of the PWM period, and is off during the low period. As the duty cycle is decreased, the LED is on for a shorter and shorter portion of the PWM period, so it appears dimmer. The frequency is set high enough that the human eye cannot detect the individual blinks of each period, but sees the LED light as continuously on.

3.12.2 Using the CCP Module

Timer2 is used to set the period, or frequency, of the PWM waveform. Timer2 operation is very similar to Timer0 discussed in Lesson 5, with a few differences. Namely, Timer2 is always an 8-bit timer.

Timer2 also has a postscaler, but the postscaler does not affect the CPP module operation PWM timebase, so its settings are "don't care." The Timer2 module also has a Period Register, known as PR2. This Special Function Register is the maximum to which Timer2 can count before being reset to 0.

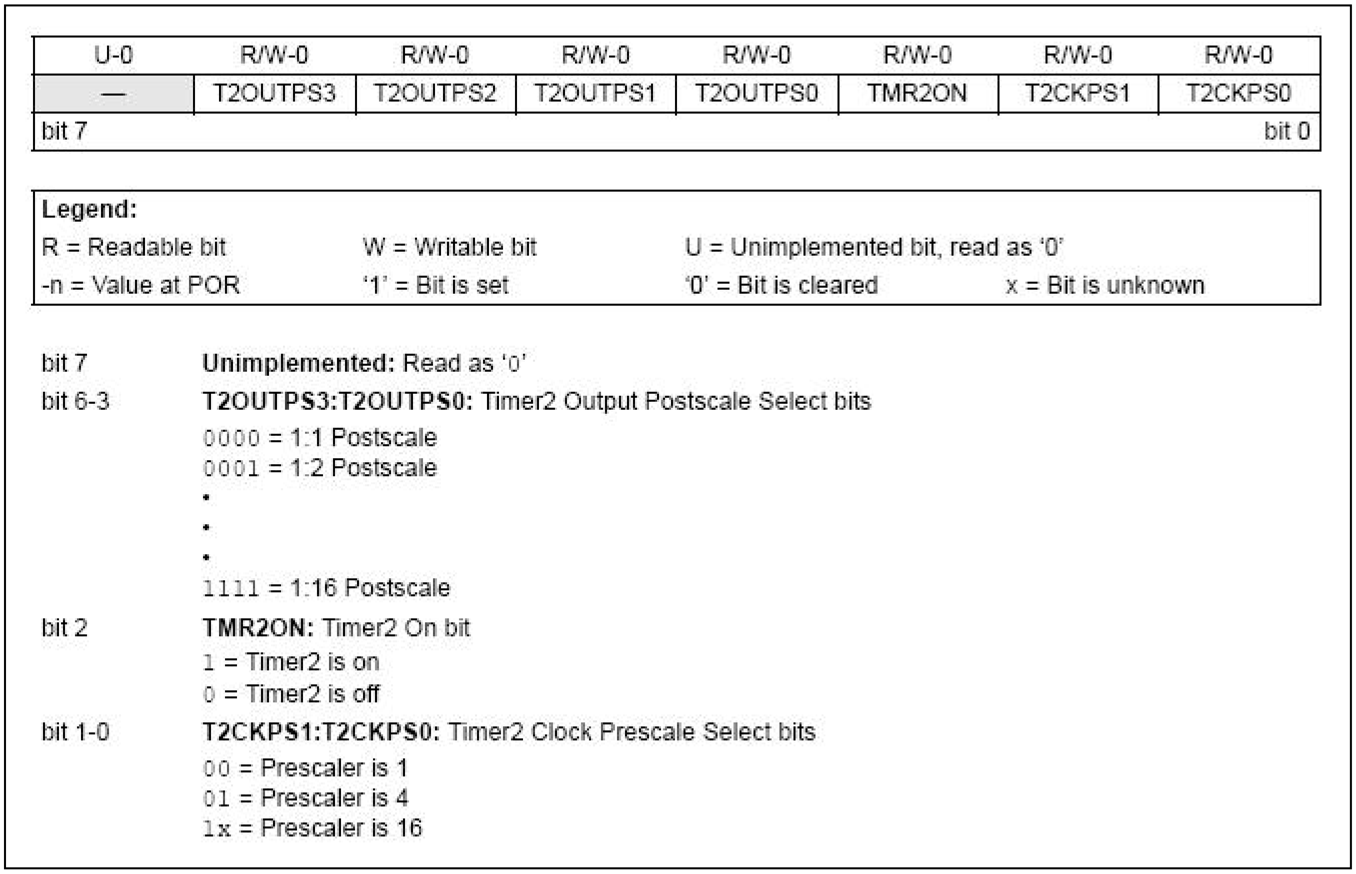
Normally, an 8-bit timer would count up to 255 before resetting to 0 and beginning to count again. With the PR2 register, the timer counts up to the value in PR2. When it reaches this value, the timer is reset to 0. For example if PR2 = 3, then Timer2 would count 0-1-2-3-0-1-2-3-0-1-2-3- etc.

The count cycle from zero up until Timer2 reaches the PR2 in conjunction with the timer prescaler (which determines how long each timer count takes) determines the PWM frequency. The time between each reset to 0 in Timer2 is the PWM period. For example, assume we want a PWM frequency of 62.5Hz, which has a period of 16ms.

Our clock is the internal oscillator block default, 1 MHz, which gives a 250 kHz instruction rate. 250,000 Hz / 62.5 Hz = 4000. Thus, we need to count 4000 times at 250 kHz before each Timer2 reset. However, Timer2 is 8 bits and can count to a maximum of 255. So we must use the prescaler to slow down the counting. Timer2 has 3 prescaler options: 1:1, 1:4, or 1:16 (Figure 3-56). 4000 / 256 = 15.6 so it requires a prescaler of 1:16.

With the prescaler set to 1:16, the count frequency of Timer 2 is 250,000 Hz / 16 = 15625 Hz. To get our PWM frequency of 62.5 Hz, Timer 2 must count 15625 / 62.5 = 250 times. Since Timer2 starts at 0, we set PR2 = 249, so it counts 0-249 (250 counts), resets to zero, and counts back to 249. A simplified diagram of the PWM module is shown in Figure 3-57.

FIGURE 3-56: T2CON: TIMER2 CONTROL REGISTER



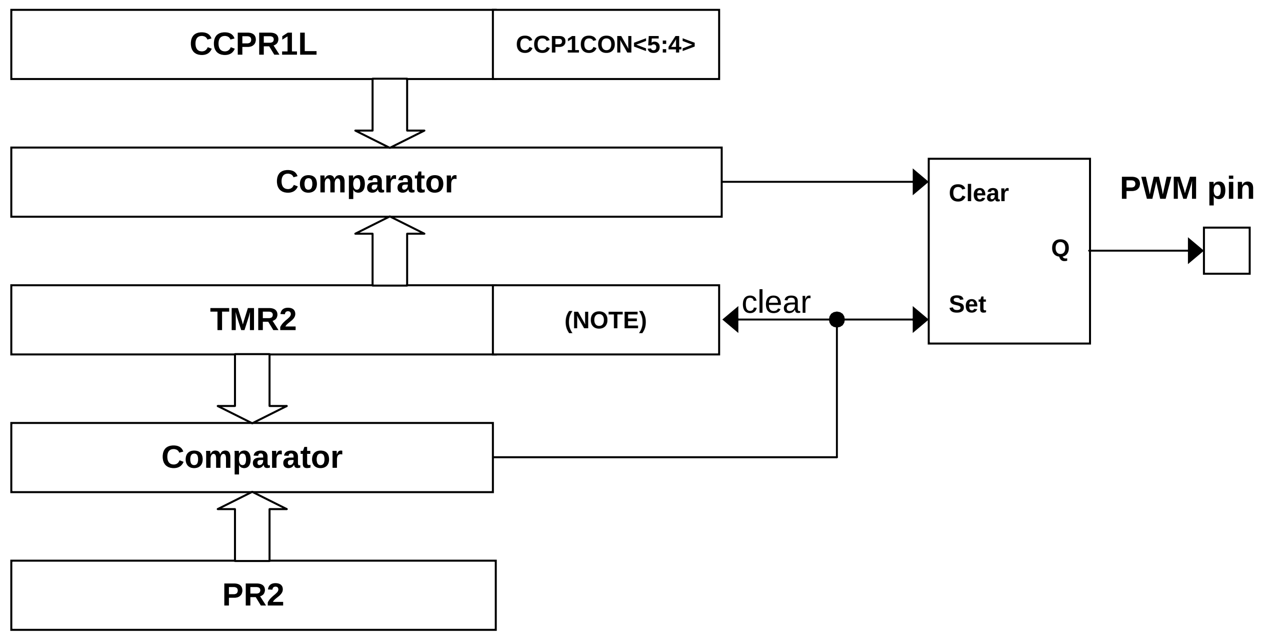
U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

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	**Unimplemented:** Read as '0'
bit 6-3	**T2OUTPS3:T2OUTPS0:** Timer2 Output Postscale Select bits 0000 = 1:1 Postscale 0001 = 1:2 Postscale • • • 1111 = 1:16 Postscale
bit 2	**TMR2ON:** Timer2 On bit 1 = Timer2 is on 0 = Timer2 is off
bit 1-0	**T2CKPS1:T2CKPS0:** Timer2 Clock Prescale Select bits 00 = Prescaler is 1 01 = Prescaler is 4 1x = Prescaler is 16

FIGURE 3-57: SIMPLIFIED PWM BLOCK DIAGRAM

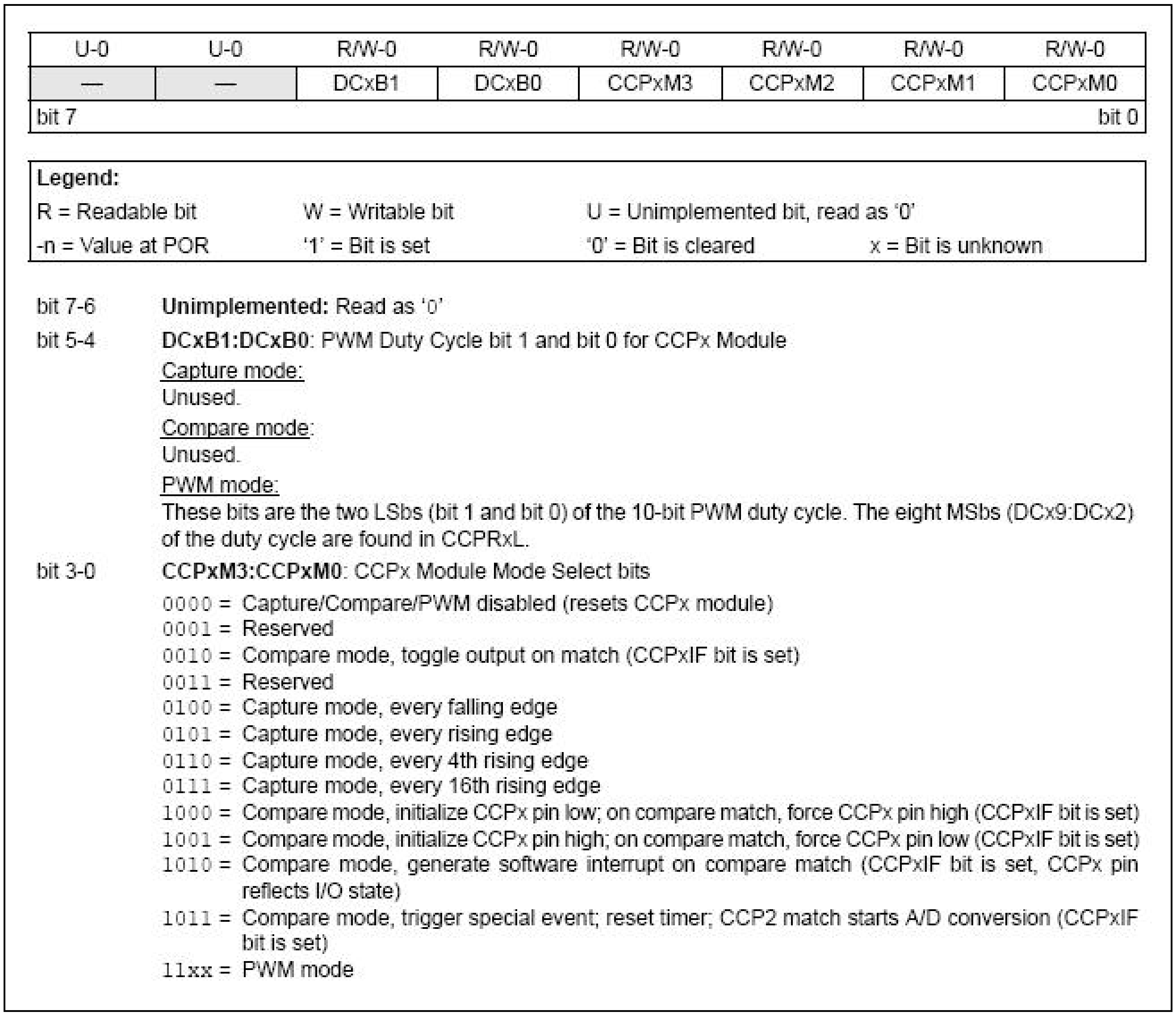


NOTE: To create a 10-bit timebase, the 8-bit TMR2 register is concatenated with the 2-bit internal Q clock, or the 2 most significant bits of the prescaler.

Now that the frequency has been determined, it is necessary to set up the CCP1 module for PWM using the CCP1CON register. Bits CCP1Mx determine the module mode; there is only one value to select for PWM, CCP1Mx = 0b11xx where the 'x' bits are "don't care" so 0b1100 will work. The two DC1Bx bits in CCP1CON are the 2 least significant bits of the 10-bit PWM duty cycle value. The 8 most significant of the 10 bits are written to CCPR1L.

The duty cycle value is determined by the duty cycle percentage (DC%) times the 10-bit timebase (PR2 * 4). DCValue = DC% * (PR2 * 4). For example, to get a duty cycle of 50%, the value would be 50% * (250 * 4) = 500. 500 decimal is 0x1F4 hex or 0b01 1111 0100 binary. The 8 most significant bits, 0b01 1111 01 or 0x7D are written to CCPR1L, and the 2 LSbs are written to the DC1B1 and DC1B0 bits in CCP1CON.

FIGURE 3-58: CCPxCON: CCPx CONTROL REGISTER



U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
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 7-6 **Unimplemented:** Read as '0'

bit 5-4 **DCxB1:DCxB0**: PWM Duty Cycle bit 1 and bit 0 for CCPx Module

Capture mode:
Unused.

Compare mode:
Unused.

PWM mode:
These bits are the two LSbs (bit 1 and bit 0) of the 10-bit PWM duty cycle. The eight MSbs (DCx9:DCx2) of the duty cycle are found in CCPRxL.

bit 3-0 **CCPxM3:CCPxM0**: CCPx Module Mode Select bits

0000 =	Capture/Compare/PWM disabled (resets CCPx module)
0001 =	Reserved
0010 =	Compare mode, toggle output on match (CCPxIF bit is set)
0011 =	Reserved
0100 =	Capture mode, every falling edge
0101 =	Capture mode, every rising edge
0110 =	Capture mode, every 4th rising edge
0111 =	Capture mode, every 16th rising edge
1000 =	Compare mode, initialize CCPx pin low; on compare match, force CCPx pin high (CCPxIF bit is set)
1001 =	Compare mode, initialize CCPx pin high; on compare match, force CCPx pin low (CCPxIF bit is set)
1010 =	Compare mode, generate software interrupt on compare match (CCPxIF bit is set, CCPx pin reflects I/O state)
1011 =	Compare mode, trigger special event; reset timer; CCP2 match starts A/D conversion (CCPxIF bit is set)
11xx =	PWM mode

For more information on Timer2 see section 13.0 Timer2 Module of the PIC18F46K20 Datasheet. More info on the CCP module PWM functionality can be found in section 15.0 Capture/Compare/Pwm (CCP) Module, and section 15.4 PWM Mode.

3.12.3 Exploring the Lesson 12 Source Code

The PWM signal from the CCP1 module is normally output on the CCP1/RC2 pin. However, this pin is not connected to any demo board LEDs. To output a signal on an LED pin, the Enhanced CCP module (ECCP) on the PIC18F46K20 is utilized. This functionality is selected in the upper 2 bits of CCP1CON, (P1Mx) which are set to 0b01 so the modulated PWM signal appears on the P1D/RD7 which drives LED 7. No other aspect of the enhanced PWM functionality is used; for more information see section 16.0 Enhanced Capture/Compare/Pwm (ECCP) Module.

The first thing done in the lesson source code is to set PWM pin RD7 to an output.

```
      TRISDbits.TRISD7 = 0;
```

Timer2 is then configured to generate the PWM period of 16ms as discussed previously in this lesson.

```
      T2CON = 0b00000111;// Prescale = 1:16, timer on
      PR2 = 249;         // Timer 2 Period Register = 250 counts
```

Finally, the CCP1 module is configured for PWM operation with a duty cycle of 50% as described previously in this lesson:

```
      CCPR1L = 0x7D;   // The 8 most significant bits of the period are 0x7D
      CCP1CON = 0b01001100; // The 2 LSbs are 0b00, and CCP1Mx = 110 for PWM
```

At this point in the program in the module running, generating and outputting a PWM signal on RD7/P1D with 50% duty cycle at 62.5 Hz.

To make the LED get brighter and then dimmer, we have a loop that changes the duty cycle. The first `do while` loop increases the brightness over 2 seconds by increasing the duty cycle. As the duty cycle is increased, the LED is on for a longer period of time so it appears brighter. Note that for simplicity, the lesson program only changes the 8 MSbs of the duty cycle value in CCPR1L.

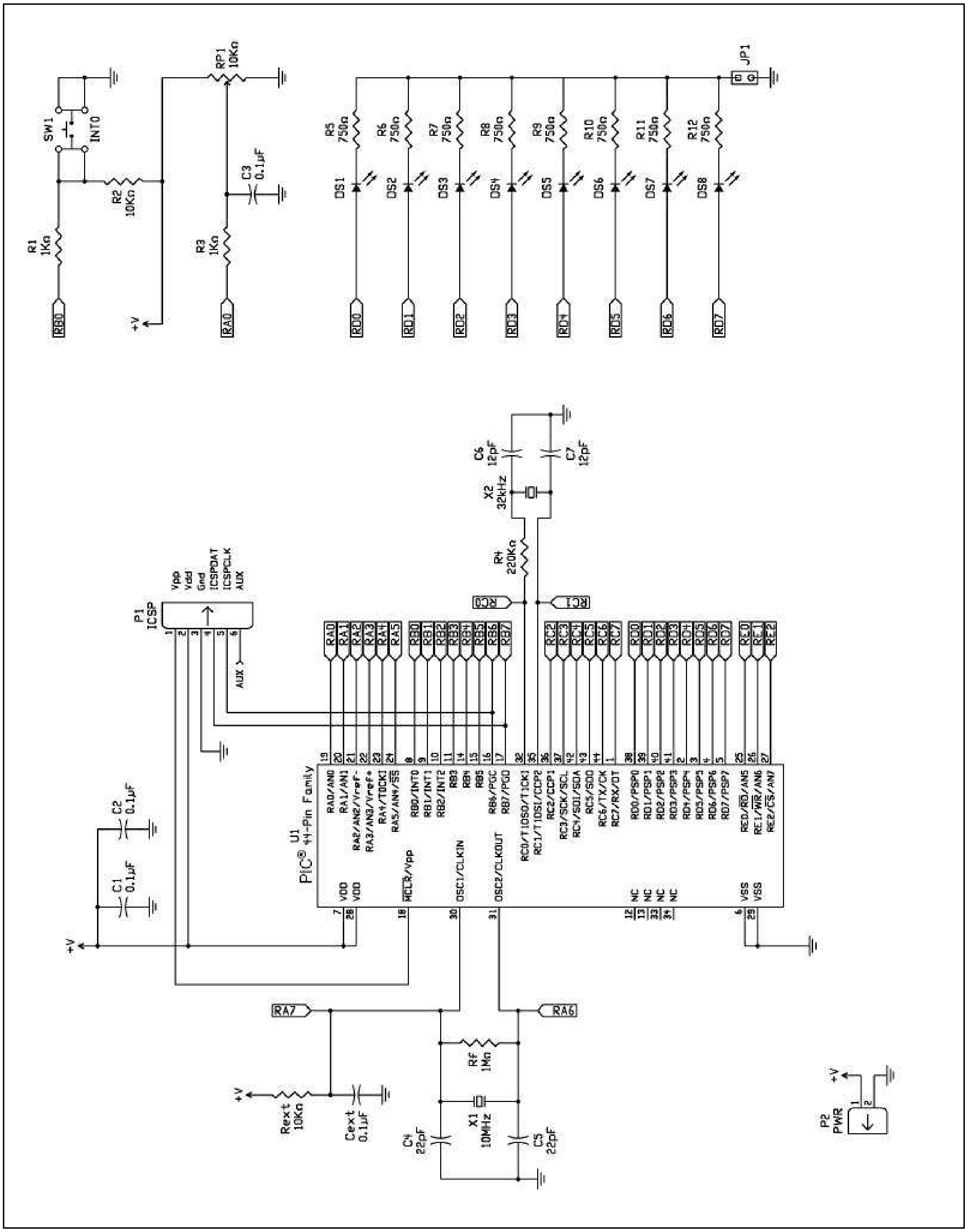
The second `do while` loop decreases the brightness over 2 seconds by reducing the duty cycle. As the duty cycle is decreased, the LED is on for shorter and shorter periods of time, making it appear dimmer.

3.12.4 Build and Run the Lesson 12 Code with PICkit 2 Debug Express

Build and program the lesson 12 project, then Run the application in the debugger. You will see the demo board LED 7 continuously get brighter then dimmer! If you have an oscilloscope available, connect a probe to one of the RD7 signal points on the demo board to see the changing the PWM waveform.

Appendix: 44-Pin Demo Board Schematics.

FIGURE A-1: 44-PIN DEMO BOARD SCHEMATIC DIAGRAM


SW1
INT0
R1 1KΩ
R2 10KΩ
RP1 10KΩ
C3 0.1µF
R3 1KΩ
RB0
+V
RA0
R5 750Ω
R6 750Ω
R7 750Ω
R8 750Ω
R9 750Ω
R10 750Ω
R11 750Ω
R12 750Ω
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
JP1
RD0
RD1
RD2
RD3
RD4
RD5
RD6
RD7
C6 12pF
C7 12pF
X2 32kHz
R4 220KΩ
RC0
RC1
P1 ICSP
Vpp
Vdd
Gnd
ICSPDAT
ICSPCLK
AUX
U1
PIC® 44-Pin Family
C1 0.1µF
C2 0.1µF
RA7
RA6
Rf 1MΩ
X1 10MHz
C4 22pF
C5 22pF
Rext 10KΩ
Cext 0.1µF
P2 PWR