MPLAB® C17 C COMPILER USER'S GUIDE

© 2002 Microchip Technology Inc.

Note the following details of the code protection feature on PICmicro® MCUs.

- The PICmicro family meets the specifications contained in the Microchip Data Sheet.
- Microchip believes that its family of PICmicro microcontrollers is one of the most secure products of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the PICmicro microcontroller in a manner outside the operating specifications contained in the data sheet. The person doing so may be engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable".
- Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our product.

If you have any further questions about this matter, please contact the local sales office nearest to you.

Information contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. No representation or warranty is given and no liability is assumed by Microchip Technology Incorporated with respect to the accuracy or use of such information, or infringement of patents or other intellectual property rights arising from such use or otherwise. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip. No licenses are conveyed, implicitly or otherwise, under any intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, KEELOQ, MPLAB, PIC, PICmicro, PICSTART and PRO MATE are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, microID, MXDEV, MXLAB, PICMASTER, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

dsPIC, dsPICDEM.net, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICC, PICDEM, PICDEM.net, rfPIC, Select Mode and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

Serialized Quick Turn Programming (SQTP) is a service mark of Microchip Technology Incorporated in the U.S.A.

All other trademarks mentioned herein are property of their respective companies.

© 2002, Microchip Technology Incorporated. Printed in the U.S.A., All Rights Reserved.

Printed on recycled paper.

DNV Certification, Inc. USA
ANSI • RAB QMS ACCREDITED
DNV
DNV MSC The Netherlands Accredited by the RvA
ISO 9001 / QS-9000 REGISTERED FIRM

Microchip received QS-9000 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona in July 1999 and Mountain View, California in March 2002. The Company's quality system processes and procedures are QS-9000 compliant for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, non-volatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001 certified.

Table of Contents

Preface 1

SECTION 1 – MPLAB C17 BASICS

Chapter 1. Compiler Overview and Installation
1.1 Introduction 9
1.2 Highlights 9
1.3 MPLAB C17 Description 9
1.4 Basic Functionality 10
1.5 Input/Output Files 12
1.6 Reserved Resources 12
1.7 Host Computer System Requirements 12
1.8 Compiler Versions 13
1.9 Install/Uninstall the Compiler 13

Chapter 2. Differences Between MPLAB C17 and ANSI C
2.1 Introduction 15
2.2 Highlights 15
2.3 MPLAB C17 vs. ANSI C 15
2.4 Components of a Basic MPLAB C17 Program 16
2.5 Keyword Differences 16
2.6 Statement Differences 17

Chapter 3. Using MPLAB C17 with MPLAB IDE
3.1 Introduction 23
3.2 Highlights 23
3.3 MPLAB Projects Overview 23
3.4 Using MPLAB C17 with MPLAB IDE 25
3.5 Code Development 36
3.6 Additional Options and Library Information 36

Chapter 4. Using MPLAB C17 on the Command Line
4.1 Introduction 37
4.2 Highlights 37
4.3 Command Line Overview 37
4.4 Using MPLAB C17 on the Command Line 39
4.5 Code Development 43
4.6 Library Information 43

© 2002 Microchip Technology Inc.

SECTION 2 – MPLAB C17 ADVANCED USAGE

Chapter 5. Runtime Environment

5.1 Introduction 47
5.2 Highlights 47
5.3 Code and Data Sections 47
5.4 Startup and Initialization 49
5.5 Memory Models 51
5.6 Locating Code 51
5.7 Locating Data 51
5.8 Software Stack 52
5.9 Software Stack Call Conventions 53
5.10 Function Call Conventions 53
5.11 Interrupt Support Macros 54

Chapter 6. Data Types

6.1 Introduction 59
6.2 Highlights 59
6.3 Data Representation 59
6.4 Integer 59
6.5 Floating Point 60

Chapter 7. Device Support Files

7.1 Introduction 61
7.2 Highlights 61
7.3 Processor Header File 61
7.4 Register Definitions File 62
7.5 Using SFRs 64

Chapter 8. Interrupts

8.1 Introduction 65
8.2 Highlights 65
8.3 Writing an Interrupt Service Routine 66
8.4 Writing the Interrupt Vector 67
8.5 Interrupt Service Routine Context Saving 68
8.6 Latency 68
8.7 Nesting Interrupts 68
8.8 Enabling/Disabling Interrupts 69

Chapter 9. Mixing Assembly Language and C Modules

9.1 Introduction 71
9.2 Highlights 71
9.3 Internal Assembler 71
9.4 Calling Conventions 72

© 2002 Microchip Technology Inc.

9.5 Mixing Assembly Language and C Variables and Functions 72
9.6 Using Inline Assembly Language 73

Chapter 10. Writing Efficient Code

10.1 Introduction 75
10.2 Highlights 75
10.3 Static Locals And Parameters 75
10.4 Optimization Tips 75

SECTION 3 – REFERENCES

Chapter 11. Enabling/Disabling Interrupts

11.1 Introduction 79
11.2 Highlights 79
11.3 Enabling Interrupts 79
11.4 Disabling Interrupts 102

Chapter 12. Implementation-Defined Behavior

12.1 Introduction 113
12.2 Highlights 113
12.3 Identifiers 113
12.4 Characters 113
12.5 Integers 114
12.6 Floating Point 114
12.7 Arrays and Pointers 115
12.8 Registers 115
12.9 Structures and Unions 115
12.10 Bit-Fields 115
12.11 Enumerations 115
12.12 Switch Statement 116
12.13 Preprocessing Directives 116

Chapter 13. MPLAB C17 Diagnostics

13.1 Introduction 117
13.2 Highlights 117
13.3 Errors 117
13.4 Warnings 121

SECTION 4 – APPENDICES

Appendix A. Reference Documents

A.1 Introduction 125
A.2 Highlights 125
A.3 C Standards Information 125
A.4 General C Information 125

Appendix B. Example Programs

B.1 Introduction 127
B.2 Highlights 127
B.3 Overview of Example Files 127
B.4 Example Details 127

Appendix C. ASCII Character Set 129

Glossary 131

Index 147

Worldwide Sales and Service 154

© 2002 Microchip Technology Inc.

Preface

INTRODUCTION

The purpose of this user's guide is to help you get up and running with Microchip's MPLAB C17 C Compiler.

This manual is written with the intent that you are at least familiar with using the C programming language. If not, check Appendix A for a list of reference books that cover C programming.

HIGHLIGHTS

Items discussed in this chapter are:

- About this Guide
- Warranty Registration
- Recommended Reading
- Troubleshooting
- Microchip On-Line Support
- Customer Change Notification Service
- Customer Support

ABOUT THIS GUIDE

Document Layout

This document describes how to use MPLAB C17 to write C code for PICmicro® microcontroller applications. For a detailed discussion about basic MPLAB IDE functions, refer to the *MPLAB IDE User's Guide (DS51025)*.

This user's guide layout is as follows:

Section 1 – MPLAB C17 Basics

- **Chapter 1: Compiler Overview and Installation** – provides an overview of the MPLAB C17 C compiler, including compiler operation, input/output files and resource requirements. Also gives instructions on how to install or uninstall the compiler onto your system.
- **Chapter 2: Differences Between MPLAB C17 and ANSI C** – describes how MPLAB C17 differs from standard ANSI C. Compares MPLAB C17 and ANSI C and then highlights keyword, statement and standard function differences.
- **Chapter 3: Using MPLAB C17 with MPLAB IDE** – describes how to use MPLAB C17 with the MPLAB IDE v5.xx or below. Code development and other hardware tools are available when using MPLAB IDE.
- **Chapter 4: Using MPLAB C17 on the Command Line** – describes how to use the MPLAB C17 compiler from the command line interface. More compiler options are available on the command line.

© 2002 Microchip Technology Inc.

Section 2 – MPLAB C17 Advanced Usage

- **Chapter 5: Runtime Environment** – describes the MPLAB C17 runtime environment. Includes information on code and data sections, startup and initialization, memory models and the software stack.
- **Chapter 6: Data Types** – describes MPLAB C17 data types.
- **Chapter 7: Device Support Files** – discusses the device support files used by MPLAB C17, namely processor header files and register definitions files.
- **Chapter 8: Interrupts** – describes how to use interrupts. Detailed interrupt usage may be found in the reference section.
- **Chapter 9: Mixing Assembly Language and C Modules** – provides guidelines to using C with assembly language modules.
- **Chapter 10: Writing Efficient Code** – provides guidelines to writing efficient MPLAB C17 code.

Section 3 – References

- **Chapter 11: Enabling/Disabling Interrupts** – detailed information on how to enable and disable interrupts.
- **Chapter 12: Implementation-Defined Behavior** – details MPLAB C17 specific parameters described as implementation-defined in the ANSI standard.
- **Chapter 13: MPLAB C17 Diagnostics** – lists errors and warnings generated by MPLAB C17.

Section 4 – Appendices

- **Appendix A: Reference Documents** – gives references that may be helpful in programming with MPLAB C17.
- **Appendix B: Example Programs** – discusses the MPLAB C17 examples included in the `examples` directory.
- **Appendix C: ASCII Character Set** – contains the ASCII character set.
- **Glossary** – A glossary of terms used in this guide.
- **Index** – Cross-reference listing of terms, features and sections of this document.
- **Worldwide Sales and Service** – gives the address, telephone and fax numbers for Microchip Technology Inc. sales and service locations throughout the world.

© 2002 Microchip Technology Inc.

Conventions Used in this Guide

This manual uses the following documentation conventions:

Table: Documentation Conventions

Description	Represents	Examples
Code (Courier font):		
Plain characters	Sample code Filenames and paths	`#define START` `c:\autoexec.bat`
Angle brackets: < >	Variables	`<label>, <exp>`
Square brackets [ ]	Optional arguments	`MPASMWIN [main.asm]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; An OR selection	`errorlevel {0|1}`
Lower case characters in quotes	Type of data	`"filename"`
Ellipses...	Used to imply (but not show) additional text that is not relevant to the example	`list ["list_option..., "list_option"]`
0xnnn	A hexadecimal number where n is a hexadecimal digit	`0xFFFF, 0x007A`
Italic characters	A variable argument; it can be either a type of data (in lower case characters) or a specific example (in uppercase characters).	`char isascii (char, ch);`
Interface (Arial font):		
Underlined, italic text with right arrow	A menu selection from the menu bar	*File > Save*
Bold characters	A window or dialog button to click	**OK**, **Cancel**
Characters in angle brackets < >	A key on the keyboard	<Tab>, <Ctrl-C>
Documents (Arial font):		
Italic characters	Referenced books	*MPLAB IDE User's Guide*

Documentation Updates

All documentation becomes dated, and this user's guide is no exception. Since MPLAB IDE, MPLAB C17 and other Microchip tools are constantly evolving to meet customer needs, some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site to obtain the latest documentation available.

Documentation Numbering Conventions

Documents are numbered with a "DS" number. The number is located on the bottom of each page, in front of the page number. The numbering convention for the DS Number is: DSXXXXXA,

where:

XXXXX	=	The document number.
A	=	The revision level of the document.

WARRANTY REGISTRATION

Please complete the enclosed Warranty Registration Card and mail it promptly. Sending in your Warranty Registration Card entitles you to receive new product updates. Interim software releases are available at the Microchip web site.

RECOMMENDED READING

This user's guide describes how to use MPLAB C17 C Compliler. For more information on included libraries and precompiled object files for the compilers, the operation of MPLAB IDE and the use of other tools, the following are recommended reading.

README.C17

For the latest information on using MPLAB C17 C Compiler, read the README.C17 file (ASCII text) included with the software. This README file contains update information that may not be included in this document.

README.XXX

For the latest information on other Microchip tools (MPLAB IDE, MPLINK™ linker, etc.), read the associated README files (ASCII text file) included with the MPLAB IDE software.

MPLAB C17 C Compiler Libraries (DS51296)

Reference guide for MPLAB C17 libraries and precompiled object files. Lists all library functions with a detailed description of their use.

MPLAB IDE User's Guide (DS51025)

Comprehensive guide that describes installation and features of Microchip's MPLAB Integrated Development Environment (IDE), as well as the editor and simulator functions in the MPLAB IDE environment.

MPASM™ User's Guide with MPLINK™ and MPLIB™ (DS33014)

This user's guide describes how to use the Microchip PICmicro MCU MPASM assembler, the MPLINK object linker and the MPLIB object librarian.

Technical Library CD-ROM (DS00161)

This CD-ROM contains comprehensive application notes, data sheets and technical briefs for all Microchip products. To obtain this CD-ROM, contact the nearest Microchip Sales and Service location (see back page).

Microchip Web Site

The Microchip web site (www.microchip.com) contains a wealth of documentation. Individual data sheets, application notes, tutorials and user's guides are all available for easy download. All documentation is in Adobe™ Acrobat (pdf) format.

Microsoft® Windows® Manuals

This manual assumes that users are familiar with the Microsoft Windows operating system. Many excellent references exist for this software program, and should be consulted for general operation of Windows.

TROUBLESHOOTING

See the `README` files for information on common problems not addressed in this user's guide.

© 2002 Microchip Technology Inc.

MICROCHIP ON-LINE SUPPORT

Microchip provides on-line support on the Microchip web site at:

www.microchip.com

A file transfer site is also available by using an FTP service connecting to:

ftp://ftp.microchip.com

The web site and file transfer site provide a variety of services. Users may download files for the latest development tools, data sheets, application notes, user's guides, articles and sample programs. A variety of Microchip specific business information is also available, including listings of Microchip sales offices and distributors. Other information available on the web site includes:

- Latest Microchip press releases
- Technical support section with FAQs
- Design tips
- Device errata
- Job postings
- Microchip consultant program member listing
- Links to other useful web sites related to Microchip products
- Conferences for products, development systems, technical information and more
- Listing of seminars and events

CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip started the customer notification service to help customers stay current on Microchip products with the least amount of effort. Once you subscribe, you will receive email notification whenever we change, update, revise or have errata related to your specified product family or development tool of interest.

Go to the Microchip web site (www.microchip.com) and click on Customer Change Notification. Follow the instructions to register.

The Development Systems product group categories are:

- Compilers
- Emulators
- In-Circuit Debuggers
- MPLAB IDE
- Programmers

Here is a description of these categories:

Compilers - The latest information on Microchip C compilers and other language tools. These include the MPLAB C17, MPLAB C18 and MPLAB C30 C Compilers; MPASM and MPLAB ASM30 assemblers; MPLINK and MPLAB LINK30 linkers; and MPLIB and MPLAB LIB30 librarians.

Emulators - The latest information on Microchip in-circuit emulators. This includes the MPLAB ICE 2000.

In-Circuit Debuggers - The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD and MPLAB ICD 2.

MPLAB - The latest information on Microchip MPLAB IDE, the Windows Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB SIM simulator, MPLAB IDE Project Manager and general editing and debugging features.

Programmers - The latest information on Microchip device programmers. These include the PRO MATE II device programmer and PICSTART Plus development programmer.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributors
- Local Sales Office
- Field Application Engineers (FAEs)
- Corporate Applications Engineers (CAEs)
- Systems Information and Upgrade Hot Line

Customers should call their distributor or field application engineer (FAE) for support. Local sales offices are also available to help customers. See the last page of this document for a listing of sales offices and locations.

Corporate applications engineers (CAEs) may be contacted at (480) 792-7627.

Systems Information and Upgrade Line

The Systems Information and Upgrade Information Line provides system users with a listing of the latest versions of all of Microchip's development systems software products. Plus, this line provides information on how customers can receive the most current upgrade kits. The Information Line Numbers are:

1-800-755-2345 for U.S. and most of Canada.

1-480-792-7302 for the rest of the world.

© 2002 Microchip Technology Inc.

Section 1 – MPLAB C17 Basics

Chapter 1. Compiler Overview and Installation 9
Chapter 2. Differences Between MPLAB C17 and ANSI C 15
Chapter 3. Using MPLAB C17 with MPLAB IDE 23
Chapter 4. Using MPLAB C17 on the Command Line 37

© 2002 Microchip Technology Inc.

Chapter 1. Compiler Overview and Installation

1.1 INTRODUCTION

This chapter provides an overview of the MPLAB C17 C compiler and how to install the compiler on your computer.

1.2 HIGHLIGHTS

This chapter covers the following topics:

- MPLAB C17 Description
- Basic Functionality
- Input/Output Files
- Reserved Resources
- Host Computer System Requirements
- Compiler Versions
- Install/Uninstall the Compiler

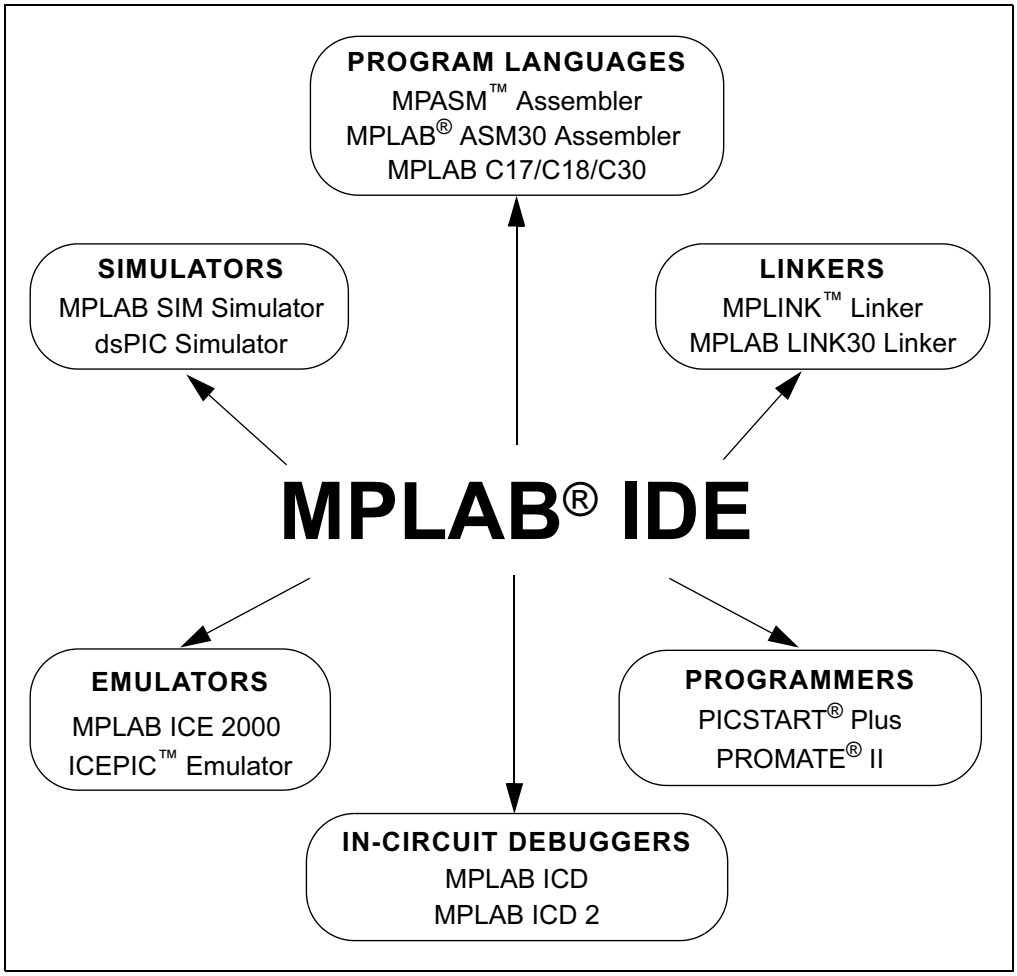
1.3 MPLAB C17 DESCRIPTION

The MPLAB C17 compiler is a full-featured ANSI C compiler for Microchip's PIC17 PICmicro microcontrollers (MCU). The compiler is fully compatible with Microchip's MPLAB Integrated Development Environment (IDE) (Figure 1-1), allowing source-level debugging with both the MPLAB ICE in-circuit emulator and the MPLAB SIM simulator. MPLAB IDE provides a convenient, project-oriented development environment that reduces development time.

MPLAB C17 has implemented extensions to the C language to provide specific support for Microchip's PICmicro MCU peripherals. The C libraries include: A/D converter, Character Classification, External LCD, I^2C™, Input Capture, Interrupt Support Macros, I/O Port, Memory/String Manipulation, Number/Text Conversion, Pulse Width Modulation, RESET, Relay, Software I^2C, Software SPI™, Software USART, SPI, Timers and USART.

© 2002 Microchip Technology Inc.

FIGURE 1-1: DEVELOPMENT SYSTEM ARCHITECTURE

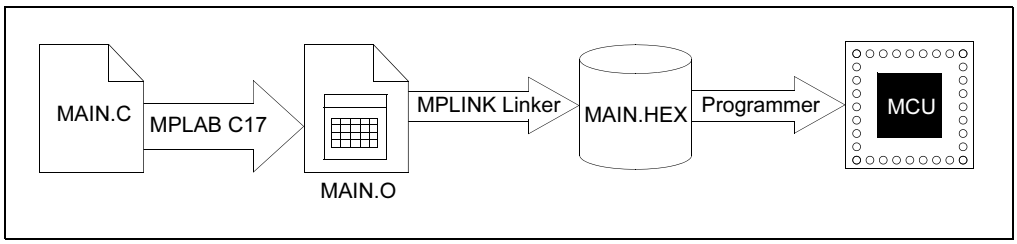


1.4 BASIC FUNCTIONALITY

MPLAB C17 generates object code from C source code. This object code is then input into Microchip's MPLINK linker to form the final executable code. A single C source file may be compiled into a single executable as shown in Figure 1-2, or it can be linked with other separately assembled or compiled modules as shown in Figure 1-3. Related modules can also be grouped and stored together in a library using Microchip's MPLIB Librarian (Figure 1-4). Required libraries can be specified at link time, and only the modules that are needed will be included in the final executable.

For more information on MPLINK linker and MPLIB librarian operation, please refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

FIGURE 1-2: GENERATING EXECUTABLE CODE FROM ONE OBJECT MODULE



© 2002 Microchip Technology Inc.

FIGURE 1-3: GENERATING EXECUTABLE CODE FROM OBJECT MODULES

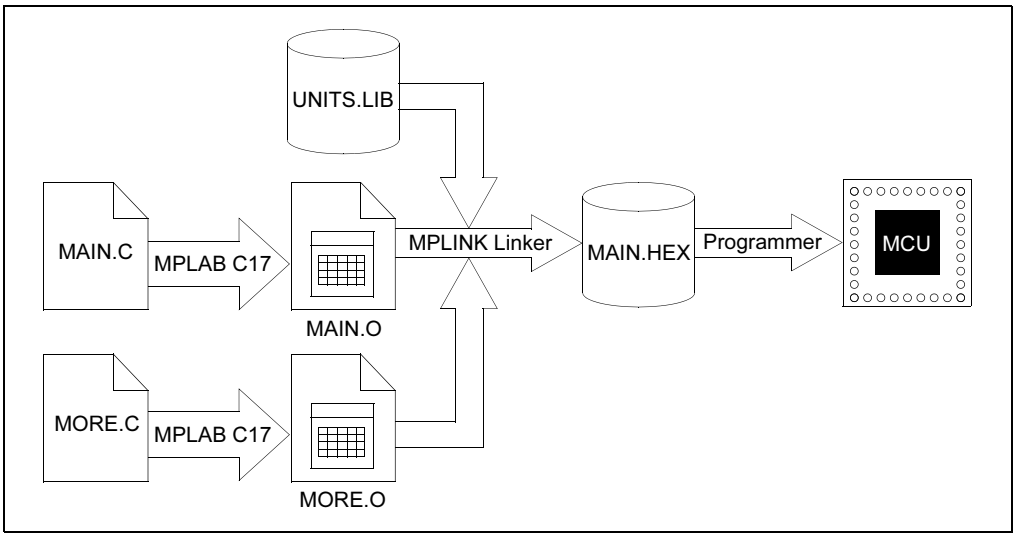
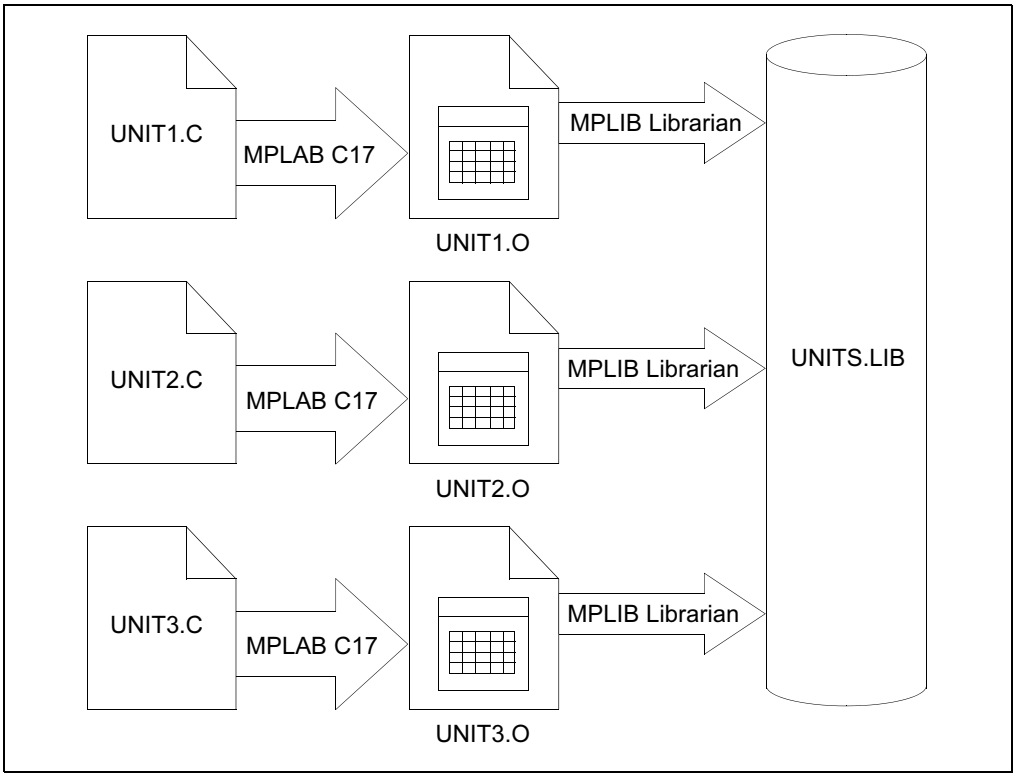


FIGURE 1-4: CREATING A REUSABLE OBJECT LIBRARY



1.5 INPUT/OUTPUT FILES

These are the default file extensions used by MPLAB C17.

TABLE 1-1: MPLAB C17 DEFAULT EXTENSIONS

Extension	Purpose
.c	Default source file extension input to MPLAB C17: <source_name>.c
.err	Output extension from MPLAB C17 for error files: <source_name>.err
.o	Output extension from MPLAB C17 for object files: <source_name>.o

1.5.1 Source Code Format (.c)

The source code file can be created using any ASCII text file editor. It should conform to C language programming guidelines. For information on how to program using C, please refer to Appendix A.

1.5.2 Error File Format (.err)

By default MPLAB C17 generates an error file. This file can be useful when debugging your code. The MPLAB IDE automatically opens this file in the case of an error. The format of the messages in the error file is:

```
<type>[<number>] <file> <line> <description>
```

For example:

```
Error[113] C:\prog.c 7 : Symbol not previously defined
 (start)
```

See the appendices for descriptions of error messages generated.

1.5.3 Object File Format (.o)

Object files are the relocatable code produced from source files.

1.6 RESERVED RESOURCES

The following are PICmicro MCU resource impacts from the compiler:

- **FSR0**: Can be used, but compiler may use also. Don't expect value to stay the same.
- **FSR1**: Reserved for compiler use.
- **PRODH, PRODL**: Can be used, but compiler may use also. Don't expect value to stay the same.
- **TBLPTRH, TBLPTRL, TBLAT**: Can be used, but compiler may use also. Don't expect value to stay the same.

1.7 HOST COMPUTER SYSTEM REQUIREMENTS

MPLAB C17 requires:

- PC-compatible 386 or better class system
- 16 MB memory (32 MB recommended)
- 5 MB hard disk space (10 MB recommended)
- MS-DOS® PC-DOS version 5.0 or greater, or Microsoft Windows® operating system (version 3.x or greater).

© 2002 Microchip Technology Inc.

1.8 COMPILER VERSIONS

There are two versions of MPLAB C17:

- a DOS-extended or Windows 3.x version, `mcc17d.exe`
- a Windows 32-bit version (Windows 95 or greater), `mcc17.exe`

You can use both versions with MPLAB IDE; however, the Windows 32-bit version is recommended.

1.9 INSTALL/UNINSTALL THE COMPILER

If you are going to use MPLAB C17 with the MPLAB IDE, install the MPLAB IDE first. To install MPLAB C17, enter your Windows operating system, run the file `SETUP.EXE` on the CD-ROM, and follow the prompts.

The install program will use or create the directory you chose in the setup program. Then, it will install the MPLAB C17 components into seven subdirectories:

- `bin` – executable versions
- `doc` – help files
- `examples` – source code examples, with example-specific header, linker and batch files
- `h` – general header files
- `lib` – library and pre-compiled object files
- `lkr` – linker script files
- `src` – source code for files found in `lib` directory

In addition, the MPLAB C17 install will create an environment variable, `MCC_INCLUDE`, in your `AUTOEXEC.BAT` file. The `MCC_INCLUDE` environment variable specifies the directories to search for included files. For more information, refer to the `#include` directive. The install program will also add the compiler `bin` directory to your `PATH` so you can run the compiler from any other directory.

To uninstall the compiler:

1. From the Start menu, select *Settings > Control* Panel to display the Control Panel directory.
2. Double-click the Add/Remove Programs icon. Scroll down the list and locate the program you want to remove. Click **Remove**.

NOTES:

© 2002 Microchip Technology Inc.

Chapter 2. Differences Between MPLAB C17 and ANSI C

2.1 INTRODUCTION

This chapter discusses the differences between MPLAB C17 and ANSI C.

2.2 HIGHLIGHTS

Items discussed in this chapter are:

- MPLAB C17 vs. ANSI C
- Components of a Basic MPLAB C17 Program
- Keyword Differences
- Statement Differences
- Oddities of Standard Functions

2.3 MPLAB C17 vs. ANSI C

Most C programmers have gained their experience programming C on computers where there was an operating system to handle such things as memory management, input/output, interdevice communications, etc. Microcontrollers, by their very nature, do not have the memory overhead for an operating system. Therefore, the compiler expects the user to implement memory allocation, I/O operation through a peripheral, and other specialized tasks. Libraries and precompiled object files are available with MPLAB C17 to aid the programmer in this endeavor.

An MPLAB C17 program is a collection of declarations, statements, comments and preprocessor directives that typically do the following:

- Declare data structures
- Allocate data space
- Evaluate expressions
- Perform program control operations
- Control PICmicro MCU peripherals

Additionally, after source code is compiled, it must be programmed into a device. In the device environment, RAM is in an undefined state on power-up. The program must take care of initializing any variables that are set in the code. This is accomplished by storing the variable values in program memory and then moving them to RAM before the `main()` function executes. There are other `main()` pre-execution items that may be necessary, such as setting up a software stack. These specialized items may be written in C or assembly code. In either case, the programmer must decide what is needed.

2.4 COMPONENTS OF A BASIC MPLAB C17 PROGRAM

The following is the shell for a basic MPLAB C17 source file, highlighting the elements of the program:

```
#include <p17CXX.h>      ← preprocessor directive
void function1(void);    ← prototyped function
void main(void)          ← main routine
{
 /* User source code here */
}
void function1(void)     ← function definition
{
 /* User function code here */
}
```

The first line is a preprocessor directive that includes the processor definition file (7.3 "Processor Header File"). This file defines processor-specific information such as special function registers.

> **Note:** `p17CXX.h` includes proper processor-specific header file based on the processor selected on the command line.

The next line is a declaration of a function (2.6.9 "Functions") that will be used in the main routine (`function1`). Placing the function declaration here is called prototyping. The function itself may then be defined after the main routine. Another option is to place the entire function definition in the prototype location.

Finally, the main routine is defined, with the appropriate source code between the braces. Note that the main routine is itself a function.

2.5 KEYWORD DIFFERENCES

The ANSI C standard defines 32 keywords for use in the C language. Typically, C compilers add keywords that take advantage of the processor's architecture. The following table shows the ANSI C and the MPLAB C17 keywords, where MPLAB C17 keywords are shown in bold.

TABLE 2-1: ANSI C AND MPLAB C17 KEYWORDS

_asm	double	long	struct
_endasm	else	**near**	switch
auto	enum	**ram**	typedef
break	extern	register**	union
case	**far**	return	unsigned
char	float	**rom**	void
const	for	short	volatile
continue	goto	signed	while
default	if	sizeof	
do	int	static	

** has no effect in MPLAB C17

© 2002 Microchip Technology Inc.

2.6 STATEMENT DIFFERENCES

There are differences between how MPLAB C17 uses some statements and how these statements would be implemented under ANSI C. These specialized MPLAB C17 statements are:

- `#include` *`filename`*
- `#pragma` Statements
- Constants
- Variables
- Storage Classes
- Functions
- Operators
- `switch` Statement
- Initializing Arrays
- Pointers
- Structures
- Bit-fields

2.6.1 #include *filename*

Include the file *filename* into the MPLAB C17 program. Usually at least the processor header file is included, so that register and pin names may be used in code as opposed to addresses.

When "`filename`" is used, MPLAB C17 looks for the file in the current directory and then in the directories specified by the current include search path, which refers to the environment variable `MCC_INCLUDE` and command-line option '-i'.

When <`filename`> is used, MPLAB C17 looks for the file in the directories specified by the current include search path.

2.6.2 #pragma interrupt fname

2.6.2.1 DESCRIPTION

Declare a function to be an interrupt function. This pragma must come before the function definition, but may come after a prototype. The compiler will generate a separate temporary storage section dedicated to the function.

See Chapter 9 for more details on interrupt handling.

2.6.2.2 SYNTAX

```
interrupt-directive:
    #pragma interrupt function-name [section-name]
                      new-line
```

2.6.3 #pragma list / #pragma nolist

2.6.3.1 DESCRIPTION

The `#pragma list` directive turns on list file generation for all code following the directive. The `#pragma nolist` directive turns off list file generation for all code following the directive.

2.6.3.2 SYNTAX

```
list-directive:
    #pragma list new-line
    #pragma nolist new-line
```

2.6.4 #pragma sectiontype

2.6.4.1 DESCRIPTION

The section declaration family of pragmas changes the section into which MPLAB C17 will allocate data of the associated type. Optionally, the section may be allocated at an absolute address.

A section declaration with no name resets the allocation of data of the associated type to the default section for the current module.

A data section qualified as `shared` will be located in a `SHAREBANK` by the linker. Similarly, a data section qualified as `access` will be located in an `ACCESSBANK` by the linker.

Specifying a section name which has been previously declared causes MPLAB C17 to resume allocating data of the associated type into the specified section. The section qualifiers must match the previous declaration.

For `udata` and `idata` sections in MPLAB C17, the data section type, SFR or GPR, and a bank number may be optionally specified instead of an absolute address. This is functionally equivalent to specifying a `varlocate` pragma with the same information for each symbol declared in the section. Like `varlocate`, this qualifier provides information to the compiler only and is not enforced by the linker; therefore, care should be exercised in its use.

> **Note:** Logical sections are used to specify which of the defined memory regions should be used for a portion of source code. For more on sections, refer to the MPLINK linker section of the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

2.6.4.2 SYNTAX

```
section-directive:
    #pragma udata [data-qualifier-list] [section-name
                   [location]] new-line
    #pragma idata [data-qualifier-list] [section-name
                   [location]] new-line
    #pragma romdata [overlay] [section-name] new-line
    #pragma code [overlay] [section-name] new-line
data-qualifier:
    shared
    overlay
location:
    = address
    gpr bank-number
    sfr bank-number
```

2.6.4.3 EXAMPLE

Declare a section for udata allocation at address `0x120`. The linker will enforce that the section will be located at address `0x120`.

```
#pragma udata myNewDataSection = 0x120
```

Resume allocation of romdata into the default section.

```
#pragma romdata
```

Declare a new code section at address `0x8000`.

```
#pragma code myExternalCodeSection=0x8000
```

2.6.4.4 SEE ALSO

```
#pragma varlocate
```


© 2002 Microchip Technology Inc.

2.6.5 #pragma varlocate n name
#pragma varlocate {gpr | sfr} name

2.6.5.1 DESCRIPTION

The `varlocate` pragma tells the compiler where a variable will be located at link time, enabling the compiler to perform more efficient bank switching. The bank may be specified (`n`) or the GPR or SFR address range may be specified.

`varlocate` specifications are not enforced by the compiler or linker. The sections which contain the variables should be assigned explicitly in the linker script, or via absolute sections in the module(s) where they are defined, into the correct bank.

2.6.5.2 SYNTAX

```
variable-locate-directive:
    #pragma varlocate bank variable-name new-line
    #pragma varlocate [bank-reg] variable-name new-line
#pragma varlocate section-name variable-name new-line
```

2.6.6 Constants

The MPLAB C17 compiler supports the usual four different kinds of constants:

- Integers
- Floating-point numbers
- Characters
- Strings

See Chapter 6 for more on data types.

2.6.7 Variables

The MPLAB C17 compiler supports the standard integer and floating-point numeric types. A plain `char` is signed by default. See Chapter 6 for more on data types.

The ranges of the standard integer types are documented in Table 2-2.

TABLE 2-2: STANDARD INTEGER TYPES

Name	Value	Meaning
CHAR_BIT	8	Width of `char` type, in bits
SCHAR_MIN	-128	Minimum value of `signed char`
SCHAR_MAX	127	Maximum value of `signed char`
UCHAR_MAX	255	Maximum value of `unsigned char`
SHRT_MIN	-32,768	Minimum value of `short int`
SHRT_MAX	32,767	Maximum value of `short int`
USHRT_MAX	65,535	Maximum value of `unsigned short`
INT_MIN	-32,768	Minimum value of `int`
INT_MAX	32,767	Maximum value of `int`
UINT_MAX	65,535	Maximum value of `unsigned int`
LONG_MIN	-2,147,483,648	Minimum value of `long int`
LONG_MAX	2,147,483,647	Maximum value of `long int`
ULONG_MAX	4,294,967,295	Maximum value of `unsigned long`
CHAR_MIN	If type `char` is signed by default, then SCHAR_MIN else 0.	Minimum value or `char`
CHAR_MAX	If type `char` is signed by default then SCHAR_MAX, else UCHAR_MAX.	Maximum value of `char`

The MPLAB C17 compiler supports the `float` and `double` types, both of which are 32-bit floating-point types. The ranges of the floating-point type are documented in Table 2-3.

TABLE 2-3: MPLAB C17 `Float` AND `DOUBLE` Types

Name	Value	Meaning
FLT_MIN DBL_MIN	1.7549435E-38	Minimum normalized positive number
FLT_MAX DBL_MAX	6.80564693E+38	Maximum representable finite number

The sizes of basic types are documented in Table 2-4.

TABLE 2-4: BASIC SIZES

Type	Size in Bits
char, signed char, unsigned char	8
short, signed short, unsigned short	16
int, signed int, unsigned int	16
short long, signed short long, unsigned short long	24
long, signed long, unsigned long	32
float	32
double	32

2.6.8 Storage Classes

In addition to the standard storage classes (`auto`, `extern`, `register`, `static` and `typedef`) the MPLAB C17 compilers include four new classes:

- far Paging/banking of data required.
- near No paging/bank of data required.
- rom Locate the object in program memory.
- ram Locate the object in data memory.

The compilers ignore the `register` storage class specifier.

2.6.9 Functions

Function operation in MPLAB C17 is discussed in the following sections. For information on function call conventions, see 5.10 "Function Call Conventions".

2.6.9.1 PASSING ARGUMENTS TO FUNCTIONS

Function parameters can have storage class `auto` or `static`. An `auto` parameter is placed on the software stack, enabling reentrancy and a `static` parameter is allocated globally, enabling direct access and, therefore, smaller code. See 10.3 "Static Locals And Parameters" for more information on static parameters.

If the first parameter to a function is static and is 8 bits wide, the argument will be passed to the function in `PRODL`. If it is static and 16 bits wide, the argument will be passed in `PROD`.

MPLAB C17 uses a software stack for passing variables into functions and for returning values from functions. This makes it possible to support quite complex functions and allows recursive functions, but there is some overhead in managing the software stack. When compiling, the compiler will examine the function and only include the appropriate level of stack support code.

© 2002 Microchip Technology Inc.

2.6.9.2 RETURNING VALUES FROM FUNCTIONS

If the value being returned is 8 bits wide, it is returned in `WREG`. If it is 16 bits wide, it is returned in the `WREG/FSR1` pair. Otherwise, it is returned on the software stack.

2.6.10 Operators

The MPLAB C17 compiler supports all of the standard C operators.

2.6.11 switch Statement

A `switch` statement is functionally equivalent to multiple `if-else` statements.

The switch statement has two limitations:

- The `switch` expression must be an 8-bit integer data type
- The `case` values must be constant values.

2.6.12 Initializing Arrays

EXAMPLE 2-1: INITIALIZING ARRAYS

Because the PICmicro MCU family of microcontrollers uses separate program memory and data memory address busses in their design, MPLAB C17 requires ANSI extensions to distinguish between data located in ROM and data located in RAM. The ANSI/ISO C standard allows for code and data to be in separate address spaces, but this is not sufficient to locate data in the code space as well. To this purpose, MPLAB C17 introduces the `rom` and `ram` qualifiers. Syntactically, these qualifiers bind to identifiers just as the `const` and `volatile` qualifiers do in strict ANSI C.

The primary use of ROM data is for static strings. In keeping with this, MPLAB C17 automatically places all string literals in ROM. This type of a string literal is “array of char located in ROM.”

> **Note:** At this time, you should use manual pointer arithmetic. See the file `README.C17` for more information.

When using MPLAB C17, a string table in program memory can be declared as:

```
rom const char table[][20] =   { “string 1”, “string 2”,
                                 “string 3”, “string 4” };
rom const char *rom table2[] = { “string 1”, “string 2”,
                                 “string 3”, “string 4” };
```

The declaration of `table` declares an array of four strings that are each 20 characters long, and so takes 40 words of program memory. `table2` is declared as an array of pointers to ROM. The `rom` qualifier after the * places the array of pointers in ROM as well. All of the strings in `table2` are 3 words long, and the array is four elements long, so `table2` takes 3*4 = 12 words of program memory. Accesses to `table2` may be less efficient than accesses to `table`, however, because of the additional level of indirection required by the pointer.

An important consequence of the separate ROM and RAM address spaces for MPLAB C17 is that pointers to data in ROM and pointers to data in RAM are not compatible. That is, two pointer types are not compatible unless they point to objects of compatible types and the objects they point to are located in the same address space. For example, a pointer to a string in ROM and a pointer to a string in RAM are not compatible because they refer to different address spaces. To copy data from ROM to RAM, an explicit copy is required. For simple types, this entails only a simple assignment, but for arrays and other complex data-types it may require more.

For example, a function to copy a string from ROM to RAM could be written as follows.

```
void str2ram(static char *dest, static char rom *src)
{
 while( (*dest++ = *src++) != '\0' )
    ;
} /* end str2ram */
```

As an example, the following code will send a ROM string to USART1 on a PIC17C756 using the PICmicro MCU C libraries. The library function to send a string to the USART, `putsUSART1(const char *str)`, takes a pointer to a string as its argument, but that string must be in RAM.

Modify the library routine to read from a ROM string.

```
/* The only changes required to the library routine is to change the
name so the new routine does not conflict with the original routine and
to add the rom qualifier to the parameter.
*/
void putrsUSART1_rom( static const rom char *data )
{
 do      /* Send characters up to the null */
 {       /* Write a byte to the UASRT */
    while(BusyUSART1());
    putcUSART1(*data);
 } while(*data++);
} /* end putrsUSART1_rom */
```

2.6.13 Pointers

RAM pointers are either 8 or 16-bit, depending on whether they point to banked (far) or unbanked (near) RAM.

Pointer arithmetic is affected by the ROM paging and RAM banking of the PICmicro MCU. Pointers are assumed to be RAM pointers unless declared as ROM.

```
rom int *p;      /* ROM pointer */
char *q;         /* RAM pointer (default) */
ram char *r;     /* RAM pointer */
                 /* (explicitly declared) */
```

2.6.14 Structures

User-defined data constructs are included in the symbolic information file from the linker.

For MPLAB C17, structures located in program memory must have all elements word aligned.

MPLAB C17 supports anonymous structures.

2.6.15 Bit-fields

Bit-fields allow the specification of integer-type members of a struction, which are the specified number of bits in size. Bit-fields cannot cross byte boundaries and, therefore, cannot be greater than 8 bits in size.

© 2002 Microchip Technology Inc.

Chapter 3. Using MPLAB C17 with MPLAB IDE

3.1 INTRODUCTION

This chapter discusses how to use MPLAB C17 with MPLAB IDE.

3.2 HIGHLIGHTS

This chapter includes:

- MPLAB Projects Overview
- Using MPLAB C17 with MPLAB IDE
- Code Development
- Additional Options and Library Information

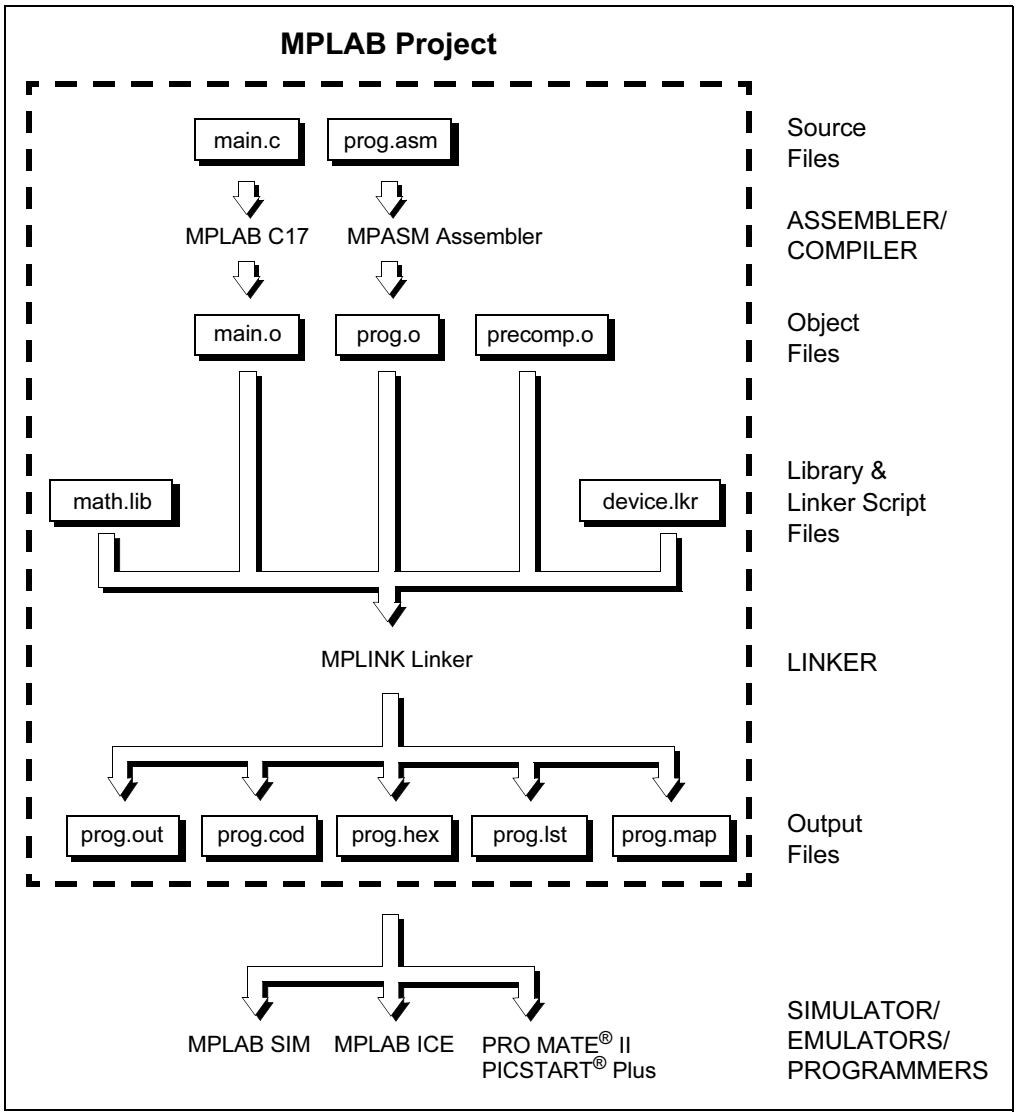
3.3 MPLAB PROJECTS OVERVIEW

MPLAB C17 may be used with MPLAB IDE, running under Windows 3.x (`mcc17d.exe`) or running under Windows 9x, Windows NT or Window 2000 (`mcc17.exe`).

MPLAB C17 is one of several tools that work with MPLAB IDE. These tools are used as part of an MPLAB Project. A project in MPLAB IDE is the group of files needed to build an application, along with their associations to various build tools. See the *MPLAB IDE User's Guide* (DS51025) for more information on MPLAB IDE and MPLAB IDE Projects.

Figure 3-1 shows a generic MPLAB Project using the MPLAB C17 compiler tool.

FIGURE 3-1: AN MPLAB PROJECT WITH MPLAB C17 – FILES AND ASSOCIATED TOOLS



In this MPLAB Project, the source file `main.c` is associated with the MPLAB C17 compiler. MPLAB IDE will use this information to generate an object file (`main.o`) for input into the linker (MPLINK linker).

An assembly source file (`prog.asm`) is shown also with its associated assembler (MPASM assembler). MPLAB IDE will use this information to generate the object file `prog.o` for input into MPLINK linker. See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on using the assembler.

In addition, precompiled object files (`precomp.o`) may be included in a project, with no associated tool required. Types of precompiled object files that are generally required in a project are listed below:

- Start up code
- Initialization code
- Interrupt service routines
- Register definitions

© 2002 Microchip Technology Inc.

Precompiled object files are often device and/or memory model dependent. For more information on available precompiled object files, see the *MPLAB C17 C Compiler Libraries* (DS51296).

Some library files are available with the compiler. Others may be built outside the project using the librarian tool (MPLIB librarian). See the *MPASM User's Guide with MPLINK and MPLIB* (DS33014) for more information on using the librarian. For more information on available libraries, see the *MPLAB C17 C Compiler Libraries* (DS51296).

The object files, along with library files and a linker script file (`device.lkr`) are used to generate the project output files via the linker (MPLINK linker). See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on linker script files and using the linker.

The main output file generated by MPLINK linker is the **HEX file** (`prog.hex`), used by simulators (MPLAB SIM), emulators (MPLAB ICE and PICMASTER emulator) and programmers (PRO MATE II and PICSTART Plus). The other output files are:

- **COFF file (.out).** Intermediate file used by MPLINK linker to generate Code file, HEX file, and Listing file.
- **Code file (.cod).** Debug file used by MPLAB IDE.
- **Listing file (.lst).** Original source code, side-by-side with final binary code.
- **Map file (.map).** Shows the memory layout after linking. Indicates used and unused memory regions.

The tools shown here are all Microchip development tools. However, many third party tools are available to work with MPLAB Projects. Please refer to the *Third Party Guide* (DS00104) for more information.

3.4 USING MPLAB C17 WITH MPLAB IDE

This section will guide you, step by step, in using MPLAB IDE and MPLAB Projects with MPLAB C17.

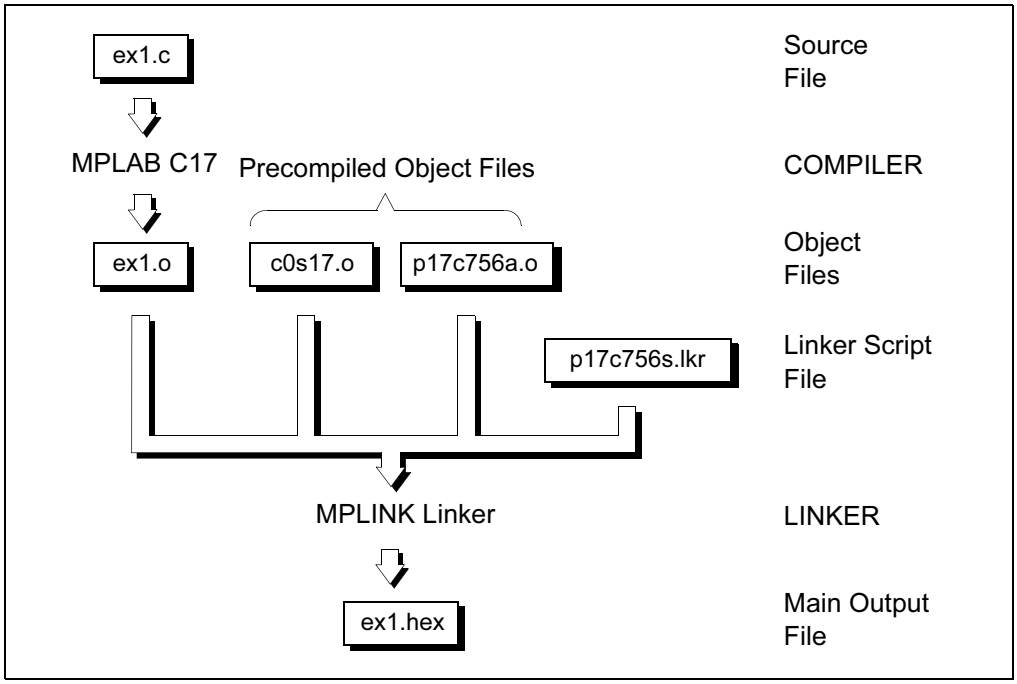
In this tutorial, you will learn how to:

- Create the source file
- Set the MPLAB IDE development mode
- Create a new project with *Project > New Project*
- Set project Node Properties to MPLINK linker
- Add the source file, setting the language tool to MPLAB C17
- Add precompiled nodes (object files)
- Add a linker script node
- Build the project

3.4.1 Overview

Figure 3-2 gives a graphical overview of the MPLAB Project using MPLAB C17. The source file `ex1.c` is associated with the compiler (MPLAB C17) to produce the object file `ex1.o`. This file and other precompiled object files are linked via MPLINK linker according to directions in the linker script file (`p17c756s.lkr`) to produce the main output file, `ex1.hex`.

FIGURE 3-2: AN MPLAB PROJECT WITH MPLAB C17



3.4.2 Create Source File

Select *File > New* to open a blank editor window. Type the following into the window and save it as `ex1.c` in a directory called, for example, `c:\proj0`. This is a very simple program that adds two numbers.

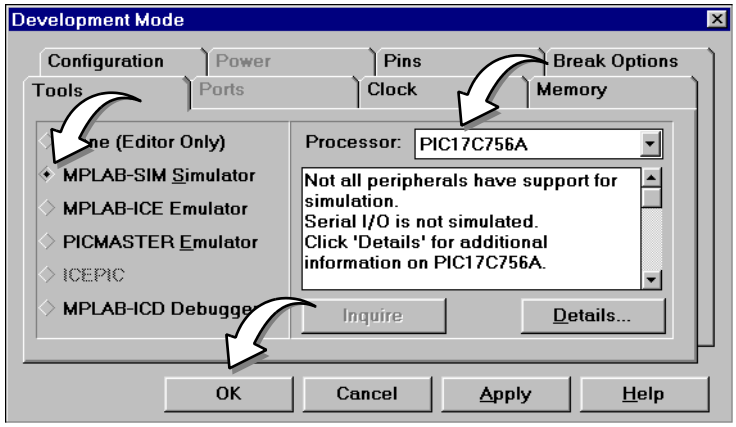
```
#include <p17c756a.h>
void main(void);
unsigned char Add(unsigned char a, unsigned char b);
unsigned char x, y, z;
void main()
{
 x = 2;4
 y = 5;
 z = Add(x,y);
}
unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```


© 2002 Microchip Technology Inc.

3.4.3 Set Development Mode

Set *Options > Development Mode* to MPLAB SIM simulator and select the PIC17C756A PICmicro MCU for this example. Click **OK**.

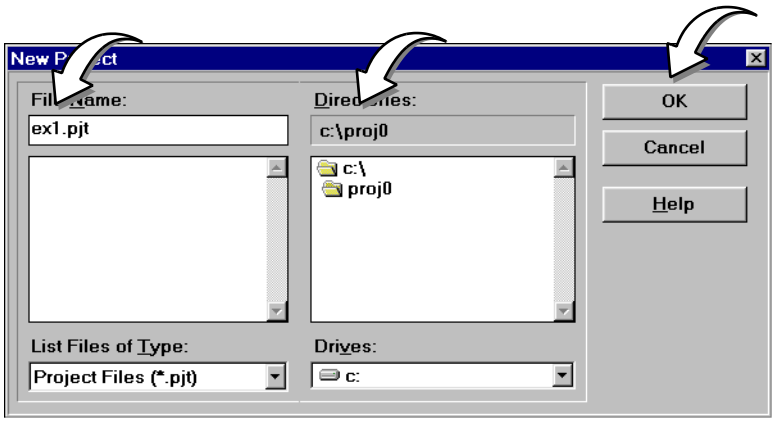
FIGURE 3-3: DEVELOPMENT MODE – PIC17C756A



3.4.4 New Project

In *Project > New Project* select the directory `c:\proj0`. Enter `ex1.pjt` as the File Name for the new project.

FIGURE 3-4: NEW PROJECT – ex1.pjt



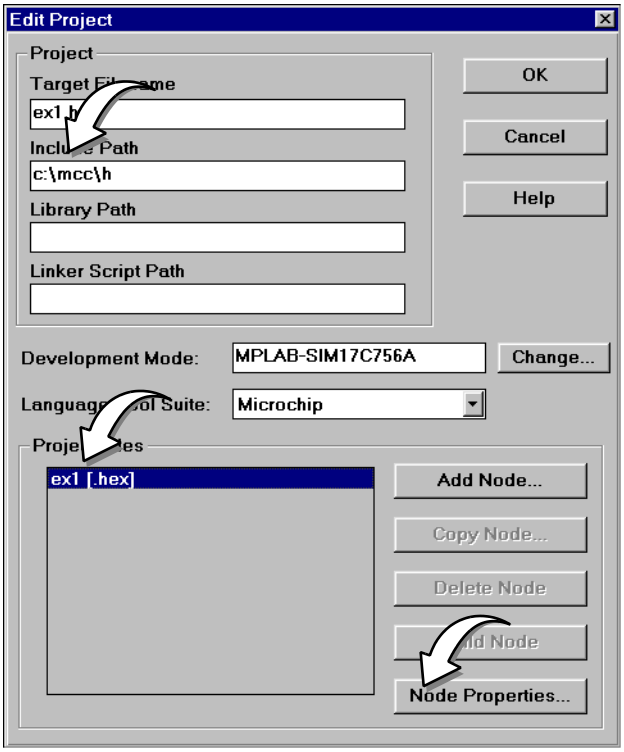
After setting the project name, click **OK** and the Edit Project dialog will be shown.

3.4.5 Edit Project

In the Project section of the Edit Project window, enter `c:\mcc\h` under Include Path.

Click on `ex1 [.hex]` in the Project Files section of the Edit Project dialog to highlight the HEX file name and activate the **Node Properties** button. Then click on **Node Properties**.

FIGURE 3-5: EDIT PROJECT – ex1.pjt

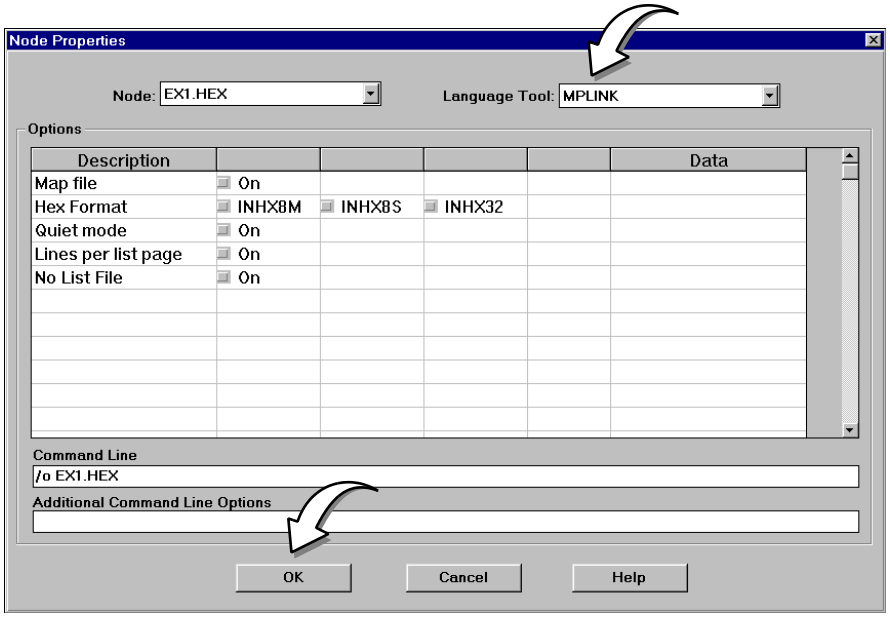


© 2002 Microchip Technology Inc.

3.4.6 Set Node Properties

In the Node Properties dialog, set the Language Tool to MPLINK linker.

FIGURE 3-6: NODE PROPERTIES – ex1.hex



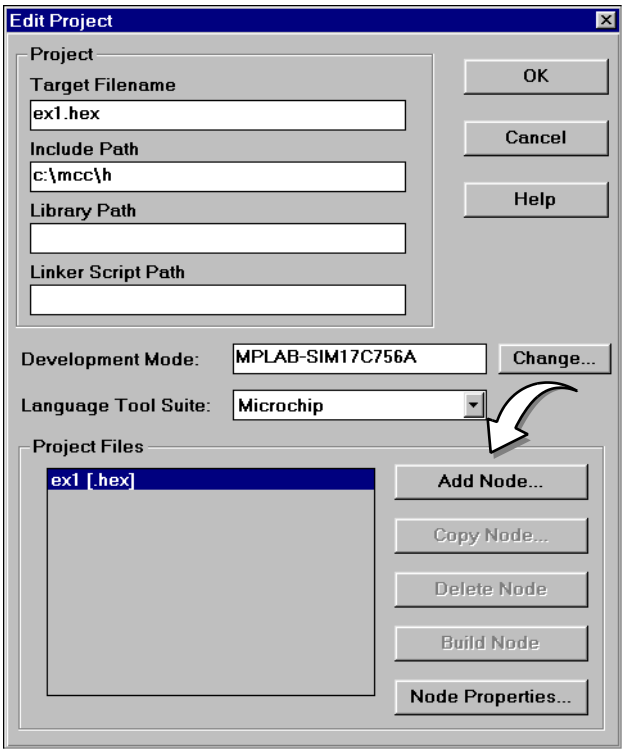
The Node Properties dialog shows the command line switches for the tool, in this case MPLINK linker. When you first open this dialog, the checked boxes represent the default values for the tool. For this tutorial, these do not need to be changed. Refer to the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on these command line switches.

Click **OK** to set these default values to `ex1.hex`.

3.4.7 Add Files to the Project

Several files (nodes) will need to be added to this project. Begin by adding `ex1.c`, the main source file, to the project. Click on **Add Node** on the Edit Project dialog.

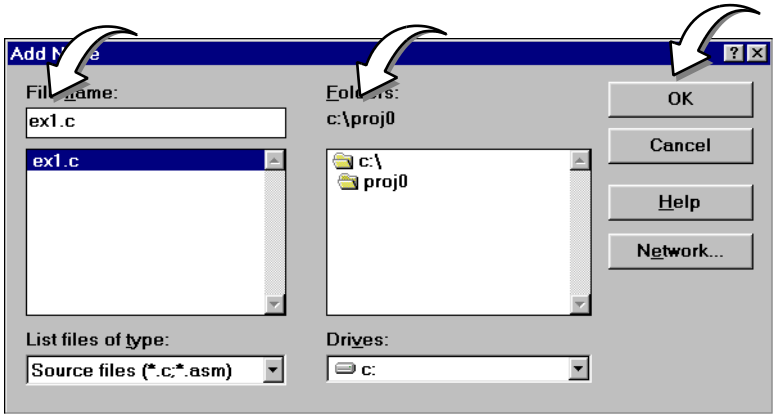
FIGURE 3-7: EDIT PROJECT – ADD NODE ex1.c



3.4.8 Add Source File

In the Add Node dialog, click on the source file, `ex1.c`, from the `c:\proj0` directory. Make sure "List files of type:" specifies 'Source files (*.c;*.asm)'. Click **OK**.

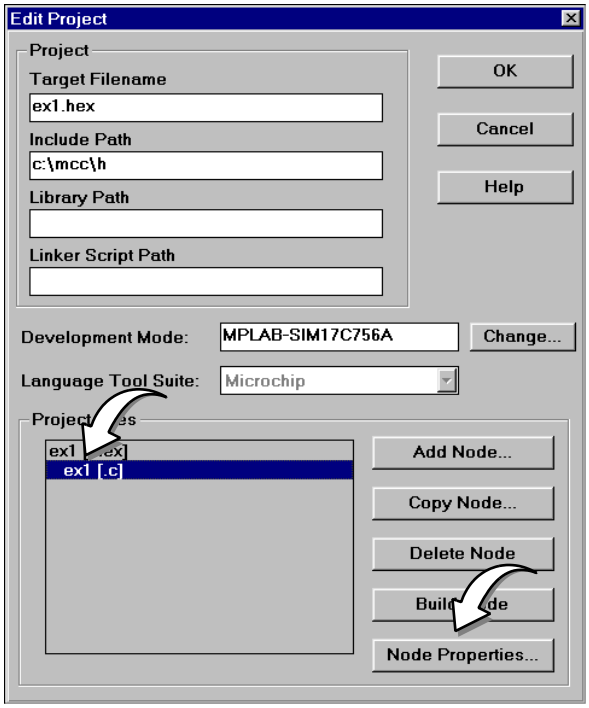
FIGURE 3-8: ADD NODE – ex1.c



© 2002 Microchip Technology Inc.

The Edit Project dialog should now look like Figure 3-9. Click on `ex1 [.c]` in the Project Files section of the dialog and then click on **Node Properties**.

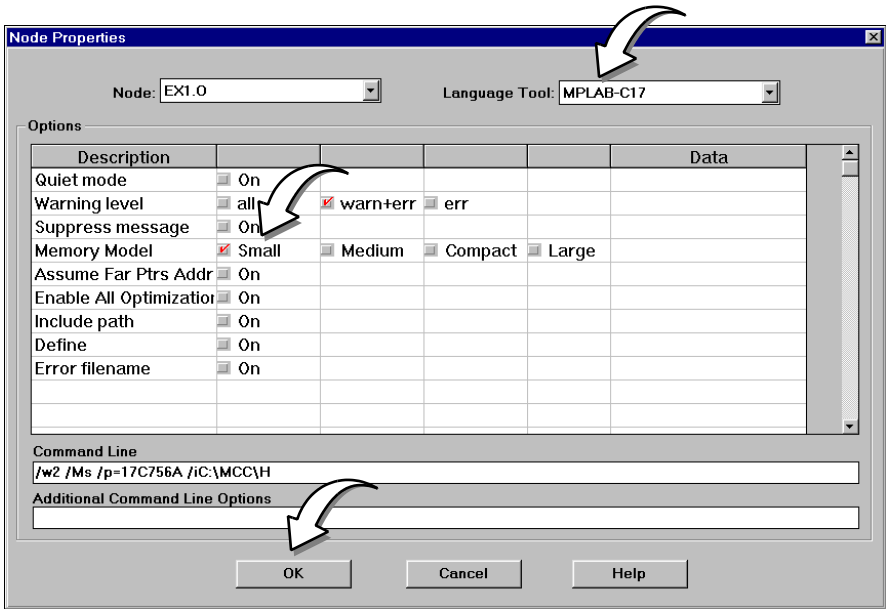
FIGURE 3-9: EDIT PROJECT – ex1.c ADDED



In the Node Properties dialog, verify that the language tool is set to MPLAB C17.

The default for Memory Model is Small, for optimization reasons. The default selection will be used for the example. However, while learning how to use the compiler, it is generally suggested that the large memory model be used, to ensure proper page and bank selection.

FIGURE 3-10: NODE PROPERTIES – ex1.o



The "Object filename" is set to `ex1.o` automatically. Nothing else needs to be changed in this dialog.

Click **OK** to apply these values to `ex1.o`.

3.4.9 Add Precompiled Object Files

In general, several precompiled object files are required for compiling a project. These files are in `c:\mcc\lib`, where `c:\mcc` is the compiler install directory.

- `c0l17.o` – Start Up Code
  (source in `c:\mcc\src\startup`)
- `idata17.o` – Code to Initialize Data
  (source in `c:\mcc\src\startup`)
- `int756al.o` – Interrupt Service Routines
  (source in `c:\mcc\src\startup`)
- `p17c756a.o` – PIC17C756A Register Definitions
  (source in `c:\mcc\src\proc`)

Examination of the source code for each file is recommended to help determine if that file should be included for a specific project.

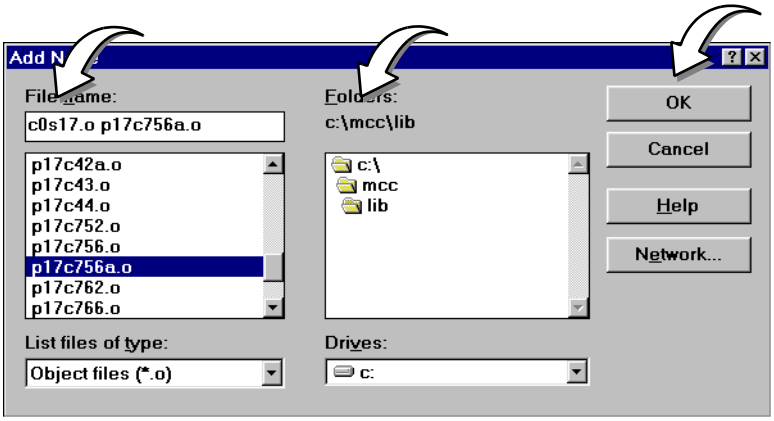
For a simple program like `ex1.c`, the small memory model startup file is used (`c0s17.o`) with no initialized data (`idata17.o`). There are no interrupts, so no interrupt service routines are needed (`int756al.o`). But processor-specific register definitions are included (`p17c756a.o`).

Use the **Add Node** button from the Edit Project dialog to add the necessary precompiled object files. Make sure "List files of type:" specifies 'Object files (*.o)'.

- `c0s17.o`
- `p17c756a.o`

To select more than one file at a time, hold down the **Ctrl** key on your keyboard while selecting the files with your mouse. Click **OK** when done.

FIGURE 3-11: ADD NODE – OBJECT FILES



Node Properties can not be set on precompiled object files, as they are already compiled.

Although there are no library files used in this tutorial project, a library file (`.lib`) may be added by following the same procedure as shown above.

For more information on libraries and precompiled object files, please refer to the *MPLAB C17 C Compiler Libraries* (DS51296).

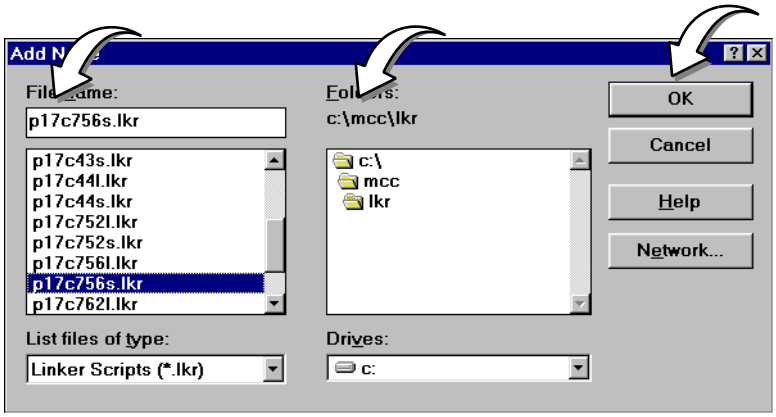
© 2002 Microchip Technology Inc.

3.4.10 Select Linker Script

Use the **Add Node** button from the Edit Project dialog to add the linker script file `p17c756s.lkr` from the `c:\mcc\lkr` directory. Make sure “List files of type:” specifies ‘Linker Scripts (*.lkr)’.

Click **OK** when done. Node Properties can not be set on a linker script.

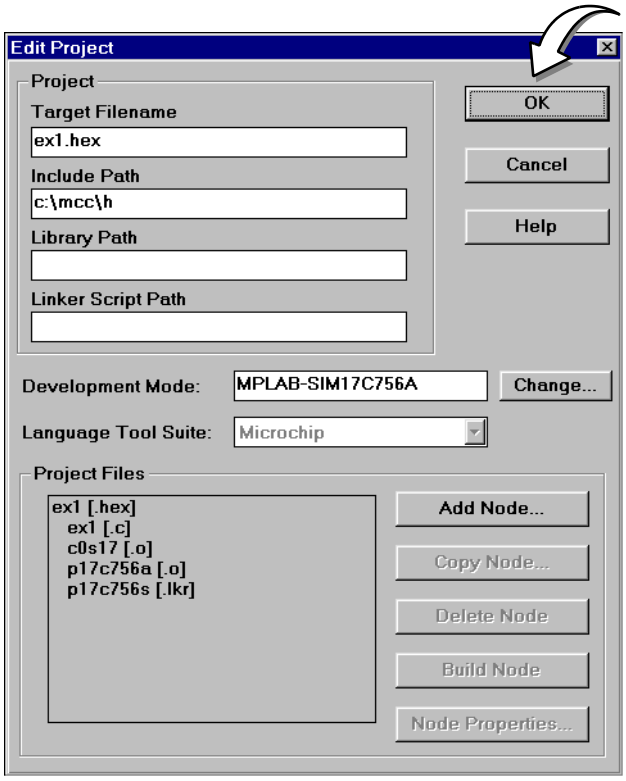
FIGURE 3-12: ADD NODE – p17c756s.lkr



3.4.11 Finish Project Edit

The Edit Project window should now look like this:

FIGURE 3-13: EDIT PROJECT – ex1.hex

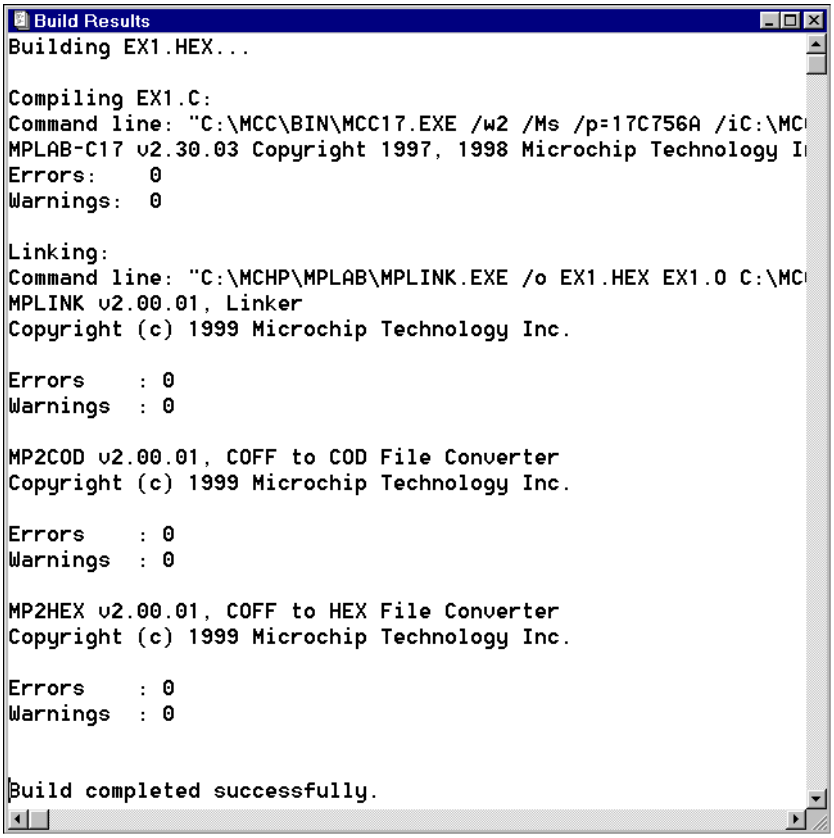


Press **OK** on the Edit Project dialog to finish editing the project.

3.4.12 Make Project

Select *Project > Make Project* from the menu to compile the application using MPLAB C17 and MPLINK linker. A Build Results window is created that shows the command lines sent to each tool. It should look like this:

FIGURE 3-14: BUILD RESULTS – ex1.hex



© 2002 Microchip Technology Inc.

3.4.13 Troubleshooting

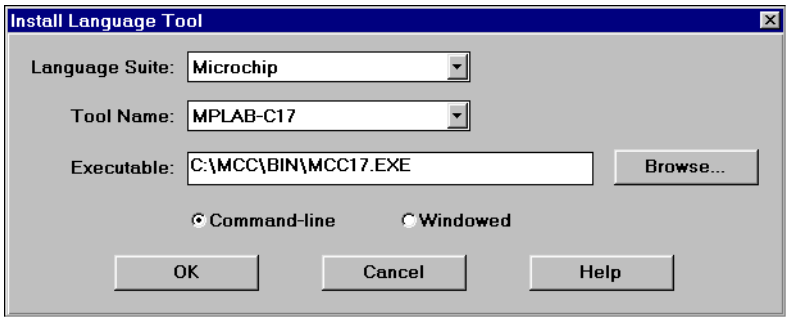
If the build did not complete successfully, check these items:

1. Select *Project > Install Language Tool...* and check that MPLAB C17 references the `mcc17.exe` executable (Figure 3-15). Your executable path may be different from the figure.

   When using MPLAB IDE in the Windows 3.x environment, the `mcc17d.exe` executable should be used instead.

   The Command-line option should be selected.

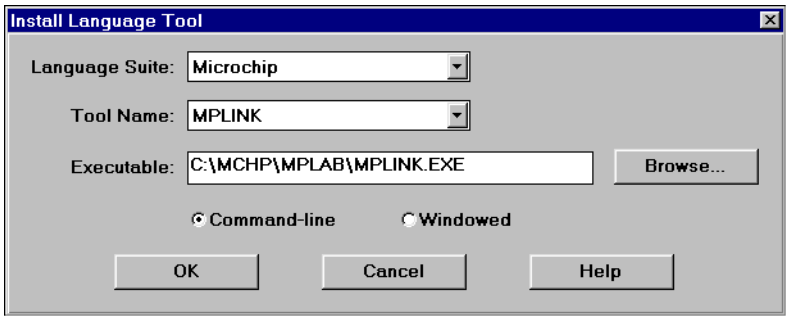
FIGURE 3-15: INSTALL LANGUAGE TOOL – MPLAB C17



2. Select *Project > Install Language Tool...* and check that MPLINK linker is pointing to the `mplink.exe` executable (Figure 3-16). Your executable path may be different from the figure.

   The Command-line option should be selected.

FIGURE 3-16: INSTALL LANGUAGE TOOL – MPLINK LINKER

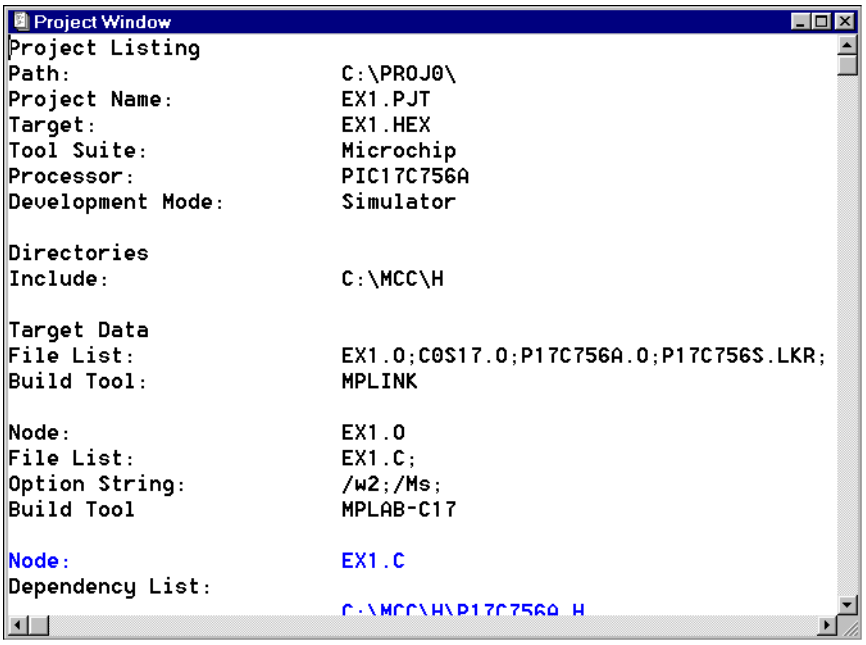


3. Check the Node Properties for the Project Files `ex1.hex` and `ex1.c`. They should be mapped to the Language Tools MPLINK linker and MPLAB C17 respectively.
4. Check the names of the files added to the project against the ones listed in Figure 3-13. If any are different, click on them individually, click **Delete Node**, and then follow the procedure in the relevant previous section for adding the correct node.
5. Check each step of this tutorial to see if you completed it correctly.
6. Compile the project in a DOS window. Cut-and-paste command-line information into a DOS window to run. Check the `autoexec.bat` file to ensure that `PATH` includes the executable directory (`c:\mcc\bin`) and that `MCC_INCLUDE` is present and represents the include directory (`c:\mcc\h`).

3.4.14 Project Window

Open the *Window > Project* window. It should look like this:

FIGURE 3-17: PROJECT WINDOW – ex1.pjt



The project window contains a good deal of useful information about the project. For more information on its contents, see the *MPLAB IDE User's Guide* (DS51025).

3.5 CODE DEVELOPMENT

Once a project has been built successfully, you can go on to simulating, emulating or programming the resulting code into the target device. If you make changes to any source code, you must rebuild the project before debugging or programming again. For more information on simulating, or using MPLAB IDE to debug your code, please refer to the *MPLAB IDE User's Guide* (DS51025).

3.6 ADDITIONAL OPTIONS AND LIBRARY INFORMATION

Not all MPLAB C17 options are available through MPLAB IDE. For additional compiler options, please see Chapter 4 "Using MPLAB C17 on the Command Line."

For a description of libraries and library functions, as well as precompiled object files, available for inclusion in your project, please refer to the *MPLAB C17 Libraries* (DS51296).

© 2002 Microchip Technology Inc.

Chapter 4. Using MPLAB C17 on the Command Line

4.1 INTRODUCTION

This chapter shows examples of how to use MPLAB C17 from the command line.

4.2 HIGHLIGHTS

This chapter includes:

- Command Line Overview
- Using MPLAB C17 on the Command Line
- Code Development
- Library Information

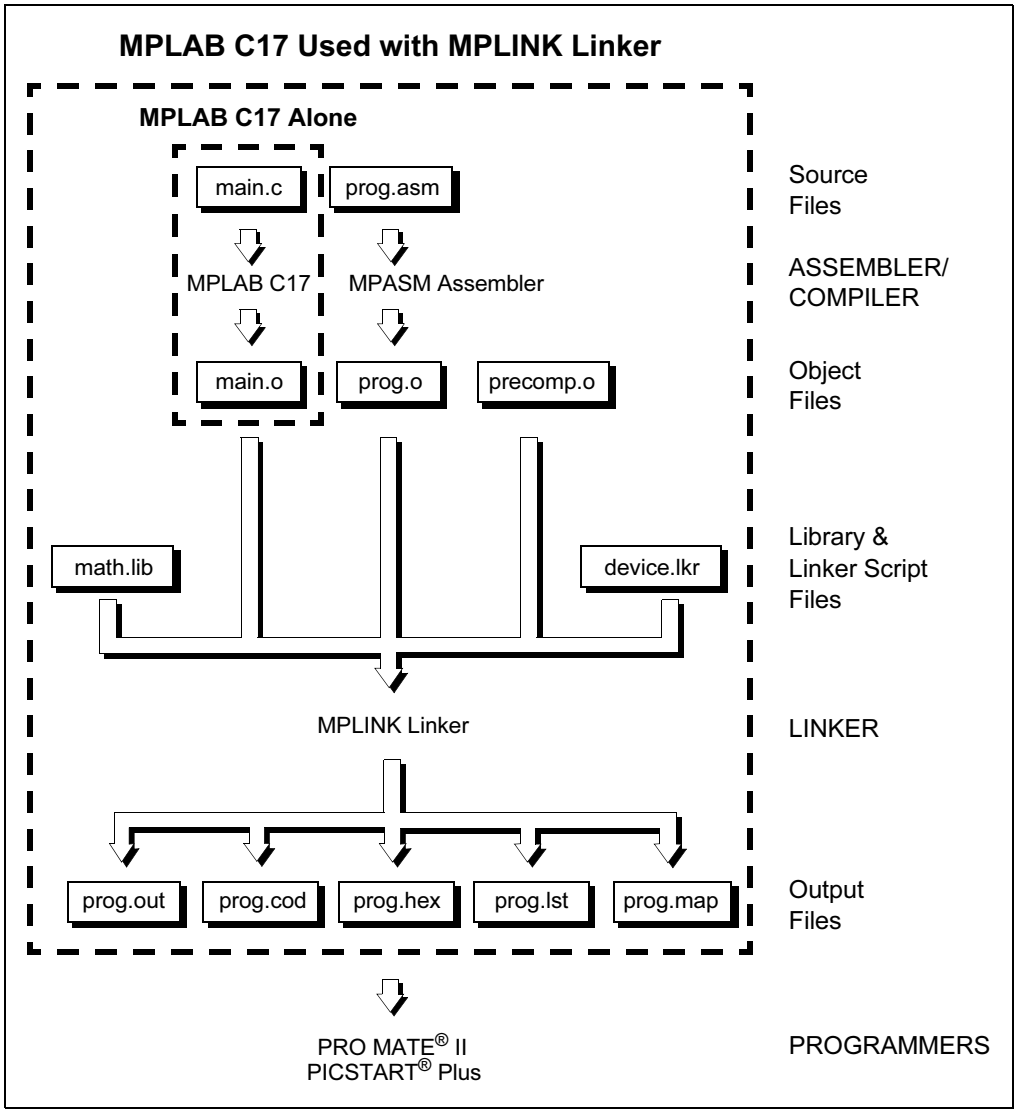
4.3 COMMAND LINE OVERVIEW

MPLAB C17 can be invoked directly on the command line in DOS or a DOS shell window of Windows 3.x (`mcc17d.exe`), or console mode in Windows 95/98 or higher (`mcc17.exe`).

MPLAB C17 may be used alone to compile individual C source files into object files. Or, it may be used in conjunction with MPLINK linker to create HEX files.

Figure 4-1 shows a generic use of the MPLAB C17 compiler tool.

FIGURE 4-1: MPLAB C17 – USED ALONE AND WITH MPLINK LINKER



In this diagram, MPLAB C17 is used alone to compile the source file `main.c` into the object file (`main.o`). However, this object file may be used for input into the linker (MPLINK linker) with other object files to produce a HEX file (`prog.hex`) for use with programmers.

An assembly source file (`prog.asm`) is shown also with its associated assembler (MPASM assembler), producing the object file `prog.o` for input into MPLINK linker. See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on using the assembler.

In addition, precompiled object files (`precomp.o`) may be included. Types of precompiled object files that are generally required for the successful build of a HEX file are listed below.

- Start up code
- Initialization code
- Interrupt service routines
- Register definitions

© 2002 Microchip Technology Inc.

Precompiled object files are often device and/or memory model dependent. For more information on available precompiled object files, see the *MPLAB C17 C Compiler Libraries* (DS51296).

Some library files are available with the compiler. Others may be built outside the project using the librarian tool (MPLIB librarian). See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on using the librarian. For more information on available libraries, see the *MPLAB C17 C Compiler Libraries* (DS51296).

The object files, along with library files and a linker script file (`device.lkr`) are used by MPLINK linker to generate output files. See the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on linker script files and using the linker.

The main output file generated by MPLINK linker is the **HEX file** (`prog.hex`). The other output files are:

- **COFF file (.out).** Intermediate file used by MPLINK linker to generate Code file, HEX file and Listing file.
- **Code file (.cod).** Debug file used by MPLAB IDE.
- **Listing file (.lst).** Original source code, side-by-side with final binary code.
- **Map file (.map).** Shows the memory layout after linking. Indicates used and unused memory regions.

The tools shown here are all Microchip development tools. However, many third party tools are available. Please refer to the *Third Party Guide* (DS00104) for more information.

4.4 USING MPLAB C17 ON THE COMMAND LINE

In this section, the following is discussed:

- Command Line Interface
- How to Compile a Single File on the Command Line
- How to Compile Multiple Files on the Command Line

4.4.1 Command Line Interface

The command line interface of MPLAB C17 is as follows:

```
mcc17 [options] filename
```

where:

`filename` is the name of the file being compiled, and

`options` is zero or more command line options.

For example, if the file `test.c` exists in the current directory, it can be compiled with the following command:

```
mcc17 -p=17c756a test.c
```

When no command line parameters are specified, or with '`-?`' or '`-h`', a help screen is displayed describing the command line usage and options.

Options to MPLAB C17 can be specified with either '`/`' or '`-`', though the '`-`' is shown in the table.

TABLE 4-1: COMMAND LINE OPTION DESCRIPTIONS

Option	Default	Description
-?,-h	–	Help screen.
-d*macro[=text]*	–	Define a macro. Equivalent to placing the following at the head of the file: #define macro text
-fe=*filename*	–	Use *filename* as the name of the output error file.
-fo=*filename*	–	Use *filename* as the name of the output object file.
-i*path*	–	Add the semicolon delimited path, path, to the search path for include files.
-m{s|m|c|l}	s	Select the memory model (see 5.5 "Memory Models"). s:small model (near ram, near rom) m:medium model (near ram, far rom) c:compact model (far ram, near rom) l: large model (far ram, far rom)
-nw#	–	Suppress message number #. Error messages cannot be suppressed.
-O	–	Optimize for smallest code. Equivalent to: -Or -Oc -Op
-Oc[+|-]	Enabled	With this optimization on, the compiler will intelligently determine the level of stack support to include for each function.
-Ol[+|-]	Enabled	When this optimization is on, the default storage class for local variables and function parameters is 'static'.
-Op[+|-]	Disable	When this optimization in on, far pointers to RAM are assumed to not point to SFRs. This simplifies setting the bank for access.
-Or[+|-]	Enabled	With this optimization on, the compiler will run an optimization pass to remove extraneous bank select and MOVLW instructions.
-p=*processor*	17C44	Select to compile for the designated processor.
-q	–	Suppress the sign-on banner (Quiet mode).
-w{1|2|3}	2	Set compiler message level. 1 display errors only 2 display errors and warnings 3 display errors, warnings and messages

Note: Example Files – There are a number of examples in the folder c:\mcc\examples. Execution of the batch file should compile each example after MPLAB C17 is set up. You can use these files as "cookbooks" to begin development of your application.

© 2002 Microchip Technology Inc.

4.4.2 Compiling a Single File on the Command Line

This section demonstrates how to compile and link a single file. For the purpose of this discussion it is assumed the compiler is installed on your `c:` drive in a directory called `mcc`. Therefore the following will apply:

Include directory: **`c:\mcc\h`**
The include directory is where the compiler stores all its system header files. The `MCC_INCLUDE` environment variable should point to that directory (From the DOS command prompt, type “set” to check this.)

Library directory: **`c:\mcc\lib`**
The library directory is where the libraries and precompiled object files reside.

Linker directory: **`c:\mcc\lkr`**
The linker directory is where device-specific linker script files may be found.

Executable directory: **`c:\mcc\bin`**
The executable directory is where the compiler programs are located. Your `PATH` variable should include this directory.

The following is a very simple program that adds two numbers.

1. Create the following program with any text editor and save it as `ex1.c` in a directory called, for example, `c:\proj0`.

```
#include <p17c756a.h>
void main(void);
unsigned char Add(unsigned char a, unsigned char b);
unsigned char x, y, z;
void main()
{
 x = 2;
 y = 5;
 z = Add(x,y);
}
unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```

   The first line of the program includes the header file `p17c756a.h` which provides definitions for all special function registers on that part. For more information on header files see Chapter 8.

2. Compile the program by typing the following at a DOS prompt:

```
mcc17 ex1.c -p=17c756a
```

   This tells the compiler to compile the program `ex1.c` for the PIC17C756A. The compiler will generate one of two files by default. The first file is `ex1.o`, which is the object file that the linker will use to generate (among other files) the executable (`.hex`) file to program your PICmicro MCU. The second file is `ex1.err`, which is the error file containing any error messages and/or warnings that the compiler generates during compilation. This file is only created when there are errors. These messages are also displayed on the screen.

3. The C object file now must be linked with other object files and a linker script to create the final executable file, `ex1.hex`.

   In general, several precompiled object files will be necessary. These files are the start-up code file, `c0l17.o`, the data initialization file, `idata17.o`, an interrupt handler file, `int756al.o`, and the processor definition file, `p17c756a.o`, to reference any special function registers. See the *MPLAB CXX Reference Guide* (DS51224) for more information on these precompiled object files.

   For a simple program like `ex1.c`, the small memory model startup file is used (`c0s17.o`) with no initialized data. There are no interrupts, so no interrupt service routines are needed. But processor-specific register definitions are included.

   Here is the linker command to produce the executable (Although shown on multiple lines here, this should be on one line when executed.):

   ```
   mplink ex1.o -l c:\mcc\lib c0s17.o p17c756a.o -k c:\mcc\lkr
   p17c756s.lkr -m exl.map -o exl.out
   ```

   The file `ex1.o` is linked with the precompiled object files `c0s17.o` and `p17c756a.o`, found in the `c:\mcc\lib` directory specified by the `-l` directive. Specific linker information is provided by the linker script file, `p17c756s.lkr`, found in the `c:\mcc\lkr` directory, specified by the `-k` directive. A map file called `ex1.map` is generated with the `-m` directive. The `-o` directive tells the linker to generate a COFF file called `ex1.out`, used to generate other output files.

   The linker produces the file `ex1.hex`, as well as several other files used for debugging. The HEX file is used by device programmers such as PRO MATE II and PICSTART Plus to program a PICmicro MCU device. For more information on the other debugging files produced by the linker, see the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

Summary:

- Create the source code program, `ex1.c`.
- Compile `ex1.c`:

  ```
  mcc17 ex1.c -p=17c756a
  ```

- Link to generate `ex1.hex`:

  ```
  mplink ex1.o -l c:\mcc\lib c0s17.o p17c756a.o -k c:\mcc\lkr
  p17c756s.lkr -m exl.map -o exl.out
  ```

4.4.3 Compiling Multiple Files on the Command Line

Move the `Add()` function into a file called `add.c` to demonstrate the use of multiple files in a project. That is:

1. **File 1**

   ```
   /* ex1.c */
   #include <p17c756a.h>

   void main(void);

   unsigned char Add(unsigned char a, unsigned char b);
   unsigned char x, y, z;
   void main()
   {
    x = 2;
    y = 5;
    z = Add(x,y);
   }
   ```


© 2002 Microchip Technology Inc.

File 2

```
/* add.c */
#include <p17c756a.h>
unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```

2. To compile these two files, the command lines would be:

```
mcc17 ex1.c -p=17c756a
mcc17 add.c -p=17c756a
```

3. Then link the resulting object files with the precompiled object files as follows (This should be entered on one line):

```
mplink ex1.o add.o -l c:\mcc\lib c0s17.o p17c756a.o -k c:\mcc\lkr
p17c756s.lkr -m exl.map -o exl.out
```

This will produce a HEX file and other output files described in the previous section.

4.5 CODE DEVELOPMENT

Once you have created an executable (HEX) file, you can go on to programming the resulting code into the target device. If you make changes to any source code, you must recompile and relink to create a new executable.

4.6 LIBRARY INFORMATION

For a description of libraries and precompiled object functions, as well as library files, available for inclusion in your project, please refer to the *MPLAB C17 C Compiler Libraries* (DS51296).

NOTES:

© 2002 Microchip Technology Inc.

Section 2 – MPLAB C17 Advanced Usage

Chapter 5. Runtime Environment.....47
Chapter 6. Data Types.....59
Chapter 7. Device Support Files 61
Chapter 8. Interrupts 65
Chapter 9. Mixing Assembly Language and C Modules 71
Chapter 10. Writing Efficient Code 75

© 2002 Microchip Technology Inc.

Chapter 5. Runtime Environment

5.1 INTRODUCTION

This section discusses the MPLAB C17 runtime environment, which is the set of assumptions under which the compiler operates.

5.2 HIGHLIGHTS

Items discussed in this chapter are:

- Code and Data Sections
- Startup and Initialization
- Memory Models
- Locating Code
- Locating Data
- Software Stack
- Software Stack Call Conventions
- Function Call Conventions
- Interrupt Support Macros

5.3 CODE AND DATA SECTIONS

PIC17 microcontroller (MCU) devices have two distinct memory regions: program memory and data memory. A section is a portion of an application located at a specific address of PIC17 memory, either program or data memory.

There are two types of sections for each type of memory.

- program memory
  - **code**: Contains executable instructions.
  - **romdata**: Contains variables and constants.
- data memory
  - **udata**: Contains statically allocated uninitialized user variables.
  - **idata**: Contains statically allocated initialized user variables.

Sections are located through the use of `#pragma` *`sectiontype`* directives, where *`sectiontype`* is either `code`, `romdata`, `udata` or `idata`. Pragma statements are described in 2.6 "Statement Differences" and the directives are described in 5.3.2 "Section Contents".

Sections are absolute, assigned or unassigned. See the MPLINK linker portion of the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more on section allocation.

5.3.1 Section Attributes

Two section attributes may optionally be included in a `#pragma` *`sectiontype`* directive:

- **shared**: Locate this section in a shared (unbanked) region of data memory. See PIC17 device data sheets for more on shared data memory.
- **overlay**: Permit other sections to be located at the same physical address. This attribute can conserve memory by locating local variables from two different functions to the same location (as long as both are not active at the same time.) The overlay attribute can also be used in conjunction with the shared attribute.

See 5.6 "Locating Code" and 5.7 "Locating Data" for more on section attributes.

5.3.2 Section Contents

The type of code or data that will go into each section type is described in the following sections.

5.3.2.1 code

A `code` section contains executable content located in program memory.

5.3.2.2 romdata

A `romdata` section contains data allocated into program memory. For example:

```
rom int l;
rom char c = 'A';
```

Constant strings are also found in program memory. The `.stringtable` section is a `romdata` section that contains all constant strings. For example:

```
strcmpram2pgm (Foo, "hello");
```

For more information on `romdata` usage (e.g., for memory-mapped peripherals) see the *MPLINK User's Guide* portion of DS33014.

5.3.2.3 udata

A `udata` section contains uninitialized data statically allocated into data memory. For example, global variables are allocated as:

```
int i;
char c;
```

Compiler temporary variables are placed in a `udata` section named `_tmpstore`. For an interrupt, compiler temporary variables are created in a `udata` section named `_tmpsection_`*function_name*.

For example:

```
void foo(void);
...
#pragma interrupt foo
void foo(void)
{
   /* perform interrupt function here */
}
```

The compiler temporary variables for interrupt function `foo` will be placed in the `udata` section `_tmpsection_foo`.

© 2002 Microchip Technology Inc.

5.3.2.4 idata

A `idata` section contains initialized data statically allocated into data memory. For example, global variables are allocated as:

```
int i = 0;
char c = 'A';
const j = 5;
```

5.3.3 Default Sections and Names

A default section exists for each section type in MPLAB C17.

Table 5.1: DEFAULT VALUES FOR SECTIONS

Section Type	Default Name
code	`.code_`*filename*
romdata	`.romdata_`*filename*
udata	`.udata_`*filename*
idata	`.idata_`*filename*
filename is the name of the object file being generated.	

5.4 STARTUP AND INITIALIZATION

The Startup file for PIC17 devices is an assembly file that is assembled and linked with the C program files. This file performs these main tasks:

1. Optionally calls the function `__STARTUP()` upon RESET.
2. Optionally calls the code which copies initialized data from program memory to data memory (`idata`).
3. Sets up the software stack used by the compiler.
4. Transfers control to the C function `main()` which is the entry point for C programs.

There are two startup files for the PIC17 family. The first is `c0s17.asm` which uses short GOTOs and CALLs. `c0s17.asm` should be assembled and linked with the small model (code less than 8K). The other startup file is `c0l17.asm` which uses long jumps and LCALLs. `c0l17.asm` should be used with projects targeting memory larger than 8K. A data initialization file, `idata.asm`, may be associated with either startup file.

5.4.1 __STARTUP()

To execute some code immediately after a device RESET but before any other code generated by the compiler is executed, optionally create a function by the name `__STARTUP()`. This will be the first code executed upon a RESET. To use a `__STARTUP()` function in a program:

1. Define a `__STARTUP()` function in a C program as follows:

```
void __STARTUP(void)
{
   // Initialize some registers to 0
   TRISB = 0;
   TRISC = 0;
}
```

> **Note:** The space shown between the two underlines preceding `STARTUP()` is for illustration and should not be used in actual code (i.e., there should be no space).

2. In `c0l17.asm` or `c0s17.asm`, uncomment the line:

   `#DEFINE USE_STARTUP`

3. Compile the source file, assemble `c0l17.asm` or `c0s17.asm` and link.

> **Note:** Since __STARTUP() is executed before the stack is initialized, `auto` variables may not be used.

5.4.2 Initialized Data Support

When declaring initialized data (such as: `int x = 5;`), the variable is allocated in data memory but the value is stored in program memory. Before the data is usable in any program, the values must be copied from program memory into the variable in data memory.

The size of the MPLAB C17 initialization code is approximately 50 words. Therefore, to only initialize a few variables, do not use that feature and initialize the variables manually in the code. If initializing many variables (10 or more integers or 20 or more characters) as they are declared, then the initialization code is the better option in terms of code size.

To use initialized data with `c0s17.asm` in a MPLAB C17 program:

1. Uncomment the following line in `c0s17.asm`:
   `#DEFINE USE_INITDATA`
2. Assemble `c0s17.asm` to produce `c0s17.o`.
3. Assemble `idata17.asm` to produce `idata17.o`, or use `idata17.o` directly.
4. Link the above files with the C object code.

To use initialized data with `c0l17.asm` in a MPLAB C17 program:

1. Assemble `c0s17.asm` to produce `c0s17.o`, or use `c0s17.o` directly.
2. Assemble `idata17.asm` to produce `idata17.o`, or use `idata17.o` directly.
3. Link the above files with the C object code.

> **Note:** `c0s17.asm` is assembled with `USE_INITDATA` undefined by default. `c0l17.asm` is assembled with `USE_INITDATA` defined by default.

5.4.3 Stack Initialization

The stack initialization simply points the compiler stack pointer to the right location in data memory.

5.4.4 Branching to main()

After the startup code optionally calls `__STARTUP()` and/or copies initialized data, and sets up the stack, it calls the `main()` function of the C program. There are no arguments passed to `main()`.

MPLAB C17 transfers control to `main()` via a `goto`, i.e.;

```
goto main
```


© 2002 Microchip Technology Inc.

5.5 MEMORY MODELS

Different devices access memory differently. Depending on your selected device, you will need to use different memory model versions of libraries and/or precompiled object files. See the *MPLAB C17 C Compiler Libraries* (DS51296) for a list of libraries and precompiled object files available for different memory models.

For PIC17 devices with program memory (ROM) over 8K, the memory is broken up in to pages. For PIC17 devices with data memory (RAM) over 256, the memory is broken up into banks. Table 5.1 spells out the memory models available for these devices via MPLAB C17.

Table 5.1: Memory Model Usage – MPLAB C17

Memory Model		Device Description
s	small	`near rom` – program memory ≤ 8K, `near ram` – data memory ≤ 256
m	medium	`far rom` – program memory > 8K, `near ram` – data memory ≤ 256
c	compact	`near rom` – program memory ≤ 8K, `far ram` – data memory > 256
l	large	`far rom` – program memory > 8K, `far ram` – data memory > 256

The usage of the keywords near, far, ram and rom is discussed in Chapter 6.

5.6 LOCATING CODE

5.6.1 Section Types Used to Place Code

Following a `#pragma code`, all generated code will be assigned to the specified section until another `#pragma code` is encountered. An absolute code section allows the location of code to a specific address.

For example:

```
#pragma code myCode=0x2000
```

5.6.2 Attributes – Overlay

Code sections that have the `overlay` attribute can be located at an address that overlaps other overlay code sections. This feature is generally not needed, but could be useful with multiple external memory sources.

For example:

```
#pragma code overlay myScnName=0x1000
```

5.7 LOCATING DATA

5.7.1 Section Types Used to Place Data

The different types of data sections are:

- **udata**: Statically allocated uninitialized global data is placed here. Used for data memory allocation.
- **idata**: Statically allocated initialized global data goes here. Used for data memory allocation.
- **romdata**: Variables declared with the `rom` attribute are placed here. Used for program memory allocation.

5.7.2 Attributes – Overlay

Data sections that have the `overlay` attribute can be located at an address that overlaps other overlay data sections. This feature can be useful for allowing a single data range to be used for multiple variables that are never active simultaneously.

For example:

```
#pragma udata overlay myOverlayData1=0x1fc
int intVar1, intVar2;  // 4 bytes will be located at
                       // 0x1fc and 0x1fe

#pragma udata overlay myOverlayData2=0x1fc
long longVar;  // 4 bytes will be located at 0x1fc
```

5.7.3 Attributes – Shared

Data sections declared with the `shared` attribute will be placed into memory regions that are defined as `SHAREBANK` in the linker script file. These regions are those that can be accessed without banking. Variables declared with the `near` keyword will be accessed without banking.

For example:

```
#pragma udata shared unbankedRAM
near unsigned char cv1, cv2;  // all accesses to these
                              // will be unbanked
```

5.7.4 Locating Data in Program Memory

Variables can be placed in either data or program memory with the MPLAB C17 compilers. Variables that are placed in on-chip program memory can be read but not written without additional user-supplied code. Variables placed in external program memory can be either read or written without additional user-supplied code.

For information on writing to on-chip program memory, see individual device data sheets. For detailed examples of variable placement, see the *MPLINK User's Guide* portion of DS33014.

To specify to the compiler that a variable should be placed in program memory, use the `rom` keyword. The compiler will then allocate this variable into the current `romdata` type section.

For example:

```
#pragma romdata constTable
const rom char myConstArray[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

5.8 SOFTWARE STACK

The compiler uses a software stack for storing local variables and for passing arguments to and returning values from functions. The software stack should not be confused with the hardware stack that the PICmicro MCU uses for storing return addresses during function calls and interrupts.

5.8.1 Changing Stack Size and Location

The stack is sized and placed in the linker script via the `STACK` directive. The `STACK` directive has two arguments: `SIZE` and `RAM` to control the allocated stack size and its location respectively.

For example, to allocate a 128 byte stack and place that stack in the memory region `gpr3`:

```
STACK SIZE=0x80 RAM=gpr3
```


© 2002 Microchip Technology Inc.

The size of the software stack required by a program varies with the complexity of the program. The following should be kept in mind:

- One RAM location will be reserved by the compiler for use as the Stack Pointer.
- When nesting function calls, all arguments and local variables (auto variables included) of the calling function will remain on the stack. Therefore, the stack must be large enough to accommodate the requirements by all functions in a tree.

> **Note:** Stack Overflow Avoidance – For MPLAB SIM or MPLAB ICE 2000, use a `break` statement at the last location on the stack. If the program breaks, then a stack overflow would have occurred in the next byte

5.8.2 Controlling What Goes on the Stack

MPLAB C17 supports parameters and local variables allocated either on the software stack or directly from global memory. The `static` keyword places a local variable or a function parameter in global memory instead of on the software stack. Stack-based local variables and arguments require more code to access than static local variables and supporting arguments. See 10.3 “Static Locals And Parameters” for more information on static parameters and locals. Functions that use stack-based variables are more flexible in that they are reentrant and recursive.

5.9 SOFTWARE STACK CALL CONVENTIONS

The MPLAB C17 software stack is an upwards growing stack data structure on which the compiler places function arguments and local variables. The software stack is distinct from the hardware stack upon which the PICmicro MCU places function call return addresses.

The stack pointer always points to the next available stack location. MPLAB C17 allocates the first location of the stack for use as the stack pointer, leaving both FSR0 and FSR1 available for other use by the compiler.

When a function is invoked, its stack based arguments are pushed onto the stack in right-to-left order and the function is called. The leftmost function argument is on the top of the software stack upon entry into the function.

MPLAB C17 immediately adds the total size of the stack based local variables for the function to the stack pointer, allocating space for instances of those variables. References to stack based argument values and stack based local variables are resolved according to offsets from the stack pointer.

5.10 FUNCTION CALL CONVENTIONS

A function is reentrant if it may be safely called from multiple threads of control at the same time. For instance, if a function is called by an interrupt service routine, it must be reentrant because if an interrupt occurs while the function is executing, the function will be called again by the interrupt routine.

In general, a function which only manipulates stack-based local variables and stack-based parameters is reentrant. References to global variables or the use of static parameters or static local variables may cause a function be non-reentrant.

5.11 INTERRUPT SUPPORT MACROS

MPLAB C17 provides interrupt support macros and code for saving and restoring context during interrupt handling. The use of such macros and code are optional. It is recommended that interrupt handling be done in the assembler to reduce latency and minimize overhead.

Each PICmicro MCU processor has two interrupt support assembly files. One is for the small model and the other for the large model as before. These files contain code fragments that save critical special function registers, call the interrupt handling function and returns from the interrupt. The registers are saved as follows:

- First ALUSTA is saved primarily to preserve the Z bit. The saved ALUSTA can go in any bank (since BSR isn't known at that time) but always at location 0xFF. The interrupt support code reserves location 0xFF in all banks for `save_ALUSTA`.
- Second, PCLATH is saved at location 0xFE or the equivalent location in the same manner as with ALUSTA. The interrupt support code reserves location 0xFE in all banks for `save_PCLATH`.
- Finally BSR and WREG are saved in bank 0 at locations 0xFD and 0xFC, respectively. The interrupt support code reserves locations 0xFD and 0xFC in bank 0 for `save_BSR` and `save_WREG`.

> **Note:** Startup code supplied with MPLAB C17 does not support nested interrupts.

Here is how the highest addresses in data memory would look if an interrupt occurred while BSR was pointing to bank 2 on the PIC17C756. (For the PIC17C44 only banks 0 and 1 are used.)

Table 5.2: Interrupt Example

	Bank 0	Bank 1	Bank 2	Bank 3
0xFB	<Available>	<Available>	<Available>	<Available>
0xFC	**save_WREG**	<Available>	<Available>	<Available>
0xFD	**save_BSR**	<Available>	<Available>	<Available>
0xFE	<Reserved>	<Reserved>	**save_PCLATH**	<Reserved>
0xFF	<Reserved>	<Reserved>	**save_ALUSTA**	<Reserved>

© 2002 Microchip Technology Inc.

The ALUSTA, PCLATH, BSR and WREG are the registers that absolutely need to be saved before we branch to the interrupt service function. However, there are other registers used by the compiler that are worth saving under certain circumstances. The following is an example that uses the Timer 0 Overflow Interrupt.

```
#include <p17c44.h>
unsigned char x;
void _ _TMR0()
{
 x++;
 PORTB = x;
}
void main()
{
 x = 1;
// Install interrupt handler for timer 0 interrupt
 Install_TMR0(_ _TMR0);
// Set prescale value for TMR0
 T0STA = 0b11100110;
// Unmask TMR0 overflow interrupt
 INTSTA = 0b00000010;
// Enable all unmasked interrupts
 CPUSTA = 0;
// Set Port B in o/p mode
 TRISB = 0;
 while(1)
 {
 // Loop and wait for an interrupt to take place!
 }
}
```

Install `_TMR0 (_ _TMR0)` sets the function `_ _TMR0()` as the interrupt handler for Timer 0 overflow interrupts. Then the appropriate prescale value, interrupt flag and global interrupt enable flag are set. The program enters into an infinite loop when it reaches the `while(1)` statement. When Timer 0 overflows, program control goes to the `_ _TMR0()` function where the value of ‘x’ is sent to PORT B and possibly displayed on LEDs.

In this simple program the PICmicro MCU wasn't doing much at the time the interrupt occurred. Therefore do not save any more registers in addition to what the compiler interrupt code saved. However, in a more complex application, the interrupt may occur at any point in the program. Therefore other registers may need to be saved. The best way to find out is to look at the generated code for the interrupt handling function. Find out which registers are used by the compiler inside the function and make sure to save them at the beginning and restore them at the end of the function. Looking at the following example's generated code, determine that registers PRODL and PRODH are used both inside and outside the interrupt function.

```
#include <p17c44.h>
#pragma udata intSave = 0xFa
 unsigned char save_PRODL;      // 0xF9
 unsigned char save_1F;         // 0xFA
 unsigned char save_1E;         // 0xFB
#pragma udata anywhere
 unsigned char x, y;
void _ _TMR0()
{
_asm
 movpf PRODL, save_PRODL
 movpf PRODH, save_1E
 movpf PRODL, save_1F
_endasm
 x++;
 PORTB = x;
 y = y * x;
_asm
 movlr 0                  // Switch to bank 0
 movfp save_PRODL, PRODL
 movfp save_1E, PRODH
 movfp save_1F, PRODL
_endasm
}
void main()
{
   x = y = 1;
   Install_TMR0(_ _TMR0);
// Set prescale value for TMR0
   T0STA = 0b11100110;
// Unmask TMR0 overflow interrupt
   INTSTA = 0b00000010;
// Enable all unmasked interrupts
   CPUSTA = 0;
// Set Port B in o/p mode
   TRISB = 0;
   while(1)
   {
     x = x * 5;
   }
}
```

The registers PRODH and PRODL are saved in `save_1F`, `save_1E` and `save_PRODL`, respectively. These variables are declared globally and allocated at locations 0xFa to 0xFB in bank 0 using the `#pragma udata` directive. This places them at the end of the bank just before `save_B` and guarantees they are in bank 0. Since BSR is cleared in the interrupt support code, don't do any bank switching to save those three registers. However, clear the BSR (using `MOVLR 00`) before restoring them as the interrupt function code could have switched banks.

© 2002 Microchip Technology Inc.

The following are merely guidelines as to what the compiler might be using for certain tasks. However, the best guarantee that the context is saved and restored correctly is by looking at generated code.

1. **WREG:** This is necessary if the program is doing anything other than looping when an interrupt occurs. It is best to save WREG at all times.
2. **FSR0, FSR1:** Save FSR0 if the interrupt handling function uses arrays or pointers.
3. **PRODL, PRODH:** Save these registers if performing multiplication in the interrupt function. The compiler potentially uses PRODL and PRODH if it is evaluating a complex expression.
4. **TBLPTRL, TBLPTRH:** These two registers are used for table read and write operations. However, the compiler rarely uses them for temporary storage. In general, it is not recommended to do table reads or writes in the interrupt functions if done elsewhere in the program. Table reads and writes use the 16-bit TBLAT register for latching data transferred from and to program memory. Since TBLAT is not an addressable register it cannot be saved or restored during interrupts.

NOTES:

© 2002 Microchip Technology Inc.

Chapter 6. Data Types

6.1 INTRODUCTION

This section discusses the MPLAB C17 data types. For information on how MPLAB C17 data types differ from ANSI C data types, see 2.6 "Statement Differences".

6.2 HIGHLIGHTS

Items discussed in this chapter are:

- Data Representation
- Integer
- Floating Point

6.3 DATA REPRESENTATION

Multibyte quantities are stored in "little endian" format, which means:

- The least significant byte is stored at the lowest address
- The least significant bit is stored at the lowest-numbered bit position

As an example, the long value of `0x12345678` is stored at address `0x100` as follows:

0x100	0x78	0x56	0x101
0x102	0x34	0x12	0x103

As another example, the long value of `0x12345678` is stored in registers `m` and `m+1`:

m	m+1
0x5678	0x1234

6.4 INTEGER

Table 6-1 shows integer data types are supported in MPLAB C17.

TABLE 6-1: INTEGER DATA TYPES

type	bits	min	max
`char, signed char`	8	-128	127
`unsigned char`	8	0	255
`short, signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int, signed int`	16	-32768	32767
`unsigned int`	16	0	65535
`long, signed long`	32	-2^{31}	$2^{31} - 1$
`unsigned long`	32	0	$2^{32} - 1$
** ANSI-89 extension			

For information on implementation-defined behavior of integers, see 12.5 "Integers".

6.5 FLOATING POINT

The floating point representation used for the MPLAB C17 compiler is a variant of the IEEE 754 standard, see 12.6 "Floating Point" for more information. The following table compares the two.

TABLE 6-2: MPLAB C17 FLOATING POINT FORMAT VS IEEE 754

Standard	Exponent Byte	Byte 0	Byte 1	Byte 2	Byte 3
IEEE754	sxxx xxxx	yxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx
C17	xxxx xxxx	sxxx xxxx	xxxx xxxx	xxxx xxxx	xxxx xxxx

© 2002 Microchip Technology Inc.

Chapter 7. Device Support Files

7.1 INTRODUCTION

This section discusses device support files used in support of MPLAB C17 compilation.

7.2 HIGHLIGHTS

Items discussed in this chapter are:

- Processor Header File
- Register Definitions File
- Using SFRs

7.3 PROCESSOR HEADER FILE

The processor header file is a C file that contains external declarations for the special function registers. Register definitions are found in the Register Definitions File (7.4 "Register Definitions File").

In addition to register declarations, the header file defines in-line assembly macros (Table 7-1) and interrupt install macros.

There are certain instructions on PICmicro MCUs that may need to execute from the C code. They can be included as in-line assembler instructions but for convenience they are also available as macros in C. They are listed in the following table:

TABLE 7-1: INSTRUCTION MACRO ACTIONS

Instruction Macro	Action
`Nop()`	Executes a no operation (`NOP`).
`ClrWdt()`	Clears the watchdog timer (`CLRWDT`).
`Sleep()`	Executes a `SLEEP` instruction.
`Reset()`	Executes a device reset (`RESET`).
`Rlcf(var,dest,access)`	Rotates '*var*' to the left through the carry bit.
`Rlncf(var,dest,access)`	Rotates '*var*' to the left without going through the carry bit.
`Rrcf(var,dest,access)`	Rotates '*var*' to the right through the carry bit.
`Rlncf(var,dest,access)`	Rotates '*var*' to the right without going through the carry bit.
`Swapf(var,dest,access)`	Swaps the upper and lower nibble of '*var*'.

Note: '`var`' must be an 8-bit quantity (e.g., char) and not located on the stack.

Header files are device (processor) specific, (i.e., choose the header file `p17c756.h` when coding for the PIC17C756). These files are contained in the `c:\mcc\h` directory, where `c:\mcc` is the compiler install directory.

Processor header files for PIC17 devices contain external declarations for the special function registers, in-line assembly macros and interrupt install macros.

PIC17 header files have four macros for installing interrupt service routines to the four interrupt vectors available. Call these macros as part of setting up the interrupt handler functions. Specify which C function should act as the interrupt handling function for a particular interrupt vector. For more information on how interrupts are handled by MPLAB C17, please refer to Chapter 9. Interrupt support macros are listed in the following table:

TABLE 7-2: MACRO ACTIONS

Macro	Action
`Install_INT(`*func*`)`	Sets '*func*' as the handler for the INT interrupt.
`Install_TMR0(`*func*`)`	Sets '*func*' as the handler for the TMR0 interrupt.
`Install_T0CKI(`*func*`)`	Sets '*func*' as the handler for the T0CKI interrupt.
`Install_PIV(`*func*`)`	Sets '*func*' as the handler for the PIV interrupt.

7.4 REGISTER DEFINITIONS FILE

The register definitions file is an assembly file that contains declarations for all the special function registers on the device. Every register definitions file is associated with a C header file (7.3 "Processor Header File") that contains, among other things, external declarations for the special function registers.

The register definitions file, when compiled, will become an object file that will need to be linked to the source file. As an example, `p17c756.asm` compiles to `p17c756.o`, added to the tutorial project in Chapter 5. These files are device (processor) specific.

Register definitions file source code is found in the `c:\mcc\src\proc` directory and compiled object code is found in the `c:\mcc\lib` directory, where `c:\mcc` is the compiler install directory.

An example of a use for a register definitions file in MPLAB C17 is as follows.

EXAMPLE 7-1: PIC17C44 PORT A DEFINITION

Here Port A is defined in the register definitions file `p17c44.asm` as:

```
BANK0_SFR_SEC   DATA   H'010'
PORTAbits
PORTA   RES   1    ; 010h
TRISB   RES   1    ; 011h
.
.
```

and so on.

© 2002 Microchip Technology Inc.

The first line specifies the file register bank where Port A is located and the starting address for that bank. Port A has two labels, `PORTAbits` and `PORTA`, both referring to the same location (in this case `010h` in bank 0). So the above definition reserves 1 byte for `PORTA` and `PORTAbits` at location `010h`.

In `p17c44.h`, Port A is declared as:

```
volatile extern far unsigned char PORTA;
```

and as:

```
extern far volatile union
{
 struct
 {
      unsigned RA0:1;     /* Bit 0 */
      unsigned RA1:1;
      unsigned RA2:1;
      unsigned RA3:1;
      unsigned RA4:1;
      unsigned RA5:1;
      unsigned :1;
      unsigned NOT_RBPU:1;
 };
 struct
 {
      unsigned INT:1;     /* Alternate name for bit 0 */
      unsigned T0CKI:1;   /* Alternate name for bit 1 */
      unsigned :6;        /* pad next 6 locatons */
 };
} PORTAbits;
```

The first declaration specifies that `PORTA` is a byte (unsigned char), whereas the second one declares `PORTAbits` as a union of bit-addressable structures. Since individual bits in a special function register may have more than one function (and hence more than one name), there are multiple structure definitions inside the union all referring to the same register. Respective bits in all structure definitions refer to the same bit in the register. Where a bit has only one function for its position, it is simply padded in other structure definitions. For example, bits 2 through 7 on Port A are simply padded in the second structure definition using the statement `unsigned :6`.

When using a special function register such as Port A, write the following statements:

```
PORTA = 0x34;       /* Assigns the value 0x34 to the */
                    /* whole port */
PORTAbits.INT = 1; /* Sets the INT pin high */
PORTAbits.RA0 = 1; /* Sets the RA0 pin high, same as */
                    /* above statement */
```

The `extern` modifier is needed since the variables are declared in the register definitions file. The `volatile` modifier tells the compiler that it cannot assume that Port A retains values assigned to it. The `far` modifier specifies that the port needs a bank switching instruction prior to access.

7.5 USING SFRS

There are three steps to follow when using SFR's in an application.

1. Include the processor header file for the appropriate device. This provides the source code with the SFR's that are available for that device. For instance, the following statement includes the header files for the PIC17C756A part:

   ```
   #include <p17c756a.h>
   ```

2. Access SFR's like any other C variables. The source code can write to and/or read from the SFR's.

   For example, the following statement clears all the bits to zero in the special function register for Timer1.

   ```
   TMR1 = 0;
   ```

   This next statement represents the 0 bit in the TCON2 register which is the 'Timer 1 on' bit. It sets the bit named `TMR1ON` to `1` which starts the timer.

   ```
   TCON2bits.TMR1ON = 1;
   ```

3. Link with the register definition file or linker script for the appropriate device. The linker provides the addresses of the SFR's. (Remember the bit structure will have the same address as the SFR at link time.) Example 7.1 would use:

   ```
   p17c756a.lkr
   ```

See *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014) for more information on using linker scripts.

© 2002 Microchip Technology Inc.

Chapter 8. Mixing Assembly Language and C Modules

8.1 INTRODUCTION

This section describes how to use assembly language and C modules together. It gives examples of using C variables and functions in assembly code and examples of using assembly language variables and functions in C.

8.2 HIGHLIGHTS

This chapter covers the following topics:

- Internal Assembler
- Calling Conventions
- Mixing Assembly Language and C Variables and Functions
- Using In-Lline Assembly Langauge

8.3 INTERNAL ASSEMBLER

MPLAB C17 has an internal assembler using a syntax similar to MPASM assembler. The block of assembly code must begin with `_asm` and end with `_endasm`. The syntax within the block is:

```
<instruction> [arg1][, arg2][, arg3]
```

Comments must be C or C++ type notation.

EXAMPLE 8-1: IN-LINE ASSEMBLY CODE

```
_asm
/* User assembly code */
movlw 7       // Load 7 into WREG
movwf PORTB   // and send it to PORTB
_endasm
```

It is generally recommended to limit the use of in-line assembly to a minimum. To write large fragments of assembly code, use the stand-alone assembler and link the modules to the C modules using the MPLINK linker.

The in-line assembler differs from the stand-alone assembler as follows:

- No directive support.
- Full text mnemonics must be used for table reads/writes (i.e., TABLWT).
- No defaults for instruction parameters. All parameters must be fully specified.

8.4 CALLING CONVENTIONS

The following general calling conventions should be used.

Calling Assembly Routines from C:

- Function declared as `extern` in C module.
- Label declared as `global` in ASM module.
- Function declaration may return a value and/or contain parameters.
- Functions are called using standard C function notation.

Calling C Routines from Assembly:

- C functions are inherently `global`.
- Function name must be declared as `extern` symbol in assembly file.
- `call` must be used to make function call; `RETURN 0x00` implemented at end of C function.

8.5 MIXING ASSEMBLY LANGUAGE AND C VARIABLES AND FUNCTIONS

The following example shows how to use variables and functions in both assembly language and C regardless of where they were originally defined. These example files may be found in `c:\mcc\examples\example2`, where `c:\mcc` is the compiler install directory.

The file `ex_c.c` defines `main` and `c_variable` to be used in the assembly language file. The C file also shows how to call an assembly function, `asm_function`, and how to access the assembly defined variable, `asm_variable`.

The file `ex_asm.asm` defines `asm_function` and `asm_variable` as required to use them in a linked C file. The assembly file also shows how to call a C function, `main`, and how to access a C defined variable, `c_variable`.

```
ex_c.c
// file: ex_c.c
extern unsigned asm_variable;
extern near void asm_function( void );
extern void main( void );
unsigned c_variable;
void main(void)
{
    asm_function();     // will modify 'c_variable'
    asm_variable = 0x1234;
}

ex_asm.asm
; file: ex_asm.asm
LIST P=17C44
    EXTERN main          ; defined in C module
    EXTERN c_variable    ; also defined in C module
MYCODE CODE
asm_function
    movlw 0xff
    movwf c_variable     ; put 0xffff in the C declared
                         ; variable
    movwf c_variable+1
    return
    GLOBAL asm_function  ; export so linker can see it
MYDATA UDATA
asm_variable    RES 2    ; 2 byte variable
    GLOBAL asm_variable  ; export so linker can see it
    END
```


© 2002 Microchip Technology Inc.

8.6 USING IN-LINE ASSEMBLY LANGUAGE

The following example shows how to call an assembly function with a parameter. These example files may be found in `c:\mcc\examples\example3`, where `c:\mcc` is the compiler install directory.

Most of the work is done in the file `call_asm.asm` where the parameter is taken off of the software stack. `call_c.c` calls the `asm_function` with a parameter.

```
call_c.c
// File call_c.c
unsigned char asm_function( auto unsigned char a );
unsigned char x;
void main( void )
{
 x = asm_function( 0xff );
}

call_asm.asm
; File call_asm.asm
 LIST P=17C756
 EXTERN _stack
 GLOBAL asm_function
MYCODE CODE
asm_function
 banksel _stack       ; Get the stack pointer into 0x00
 movfp   _stack, 0x01
 decf    0x01, f      ; Point FSR1 at the argument
 movfp   0x00, 0x0a   ; Get the argument
 decf    0x0a, f
       ; The convention is that we return
       ; with FSR0 pointing at the return value.
       ; We'll just reuse the space
       ; allocated for the argument since we're already
       ; pointed there.

 movwf   0x00        ; Store the return value
 return
 END
```

NOTES:

© 2002 Microchip Technology Inc.

Chapter 9. Interrupts

9.1 INTRODUCTION

Interrupt processing is an important aspect of most microcontroller applications. Interrupts may be used to synchronize software operations with events that occur in real time. When interrupts occur, the normal flow of software execution is suspended and special functions are invoked to process the event. At the completion of interrupt processing, previous context information is restored and normal execution resumes.

PIC17 devices support multiple interrupts from both internal and external sources. The MPLAB C17 compiler provides full support for interrupt processing in C or in-line assembly code. This chapter presents an overview of interrupt processing which is applicable to both device families.

9.2 HIGHLIGHTS

This chapter describes the basic steps you should take for implementing interrupt processing in your C program. The following subsections give an overview of the interrupt processing procedure.

- **Writing an Interrupt Service Routine** – You can designate one or more C functions as interrupt service routines (ISR's) to service the occurrence of an interrupt. For best performance in general, place lengthy calculations or operations that require library calls in the main application. This strategy optimizes performance and minimizes the possibility of losing information when interrupt events occur rapidly.

  MPLAB C17 provides interrupt support macros for saving and restoring context during interrupt handling.
- **Writing the Interrupt Vector** – PIC17 devices use interrupt vectors to transfer application control when an interrupt occurs. An interrupt vector is a dedicated location in program memory that specifies the address of an ISR. Applications must contain valid function addresses in these locations to use interrupts.
- **Interrupt Service Routine Context Saving** – To handle returning from an interrupt to code in the same conditional state as before the interrupt, context information from specific registers must be saved. Certain registers are managed by the compiler and may not be saved. See 1.6 "Reserved Resources" for a list of compiler-managed resources.
- **Latency** – The time between when an interrupt is called and when the first ISR instruction is executed is the latency of the interrupt.
- **Nesting Interrupts** – MPLAB C17 does not support nested interrupts.
- **Enabling/Disabling Interrupts** – Enabling and disabling interrupt sources occurs at two levels: globally and individually. These concepts are discussed in 9.8 "Enabling/Disabling Interrupts" and covered in more detail in Chapter 11.

9.3 WRITING AN INTERRUPT SERVICE ROUTINE

Following the guidelines in this section, you can write all of your application code, including your interrupt service routines (ISR's), using only C language constructs.

9.3.1 Guidelines for Writing ISR's

The guidelines for writing ISR's are:

- Declare ISR's with no parameters and a void return type
- Do not let ISR's call other functions
- Do not let ISR's be called by main line code

An MPLAB C17 ISR is like any other C function in that it can have local variables and access global variables. However, an ISR needs to be declared with no parameters and no return value. This is necessary because the ISR, in response to a hardware interrupt, is invoked asynchronously to the mainline C program. (i.e., it is not called in the normal way, so parameters and return values don't apply).

ISR's should only be invoked through a hardware interrupt and not from other C functions. There are two reasons for this. First, an interrupt service routine uses the return from interrupt (RETFIE) instruction to exit from the function rather than the normal RETURN instruction. Using a RETFIE instruction out of context can corrupt processor resources, such as WREG. Second, ISR's use a temporary data area that is distinct from that used by normal C functions. If an ISR were called by an ordinary function during the occurrence of an interrupt, then the temporary data area may become corrupted. For a similar reason, a MPLAB C17 ISR should not call any other functions; otherwise the temporary data area used by ordinary functions may become corrupted.

9.3.2 Syntax for Writing ISR's

Use the `interrupt` pragma to declare C functions as ISR's. The syntax for the MPLAB C17 `interrupt` pragma is as follows.

`#pragma interrupt` *function-name* [*section-name*]

The *function-name* parameter of the pragma names the C function serving as an ISR.

The optional *section-name* parameter names the section which allocates the temporary data of the ISR. Any temporary data required during the evaluation of expressions in the function is allocated in a private memory section and is not overlaid with the temporary locations of any other functions, including other interrupt functions. The default temporary data settings for *section-name* is as follows:

`isr_tmp` (single interrupt priority level)

9.3.3 Coding ISR's

Place the `interrupt` pragma before the C function that is serving as the ISR. Then, code the function prototype immediately after the pragma.

© 2002 Microchip Technology Inc.

9.4 WRITING THE INTERRUPT VECTOR

When using interrupts, you must generate the appropriate interrupt vector. This is code that is located in special program memory locations and is used to branch to your ISR.

PIC17 devices accomplish this by calling an install function with a function pointer to the function that performs the interrupt service.

EXAMPLE 9-1:PIC17 INTERRUPT VECTOR

```
#include <p17CXX.h>

// ---------------------------------------------------------------------
// Interrupt service routines' function prototypes.
// ---------------------------------------------------------------------
void INT_interrupt_service_routine(void);
void TMR0_interrupt_service_routine(void);
void T0CKI_interrupt_service_routine(void);
void PIV_interrupt_service_routine(void);  // peripheral ISR

void Int_interrupts(void) {
// ---------------------------------------------------------------------
// Install the interrupt service routine for an external interrupt on
// the INT pin.
// ---------------------------------------------------------------------
Install_INT(INT_interrupt_service_routine);

// ---------------------------------------------------------------------
// Install the interrupt service routine for TMR0 Overflow interrupt.
// ---------------------------------------------------------------------
Install_TMR0(TMR0_interrupt_service_routine);

// ---------------------------------------------------------------------
// Install the interrupt service routine for an external interrupt on
// the T0CKI pin.
// ---------------------------------------------------------------------
Install_T0CKI(T0CKI_interrupt_service_routine);

// ---------------------------------------------------------------------
// Install the interrupt service routine for all peripheral interrupts.
// ---------------------------------------------------------------------
Install_PIV(PIV_interrupt_service_routine);
}
```

9.5 INTERRUPT SERVICE ROUTINE CONTEXT SAVING

Interrupts, by their very nature, can occur at unpredictable times. Therefore, the interrupted code must be able to resume with the same machine state that was present when the interrupt occurred.

To properly handle a return from interrupt, the setup (prolog) code for an ISR function automatically saves the compiler-managed special function registers to a temporary location for later restoration at the end of the ISR.

If the ISR uses file registers other than those mentioned in 9.3 "Writing an Interrupt Service Routine", they must be saved/restored. To determine which registers need to be saved, you must examine the generated code and note which file registers are used. However, unless normal functions are called from the interrupt, it is not necessary to save data stored in compiler-generated temporaries, because interrupt routines get their own set of temporaries.

It is recommended that interrupt handling be done in the assembler to reduce latency and minimize overhead. However, there are several interrupt support macros available (see 5.11 "Interrupt Support Macros").

9.6 LATENCY

There are four elements that affect the number of cycles between the time the interrupt source occurs and the execution of the first instruction of your ISR code. These are:

Processor Servicing of Interrupt – The amount of time it takes the processor to recognize the interrupt and branch to the first address of the interrupt vector.

Interrupt Vector Execution – The code at the interrupt vector that branches to the ISR prolog code and saves off the ALUSTA and PCLATH values.

ISR Library Wrapper Code – MPLAB C17 implements interrupt handling using a library wrapper routine that saves STATUS and BSR and branches to your ISR.

ISR Code – MPLAB C17 does not add any additional overhead to your ISR code.)

TABLE 9-1: ISR LATENCY TIMES (CYCLES)

	Small Memory Model	Large Memory Model
Processor servicing of interrupt	*	*
Interrupt vector execution	4	7
ISR prolog code	11	11
ISR code	0	0
* Refer to the device data sheet and interrupt source being used		

9.7 NESTING INTERRUPTS

MPLAB C17 does not support nested interrupts because of the mechanism used for saving off the currently active registers (ALUSTA, PCLATH, BSR and WREG).

By default, PIC17 devices will prevent nested interrupts from occurring. Before the ISR routine is invoked, the GLINTD bit of the CPUSTA register is automatically set to disable further interrupts. Upon completion, the GLINTD bit is automatically cleared to reenable interrupts. The programmer should not attempt to modify the GLINTD bit within the ISR routine.

9.8 ENABLING/DISABLING INTERRUPTS

The PIC17 devices contain 18 sources of interrupts. All interrupts are initially disabled.

© 2002 Microchip Technology Inc.

For PIC17CXX devices, the CPUSTA register contains the Global Interrupt Disable (GLINTD) bit. Setting this bit (value on RESET) disables all interrupts. You have the capability to:

- Enable/disable global interrupts
- Enable/disable individual interrupt(s)

The details of how to enable and disable interrupts is discussed in Chapter 11.

NOTES:

© 2002 Microchip Technology Inc.

Chapter 10. Writing Efficient Code

10.1 INTRODUCTION

This section discusses how to write efficient C code for MPLAB C17.

10.2 HIGHLIGHTS

Items discussed in this chapter are:

- Static Locals And Parameters
- Optimization Tips

10.3 STATIC LOCALS AND PARAMETERS

Normally, function parameters and local variables have a storage class of `auto`, are allocated on the stack, and any initializers are executed whenever the function is invoked. However, it is possible to declare both parameters and local variables to have static storage class, meaning that they will be allocated globally. This allows for more efficient code when accessing the variable. When parameters and locals are declared `static`, the following implications arise:

- Initializers are only executed once, at application startup.

Therefore, if the function is depending on the variable being freshly initialized at each function invocation, the initialization must be moved into the function body.

- Initializers must involve values that are known at startup.

For example, a static local variable cannot be assigned a value that involves a parameter to the function or a function call. The compiler will enforce this restriction.

10.4 OPTIMIZATION TIPS

Because of the limited memory on microcontrollers, optimization becomes an issue as code complexity increases.

1. Choose the correct memory model for your libraries and precompiled object files. Don't just pick the large memory model versions for inclusion in your project. Consult 5.5 "Memory Models" for more information.
2. Use the linker script to group variables that are used together into the same data memory bank to minimize bank switching. Intelligent use of the `varlocate` pragma and the `section` directive can yield excellent results.

   To minimizing bank switching, place the following in the linker script:

   ```
   SECTION NAME=coeffs RAM=temperature
   ```

   and the following in the program:

   ```
   #pragma varlocate coeff
   ```

Use of section pragma's to effectively manage RAM and ROM. Refer to 7.3 "Processor Header File" for information on the pragma directive. For examples of use, see the MPLINK linker examples found in the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

© 2002 Microchip Technology Inc.

NOTES:

© 2002 Microchip Technology Inc.

Section 3 – References

Chapter 11. Enabling/Disabling Interrupts 79
Chapter 12. Implementation-Defined Behavior 113
Chapter 13. MPLAB C17 Diagnostics 117

© 2002 Microchip Technology Inc.

Chapter 11. Enabling/Disabling Interrupts

11.1 INTRODUCTION

This chapter discusses how to enable and disable interrupts for PIC17 devices.

11.2 HIGHLIGHTS

This chapter covers the following topics:

- Enabling Interrupts
- Disabling Interrupts

11.3 ENABLING INTERRUPTS

The PIC17 devices contain 18 sources of interrupts:

Non-Peripheral	Peripheral
External Interrupt on T0CKI Pin External Interrupt on INT Pin TMR0 Overflow Interrupt	TMR1 Overflow Interrupt TMR2 Overflow Interrupt TMR3 Overflow Interrupt PORTB Interrupt on Change Capture1 Interrupt Capture2 Interrupt Capture3 Interrupt* Capture4 Interrupt* USART1 Transmit Interrupt USART1 Receive Interrupt USART2 Transmit Interrupt* USART2 Receive Interrupt* Synchronous Serial Port Interrupt* Bus Collision Interrupt* A/D Module Interrupt

* PIC17C7XX devices only.

All Interrupts on the PIC17 devices are initially disabled. The CPUSTA register contains the Global Interrupt Disable (GLINTD) bit. Setting this bit (value on RESET) disables all interrupts. Enabling interrupts requires:

- Enabling global interrupts
- Enabling individual interrupt(s)

11.3.1 Enabling Global Interrupts

To use interrupts on the PIC17 devices, you must enable global interrupts, which will enable all unmasked interrupts. There are two ways to enable global interrupts.

Library Call

To enable global interrupts, you must include the `int16.h` header file and call the `Enable` function.

```
#include <int16.h>

Enable();
```

Modifying Register Bits

To enable global interrupts, you must clear the GLINTD bit of the CPUSTA register.

```
CPUSTAbits.GLINTD = 0;
```

11.3.2 Enabling Individual Interrupt(s)

For PIC17C4X devices, enabling individual peripheral interrupts occurs through bits in the Peripheral Interrupt Enable register (PIE). For PIC17C7XX devices, this occurs via bits in the Peripheral Interrupt Enable register 1 (PIE1), and the Peripheral Interrupt Enable register 2 (PIE2). The INTSTA register contains the enable bits for non-peripheral interrupts.

11.3.2.1 EXTERNAL INTERRUPT ON T0CKI PIN

This interrupt is edge triggered. You can specify whether this interrupt occurs on the rising or falling edge by setting or clearing, respectively, the TMR0 Status/Control (T0STA) register's Timer0 External Clock Input Edge Select (T0SE) bit – [T0STA<6>].

```
#ifdef _TRIGGER_ON_RISING_EDGE
   T0STAbits.T0SE = 1;
#else
   T0STAbits.T0SE = 0;
#endif
```

To enable the external interrupt on the T0CKI pin, you must clear the INTSTA register's External Interrupt on T0CKI Pin Flag (T0CKIF) bit – [INTSTA<6>]. This clears any previous occurrences of external interrupts on the T0CKI pin. You must set the INTSTA register's External Interrupt on T0CKI Pin Enable (T0CKIE) bit – [INTSTA<2>].

```
INTSTAbits.T0CKIF = 0;
INTSTAbits.T0CKIE = 1;
```


© 2002 Microchip Technology Inc.

11.3.2.2 EXTERNAL INTERRUPT ON INT PIN

There are two ways to enable the external interrupt on the INT pin:

Library Call

To enable the external interrupt on the INT pin through a library call, you must include the `int16.h` header file and call the `OpenRA0INT` function.

```
#include <int16.h>

  #ifdef _TRIGGER_ON_RISING_EDGE
     OpenRA0INT(INT_ON | INT_RISE_EDGE);
  #else
     OpenRA0INT(INT_ON | INT_FALL_EDGE);
  #endif
```

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on the rising or the falling edge by setting or clearing, respectively, the TMR0 Status/Control (T0STA) register's RA0/INT Pin Interrupt Edge Select (INTEDG) bit – [T0STA<7>].

```
  #ifdef _TRIGGER_ON_RISING_EDGE
     T0STAbits.INTEDG = 1;
  #else
     T0STAbits.INTEDG = 0;
  #endif
```

To enable the external interrupt on the INT pin, you must clear the INTSTA register's External Interrupt on RA0/INT Pin Flag (INTF) bit – [INTSTA<4>]. This clears any previous occurrence of external interrupts on the RA0/INT pin. You must set the INTSTA register's External Interrupt on RA0/INT Pin Enable (INTE) bit – [INTSTA<0>].

```
  INTSTAbits.INTF = 0;
  INTSTAbits.INTE = 1;
```

11.3.2.3 TMR0 OVERFLOW INTERRUPT

There are two ways to enable the TMR0 overflow interrupt:

Library Call

To enable the TMR0 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `OpenTimer0` function. The `OpenTimer0` function also allows you to select a prescale selection value for the timer, as well as the source (external/internal) of the clock.

The following table outlines the `#define`'s that exist in `timers16.h` header file for prescale selection and the prescale value that is selected:

#define	Prescale Value
T0_PS_1_1	1:1
T0_PS_1_2	1:2
T0_PS_1_4	1:4
T0_PS_1_8	1:8
T0_PS_1_16	1:16
T0_PS_1_32	1:32
T0_PS_1_64	1:64
T0_PS_1_128	1:128
T0_PS_1_256	1:256

The following table outlines the `#define`'s that exist in `timers16.h` header file for selecting an external or internal clock source:

#define	Prescale Value
T0_SOURCE_EXT	External (I/O pin)
T0_SOURCE_INT	Internal (Tosc)

```
#include <timers16.h>

// Enable the TMR0 Overflow interrupt on either the
// rising or falling edge. Set the prescale value to
// 1:8. Use an external clock.

#ifdef _TRIGGER_ON_RISING_EDGE
   OpenTimer0(TIMER_INT_ON | T0_EDGE_RISE |
      T0_PS_1_8 | T0_SOURCE_EXT);
#else
   OpenTimer0(TIMER_INT_ON | T0_EDGE_FALL |
      T0_PS_1_8 | T0_SOURCE_EXT);
#endif
```


© 2002 Microchip Technology Inc.

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on the rising or the falling edge by setting or clearing, respectively, the TMR0 Status/Control (T0STA) register's Timer0 External Clock Input Edge Select (T0SE) bit – [T0STA<6>].

```
#ifdef _TRIGGER_ON_RISING_EDGE
   T0STAbits.T0SE = 1;
#else
   T0STAbits.T0SE = 0;
#endif
```

To select the timer's prescale value, you must set the Timer0 Prescale Selection (T0PS3:T0PS0) bits – [T0STA<4>:T0STA<1>]. The following table outlines the values of these bits and the prescale value that is selected:

T0PS3:T0PS0	Prescale Value
0000	1:1
0001	1:2
0010	1:4
0011	1:8
0100	1:16
0101	1:32
0110	1:64
0111	1:128
1xxx	1:256

To set Timer0's prescale value to 1:8, perform the following:

```
T0STA = T0STA & 0b11100001;  // Reset the prescale value
T0STA = T0STA | 0b00000110;
```

To set Timer0's prescale value to 1:64, perform the following:

```
T0STA = T0STA & 0b11100001;  // Reset the prescale value
T0STA = T0STA | 0b00001100;
```

To select the source (external/internal) of the clock, you must set (internal instruction clock cycle) or clear (external clock input on the T0CKI pin) the Timer0 Clock Source Select (T0CS) bit – [T0STA<5>].

```
#ifdef _EXTERNAL_CLOCK
   T0STAbits.T0CS = 0;
#else
   T0STAbits.T0CS = 1;
#endif
```

To enable the TMR0 overflow interrupt, you must clear the INTSTA register's TMR0 Overflow Interrupt Flag (T0IF) bit – [INTSTA<5>]. This clears any previous occurrences of TMR0 overflow interrupts. You must set the INTSTA register's TMR0 Overflow Interrupt Enable (T0IE) bit – [INTSTA<1>].

```
INTSTAbits.T0IF = 0;
INTSTAbits.T0IE = 1;
```

To reset Timer0, you must clear the TMR0L and TMR0H registers.

```
TMR0L = 0;
TMR0H = 0;
```

11.3.2.4 TMR1 OVERFLOW INTERRUPT

There are two ways to enable the TMR1 overflow interrupt:

Library Call

To enable the TMR1 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `OpenTimer1` function. The `OpenTimer1` function also allows you to select the source (external/internal) of the clock, as well as whether to treat Timer1 and Timer2 as individual 8-bit timers or as one 16-bit timer.

The following table outlines the `#define`'s that exist in `timers16.h` header file for selecting an external or internal clock source:

#define	Clock Source
T1_SOURCE_EXT	External (I/O pin)
T1_SOURCE_INT	Internal (TOSC)

The following table outlines the `#define`'s that exist in `timers16.h` header file for specifying whether to treat Timer1 and Timer2 as individual 8-bit timers or as one 16-bit timer:

#define	Timer2:Timer1 Mode
T1_T2_8BIT	Timer1 and Timer2 individual 8-bit timers
T1_T2_16BIT	Timer1 and Timer2 16-bit single timer

```
#include <timers16.h>

// Enable the TMR1 Overflow interrupt.
// Use Timer1 as an individual 8-bit timer.
#ifdef _EXTERNAL_CLOCK
  OpenTimer1(TIMER_INT_ON | T1_SOURCE_EXT | T1_T2_8BIT);
#else
  OpenTimer1(TIMER_INT_ON | T1_SOURCE_INT | T1_T2_8BIT);
#endif
```

Modifying Register Bits

To select the source (external/internal) of the clock, you must clear (internal clock) or set (external clock input on the TCLK12 pin) the Timer1 Clock Source Select (TMR1CS) bit – [TCON1<0>].

```
#ifdef _EXTERNAL_CLOCK
  TCON1bits.TMR1CS = 1;
#else
  TCON1bits.TMR1CS = 0;
#endif
```

To specify whether to treat Timer1 and Timer2 as individual 8-bit timers or as one 16-bit timer, you must clear or set, respectively, the Timer2:Timer1 Mode Select (T16) bit – [TCON1<3>].

```
#ifdef _16BIT_CLOCK
  TCON1bits.T16 = 1;
#else
  TCON1bits.T16 = 0;
#endif
```


© 2002 Microchip Technology Inc.

PIC17C7XX DEVICES

To enable the TMR1 overflow interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) TMR1 Interrupt Flag (TMR1IF) bit – [PIR1<4>].
- The Timer1 On (TMR1ON) bit – [TCON2<0>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of TMR1 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) TMR1 Interrupt Enable (TMR1IE) bit – [PIE1<4>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR1ON = 0;
PIR1bits.TMR1IF = 0;
PIE1bits.TMR1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the TMR1 overflow interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) TMR1 Interrupt Flag (TMR1IF) bit – [PIR<4>].
- The Timer1 On (TMR1ON) bit – [TCON2<0>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences ofTMR1 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) TMR1 Interrupt Enable (TMR1IE) bit – [PIE<4>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR1ON = 0;
PIRbits.TMR1IF = 0;
PIEbits.TMR1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

ALL PIC17 DEVICES

To reset Timer1, clear the TMR1 register.

> **Note:** If you are using Timer1 and Timer2 as a single 16-bit timer, you must also clear the TMR2 register.

```
TMR1 = 0;
#ifdef _16BIT_CLOCK
   TMR2 = 0;
#endif
```

To start Timer1, set the Timer1 On (TMR1ON) bit – [TCON2<0>].

> **Note:** If you are using Timer1 and Timer2 as a single 16-bit timer, you must also set the Timer2 On (TMR2ON) bit – [TCON2<1>].

```
TCON2bits.TMR1ON = 1;
#ifdef _16BIT_CLOCK
TCON2bits.TMR2ON = 1;
#endif
```

11.3.2.5 TMR2 OVERFLOW INTERRUPT

There are two ways to enable the TMR2 overflow interrupt:

Library Call

To enable the TMR2 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `OpenTimer2` function. The `OpenTimer2` function also allows you to select the source (external/internal) of the clock.

The following table outlines the `#define`'s that exist in `timers16.h` header file for selecting an external or internal clock source:

#define	Clock Source
T2_SOURCE_EXT	External (I/O pin)
T2_SOURCE_INT	Internal (TOSC)

```
#include <timers16.h>

// Enable the TMR2 Overflow interrupt.
#ifdef _EXTERNAL_CLOCK
   OpenTimer2(TIMER_INT_ON | T2_SOURCE_EXT);
#else
   OpenTimer2(TIMER_INT_ON | T2_SOURCE_INT);
#endif
```

Modifying Register Bits

To select the source (external/internal) of the clock, you must clear (internal clock) or set (external clock input on the TCLK12 pin) the Timer2 Clock Source Select (TMR2CS) bit – [TCON1<1>].

```
#ifdef _EXTERNAL_CLOCK
   TCON1bits.TMR2CS = 1;
#else
   TCON1bits.TMR2CS = 0;
#endif
```


© 2002 Microchip Technology Inc.

PIC17C7XX DEVICES

To enable the TMR2 overflow interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) TMR2 Interrupt Flag (TMR2IF) bit – [PIR1<5>].
- The Timer2 On (TMR2ON) bit – [TCON2<1>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of TMR2 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) TMR2 Interrupt Enable (TMR2IE) bit – [PIE1<5>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR2ON = 0;
PIR1bits.TMR2IF = 0;
PIE1bits.TMR2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the TMR2 overflow interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) TMR2 Interrupt Flag (TMR2IF) bit – [PIR<5>].
- The Timer2 On (TMR2ON) bit – [TCON2<1>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of TMR2 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) TMR2 Interrupt Enable (TMR2IE) bit – [PIE<5>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR2ON = 0;
PIRbits.TMR2IF = 0;
PIEbits.TMR2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

ALL PIC17 DEVICES

To reset Timer2, clear the TMR2 register.

```
TMR2 = 0;
```

To start Timer2, set the Timer2 On (TMR2ON) bit – [TCON2<1>].

```
TCON2bits.TMR2ON = 1;
```

11.3.2.6 TMR3 OVERFLOW INTERRUPT

There are two ways to enable the TMR3 overflow interrupt:

Library Call

To enable the TMR3 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `OpenTimer3` function. The `OpenTimer3` function also allows you to select the source (external/internal) of the clock.

The following table outlines the `#define`'s that exist in `timers16.h` header file for selecting an external or internal clock source:

#define	Clock Source
T3_SOURCE_EXT	External (I/O pin)
T3_SOURCE_INT	Internal (TOSC)

```
#include <timers16.h>

// Enable the TMR3 Overflow interrupt.
#ifdef _EXTERNAL_CLOCK
   OpenTimer3(TIMER_INT_ON | T3_SOURCE_EXT);
#else
   OpenTimer3(TIMER_INT_ON | T3_SOURCE_INT);
#endif
```

Modifying Register Bits

To select the source (external/internal) of the clock, you must clear (internal clock) or set (external clock input on the TCLK3 pin) the Timer3 Clock Source Select (TMR3CS) bit – [TCON1<2>].

```
#ifdef _EXTERNAL_CLOCK
   TCON1bits.TMR3CS = 1;
#else
   TCON1bits.TMR3CS = 0;
#endif
```

PIC17C7XX DEVICES

To enable the TMR3 overflow interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) TMR3 Interrupt Flag (TMR3IF) bit – [PIR1<6>].
- The Timer3 On (TMR3ON) bit – [TCON2<2>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of TMR3 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) TMR3 Interrupt Enable (TMR3IE) bit – [PIE1<6>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR3ON = 0;
PIR1bits.TMR3IF = 0;
PIE1bits.TMR3IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```


© 2002 Microchip Technology Inc.

PIC17C4X DEVICES

To enable the TMR3 overflow interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) TMR3 Interrupt Flag (TMR3IF) bit – [PIR<6>].
- The Timer3 On (TMR3ON) bit – [TCON2<2>]. This stops the timer until you are ready to use it. See below for how to start the timer.
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of TMR3 overflow interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) TMR3 Interrupt Enable (TMR3IE) bit – [PIE<6>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.TMR3ON = 0;
PIRbits.TMR3IF = 0;
PIEbits.TMR3IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

ALL PIC17 DEVICES

To reset Timer3, you must clear the TMR3L and TMR3H registers.

```
TMR3L = 0;
TMR3H = 0;
```

To start Timer3, set the Timer3 On (TMR3ON) bit – [TCON2<2>].

```
TCON2bits.TMR3ON = 1;
```

11.3.2.7 PORTB INTERRUPT ON CHANGE

There are two ways to enable the PORTB Interrupt on Change:

Library Call

To enable the PORTB interrupt on change through a library call, you must include the `port16.h` header file and call the `OpenPORTB` function.

```
#include <port16.h>
OpenPORTB(PORTB_CHANGE_INT_ON);
```

Modifying Register Bits

PIC17C7XX DEVICES

To enable the PORTB interrupt on change for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) PORTB Interrupt on Change Flag (RBIF) bit – [PIR1<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of PORTB interrupt. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) PORTB Interrupt on Change Enable (RBIE) bit – [PIE1<7>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR1bits.RBIF = 0;
PIE1bits.RBIE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the PORTB interrupt on change for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) PORTB Interrupt on Change Flag (RBIF) bit – [PIR<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of PORTB interrupt. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) PORTB Interrupt on Change Enable (RBIE) bit – [PIE<7>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIRbits.RBIF = 0;
PIEbits.RBIE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.8 CAPTURE1 INTERRUPT

There are two ways to enable the Capture1 Interrupt:

Library Call

To enable the Capture1 Interrupt through a library call, you must include the `captur16.h` header file and call the `OpenCapture1` function. The `OpenCapture1` function also allows you to select the edge on which the interrupt occurs.

The following table outlines the `#define`'s that exist in `captur16.h` header file for edge selection:

#define	Edge Selection
C1_EVERY_FALL_EDGE	Every falling edge
C1_EVERY_RISE_EDGE	Every rising edge
C1_EVERY_4_RISE_EDGE	Every fourth rising edge
C1_EVERY_16_RISE_EDGE	Every sixteenth rising edge

```
  #include <captur16.h>

// Enable the Capture1 Interrupt on the specified edge
#ifdef _TRIGGER_ON_EVERY_RISING_EDGE
  OpenCapture1(CAPTURE_INT_ON | C1_EVERY_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_FALLING_EDGE
  OpenCapture1(CAPTURE_INT_ON | C1_EVERY_FALL_EDGE);
#elif defined _TRIGGER_ON_EVERY_FOURTH_RISING_EDGE
  OpenCapture1(CAPTURE_INT_ON | C1_EVERY_4_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_SIXTEENTH_RISING_EDGE
  OpenCapture1(CAPTURE_INT_ON | C1_EVERY_16_RISE_EDGE);
#endif
```


© 2002 Microchip Technology Inc.

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on every rising or falling edge, every fourth rising edge, or every sixteenth rising edge. To select the edge, you must set the Capture1 Mode Select (CA1ED1:CA1ED0) bits – [TCON1<5>:TCON1<4>]. The following table outlines the values of these bits and the edge that is selected:

CA1ED1:CA1ED0	Edge Selection
00	Every falling edge
01	Every rising edge
10	Every fourth rising edge
11	Every sixteenth rising edge

To set edge selection to every rising edge, perform the following:

```
TCON1 = TCON1 & 0b11001111;  // Reset Edge Selection
TCON1 = TCON1 | 0b00010000;
```

To set edge selection to every fourth rising edge, perform the following:

```
TCON1 = TCON1 & 0b11001111;  // Reset Edge Selection
TCON1 = TCON1 | 0b00100000;
```

PIC17C7XX DEVICES

To enable the Capture1 interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) Capture1 Interrupt Flag (CA1IF) bit – [PIR1<2>].
- The Capture1 Overflow Status (CA1OVF) bit – [TCON2<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture1 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) Capture1 Interrupt Enable (CA1IE) bit – [PIE1<2>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.CA1OVF = 0;
PIR1bits.CA1IF = 0;
PIE1bits.CA1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the Capture1 interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) Capture1 Interrupt Flag (CA1IF) bit – [PIR<2>].
- The Capture1 Overflow Status (CA1OVF) bit – [TCON2<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture1 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) Capture1 Interrupt Enable (CA1IE) bit – [PIE<2>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.CA1OVF = 0;
PIRbits.CA1IF = 0;
PIEbits.CA1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.9 CAPTURE2 INTERRUPT

There are two ways to enable the Capture2 Interrupt:

Library Call

To enable the Capture2 Interrupt through a library call, you must include the `captur16.h` header file and call the `OpenCapture2` function. The `OpenCapture2` function also allows you to select the edge on which the interrupt occurs.

The following table outlines the `#define`'s that exist in `captur16.h` header file for edge selection:

#define	Edge Selection
C2_EVERY_FALL_EDGE	Every falling edge
C2_EVERY_RISE_EDGE	Every rising edge
C2_EVERY_4_RISE_EDGE	Every fourth rising edge
C2_EVERY_16_RISE_EDGE	Every sixteenth rising edge

```
#include <captur16.h>

// Enable the Capture2 Interrupt on the specified edge
#ifdef _TRIGGER_ON_EVERY_RISING_EDGE
  OpenCapture2(CAPTURE_INT_ON | C2_EVERY_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_FALLING_EDGE
  OpenCapture2(CAPTURE_INT_ON | C2_EVERY_FALL_EDGE);
#elif defined _TRIGGER_ON_EVERY_FOURTH_RISING_EDGE
  OpenCapture2(CAPTURE_INT_ON | C2_EVERY_4_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_SIXTEENTH_RISING_EDGE
  OpenCapture2(CAPTURE_INT_ON | C2_EVERY_16_RISE_EDGE);
#endif
```


© 2002 Microchip Technology Inc.

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on every rising or falling edge, every fourth rising edge, or every sixteenth rising edge. To select the edge, you must set the Capture2 Mode Select (CA2ED1:CA2ED0) bits – [TCON1<7>:TCON1<6>]. The following table outlines the values of these bits and the edge that is selected:

CA2ED1:CA2ED0	Edge Selection
00	Every falling edge
01	Every rising edge
10	Every fourth rising edge
11	Every sixteenth rising edge

To set edge selection to every rising edge, perform the following:

```
TCON1 = TCON1 & 0b00111111;  // Reset Edge Selection
TCON1 = TCON1 | 0b01000000;
```

To set edge selection to every fourth rising edge, perform the following:

```
TCON1 = TCON1 & 0b00111111;  // Reset Edge Selection
TCON1 = TCON1 | 0b10000000;
```

PIC17C7XX DEVICES

To enable the Capture2 interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) Capture2 Interrupt Flag (CA2IF) bit – [PIR1<3>].
- The Capture2 Overflow Status (CA2OVF) bit – [TCON2<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture2 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) Capture2 Interrupt Enable (CA2IE) bit – [PIE1<3>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.CA2OVF = 0;
PIR1bits.CA2IF = 0;
PIE1bits.CA2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the Capture2 interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) Capture2 Interrupt Flag (CA2IF) bit – [PIR<3>].
- The Capture2 Overflow Status (CA2OVF) bit – [TCON2<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture2 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) Capture2 Interrupt Enable (CA2IE) bit – [PIE<3>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON2bits.CA2OVF = 0;
PIRbits.CA2IF = 0;
PIEbits.CA2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.10 CAPTURE3 INTERRUPT

Note: PIC17C7XX devices only.

There are two ways to enable the Capture3 Interrupt:

Library Call

To enable the Capture3 Interrupt through a library call, you must include the `captur16.h` header file and call the `OpenCapture3` function. The `OpenCapture3` function also allows you to select the edge on which the interrupt occurs.

The following table outlines the `#define`'s that exist in `captur16.h` header file for edge selection:

#define	Edge Selection
C3_EVERY_FALL_EDGE	Every falling edge
C3_EVERY_RISE_EDGE	Every rising edge
C3_EVERY_4_RISE_EDGE	Every fourth rising edge
C3_EVERY_16_RISE_EDGE	Every sixteenth rising edge

```
#include <captur16.h>

// Enable the Capture3 Interrupt on the specified edge
#ifdef _TRIGGER_ON_EVERY_RISING_EDGE
  OpenCapture3(CAPTURE_INT_ON | C3_EVERY_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_FALLING_EDGE
  OpenCapture3(CAPTURE_INT_ON | C3_EVERY_FALL_EDGE);
#elif defined _TRIGGER_ON_EVERY_FOURTH_RISING_EDGE
  OpenCapture3(CAPTURE_INT_ON | C3_EVERY_4_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_SIXTEENTH_RISING_EDGE
  OpenCapture3(CAPTURE_INT_ON | C3_EVERY_16_RISE_EDGE);
#endif
```


© 2002 Microchip Technology Inc.

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on every rising or falling edge, every fourth rising edge, or every sixteenth rising edge. To select the edge, you must set the Capture3 Mode Select (CA3ED1:CA3ED0) bits – [TCON3<2>:TCON1<1>]. The following table outlines the values of these bits and the edge that is selected:

CA3ED1:CA3ED0	Edge Selection
00	Every falling edge
01	Every rising edge
10	Every fourth rising edge
11	Every sixteenth rising edge

To set edge selection to every rising edge, perform the following:

```
TCON3 = TCON3 & 0b11111001;  // Reset Edge Selection
TCON3 = TCON3 | 0b00000010;
```

To set edge selection to every fourth rising edge, perform the following:

```
TCON3 = TCON3 & 0b11111001;  // Reset Edge Selection
TCON3 = TCON3 | 0b00000100;
```

To enable the Capture3 interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) Capture3 Interrupt Flag (CA3IF) bit – [PIR2<2>].
- The Capture3 Overflow Status (CA3OVF) bit – [TCON3<5>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture3 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) Capture3 Interrupt Enable (CA3IE) bit – [PIE2<2>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON3bits.CA3OVF = 0;
PIR2bits.CA3IF = 0;
PIE2bits.CA3IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.11 CAPTURE4 INTERRUPT

Note: PIC17C7XX devices only.

There are two ways to enable the Capture4 Interrupt:

Library Call

To enable the Capture4 Interrupt through a library call, you must include the `captur16.h` header file and call the `OpenCapture4` function. The `OpenCapture4` function also allows you to select the edge on which the interrupt occurs.

The following table outlines the `#define`'s that exist in `captur16.h` header file for edge selection:

#define	Edge Selection
C4_EVERY_FALL_EDGE	Every falling edge
C4_EVERY_RISE_EDGE	Every rising edge
C4_EVERY_4_RISE_EDGE	Every fourth rising edge
C4_EVERY_16_RISE_EDGE	Every sixteenth rising edge

```
#include <captur16.h>

// Enable the Capture4 Interrupt on the specified edge
#ifdef _TRIGGER_ON_EVERY_RISING_EDGE
  OpenCapture4(CAPTURE_INT_ON | C4_EVERY_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_FALLING_EDGE
  OpenCapture4(CAPTURE_INT_ON | C4_EVERY_FALL_EDGE);
#elif defined _TRIGGER_ON_EVERY_FOURTH_RISING_EDGE
  OpenCapture4(CAPTURE_INT_ON | C4_EVERY_4_RISE_EDGE);
#elif defined _TRIGGER_ON_EVERY_SIXTEENTH_RISING_EDGE
  OpenCapture4(CAPTURE_INT_ON | C4_EVERY_16_RISE_EDGE);
#endif
```

Modifying Register Bits

This interrupt is edge triggered. You can specify whether this interrupt occurs on every rising or falling edge, every fourth rising edge, or every sixteenth rising edge. To select the edge, you must set the Capture4 Mode Select (CA4ED1:CA4ED0) bits – [TCON3<4>:TCON1<3>]. The following table outlines the values of these bits and the edge that is selected:

CA4ED1:CA4ED0	Edge Selection
00	Every falling edge
01	Every rising edge
10	Every fourth rising edge
11	Every sixteenth rising edge

To set edge selection to every rising edge, perform the following:

```
TCON3 = TCON3 & 0b11100111;  // Reset Edge Selection
TCON3 = TCON3 | 0b00001000;
```

To set edge selection to every fourth rising edge, perform the following:

```
TCON3 = TCON3 & 0b11100111;  // Reset Edge Selection
TCON3 = TCON3 | 0b00010000;
```


© 2002 Microchip Technology Inc.

To enable the Capture4 interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) Capture4 Interrupt Flag (CA4IF) bit – [PIR2<3>].
- The Capture4 Overflow Status (CA4OVF) bit – [TCON3<6>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of Capture4 interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) Capture4 Interrupt Enable (CA4IE) bit – [PIE2<3>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
TCON3bits.CA4OVF = 0;
PIR2bits.CA4IF = 0;
PIE2bits.CA4IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.12 USART1 TRANSMIT INTERRUPT

There are two ways to enable the USART1 Transmit Interrupt:

Library Call

To enable the USART1 Transmit Interrupt through a library call, you must include the `usart16.h` header file and call the `OpenUSART1` function. The `OpenUSART1` function also allows you to configure the USART1. For more information on how to configure the USART1, please refer to the MPLAB CXX Reference Guide (DS51224). The following code snippet only shows how you would go about enabling the USART1 Transmit Interrupt.

```
#include <usart16.h>

OpenUSART1(USART_TX_INT_ON, 25);
```

Modifying Register Bits

This section only discusses the bits necessary to enable the USART1 Transmit Interrupt. For information on configuring the USART1, please refer to the section in the appropriate data sheet on the Transmit Status and Control Register (TXSTA).

PIC17C7XX DEVICES

To enable the USART1 transmit interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) USART1 Transmit Interrupt Flag (TX1IF) bit – [PIR1<1>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART1 transmit interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) USART1 Transmit Interrupt Enable (TX1IE) bit – [PIE1<1>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR1bits.TX1IF = 0;
PIE1bits.TX1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

PIC17C4X DEVICES

To enable the USART1 transmit interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) USART1 Transmit Interrupt Flag (TX1IF) bit – [PIR<1>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART1 transmit interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) USART1 Transmit Interrupt Enable (TX1IE) bit – [PIE<1>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIRbits.TX1IF = 0;
PIEbits.TX1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.13 USART1 RECEIVE INTERRUPT

There are two ways to enable the USART1 Receive Interrupt:

Library Call

To enable the USART1 Receive Interrupt through a library call, you must include the `usart16.h` header file and call the `OpenUSART1` function. The `OpenUSART1` function also allows you to configure the USART1. For more information on how to configure the USART1, refer to the *MPLAB C17 C Compiler Libraries* (DS51296). The following code snippet only shows how to go about enabling the USART1 Receive Interrupt.

```
#include <usart16.h>

OpenUSART1(USART_RX_INT_ON, 25);
```

Modifying Register Bits

This section only discusses the bits necessary to enable the USART1 Receive Interrupt. For information on configuring the USART1, refer to the section in the appropriate data sheet on the Receive Status and Control Register (RCSTA).

PIC17C7XX DEVICES

To enable the USART1 receive interrupt for PIC17C7XX devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register1's (PIR1's) USART1 Receive Interrupt Flag (RC1IF) bit – [PIR1<0>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART1 receive interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register1's (PIE1's) USART1 Receive Interrupt Enable (RC1IE) bit – [PIE1<0>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR1bits.RC1IF = 0;
PIE1bits.RC1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```


© 2002 Microchip Technology Inc.

PIC17C4X DEVICES

To enable the USART1 receive interrupt for PIC17C4X devices, you must clear the following bit flags:

- The Peripheral Interrupt Request Register's (PIR) USART1 Receive Interrupt Flag (RC1IF) bit – [PIR<0>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART1 receive interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register's (PIE) USART1 Receive Interrupt Enable (RC1IE) bit – [PIE<0>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIRbits.RC1IF = 0;
PIEbits.RC1IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.14 USART2 TRANSMIT INTERRUPT

Note:	PIC17C7XX devices only.

There are two ways to enable the USART2 Transmit Interrupt:

Library Call

To enable the USART2 Transmit Interrupt through a library call, you must include the `usart16.h` header file and call the `OpenUSART2` function. The `OpenUSART2` function also allows you to configure the USART2. For more information on how to configure the USART2, refer to the *MPLAB C17 C Compiler Libraries* (DS51296). The following code snippet only shows how to go about enabling the USART1 Transmit Interrupt.

```
#include <usart16.h>

OpenUSART2(USART_TX_INT_ON, 25);
```

Modifying Register Bits

This section only discusses the bits necessary to enable the USART2 Transmit Interrupt. For information on configuring the USART2, please refer to the section in the appropriate data sheet on the Transmit Status and Control Register (TXSTA2).

To enable the USART2 transmit interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) USART2 Transmit Interrupt Flag (TX2IF) bit – [PIR2<1>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART2 transmit interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) USART2 Transmit Interrupt Enable (TX2IE) bit – [PIE2<1>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR2bits.TX2IF = 0;
PIE2bits.TX2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.15 USART2 RECEIVE INTERRUPT

Note: PIC17C7XX devices only.

There are two ways to enable the USART2 Receive Interrupt:

Library Call

To enable the USART2 Receive Interrupt through a library call, you must include the `usart16.h` header file and call the `OpenUSART2` function. The `OpenUSART2` function also allows you to configure the USART2. For more information on how to configure the USART2, refer to the *MPLAB C17 C Compiler Libraries* (DS51296). The following code snippet only shows how to go about enabling the USART2 Receive Interrupt.

```
#include <usart16.h>

OpenUSART2(USART_RX_INT_ON, 25);
```

Modifying Register Bits

This section only discusses the bits necessary to enable the USART2 Receive Interrupt. For information on configuring the USART2, please refer to the section in the PIC17C7XX Data Sheet on the Receive Status and Control Register (RCSTA2).

To enable the USART2 receive interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) USART2 Receive Interrupt Flag (RC2IF) bit – [PIR2<0>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of USART2 receive interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) USART2 Receive Interrupt Enable (RC2IE) bit – [PIE2<0>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR2bits.RC2IF = 0;
PIE2bits.RC2IE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.16 SYNCHRONOUS SERIAL PORT INTERRUPT

Note: PIC17C7XX devices only.

To enable the Synchronous Serial Port Interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) Synchronous Serial Port Interrupt Flag (SSPIF) bit – [PIR2<7>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of synchronous serial port interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) Synchronous Serial Port Interrupt Enable (SSPIE) bit – [PIE2<7>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].
  - PIR2bits.SSPIF = 0;
  - PIE2bits.SSPIE = 1;
  - INTSTAbits.PEIF = 0;
  - INTSTAbits.PEIE = 1;

© 2002 Microchip Technology Inc.

11.3.2.17 BUS COLLISION INTERRUPT *

> **Note:** PIC17C7XX devices only.

To enable the Bus Collision Interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) Bus Collision Interrupt Flag (BCLIF) bit – [PIR2<6>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of bus collision interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) Bus Collision Interrupt Enable (BCLIE) bit – [PIE2<6>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR2bits.BCLIF = 0;
PIE2bits.BCLIE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.3.2.18 A/D MODULE INTERRUPT

> **Note:** PIC17C7XX devices only.

There are two ways to enable the A/D Module Interrupt:

Library Call

To enable the A/D Module Interrupt through a library call, you must include the `adc16.h` header file and call the `OpenADC` function. The `OpenADC` function also allows you to configure the A/D converter. For more information on how to configure the A/D converter, refer to the *MPLAB C17 C Compiler Libraries* (DS51296). The following code snippet only shows how to go about enabling the A/D Module Interrupt.

```
#include <adc16.h>

OpenADC(ADC_INT_ON, ADC_CH_0);
```

Modifying Register Bits

This section only discusses the bits necessary to enable the A/D Module Interrupt. For information on configuring the A/D Converter, please refer to the section in the PIC17C7XX Data Sheet on the Analog-to-Digital converter.

To enable the A/D Module Interrupt, you must clear the following bit flags:

- The Peripheral Interrupt Request Register2's (PIR2's) A/D Module Interrupt Flag (ADIF) bit – [PIR2<5>].
- The INTSTA register's Peripheral Interrupt Flag (PEIF) bit – [INTSTA<7>]. This clears any previous occurrences of A/D module interrupts. In addition, you must set the following enable bits:
  - The Peripheral Interrupt Enable Register2's (PIE2's) A/D Module Interrupt Enable (ADIE) bit – [PIE2<5>].
  - The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

```
PIR2bits.ADIF = 0;
PIE2bits.ADIE = 1;
INTSTAbits.PEIF = 0;
INTSTAbits.PEIE = 1;
```

11.4 DISABLING INTERRUPTS

The PIC17 devices contain 18 sources of interrupts. See 11.3 "Enabling Interrupts" for a list of these interrupts.

All Interrupts on the PIC17 devices are initially disabled. The CPUSTA register contains the Global Interrupt Disable (GLINTD) bit. When this bit is set (value on RESET), all interrupts are disabled. You have the capability to:

- Disable global interrupts – disables all interrupts
- Disable individual interrupt(s)

11.4.1 Disabling Global Interrupts

Disabling global interrupts disables all interrupts regardless of their individual status. There are two ways to disable all interrupts on the PIC17 devices.

Library Call

To disable global interrupts, you must include the `int16.h` header file and call the `Disable` function.

```
#include <int16.h>

Disable();
```

Modifying Register Bits

To disable global interrupts, you must set the GLINTD bit of the CPUSTA register.

```
CPUSTAbits.GLINTD = 1;
```

11.4.2 Disabling Individual Interrupt(s)

This section explains how to disable each of the individual interrupts listed in 11.3 "Enabling Interrupts". Individual peripheral interrupts are disabled through bits in the Peripheral Interrupt Enable register (PIE) for PIC17C4X devices or in the Peripheral Interrupt Enable register 1 (PIE1) and the Peripheral Interrupt Enable register 2 (PIE2) for PIC17C7XX devices. The INTSTA register contains the disable bits for non-peripheral interrupts. We will explore these registers in more detail in the sections below.

11.4.2.1 EXTERNAL INTERRUPT ON T0CKI PIN

To disable the external interrupt on the T0CKI pin, you must clear the INTSTA register's External Interrupt on T0CKI Pin Enable (T0CKIE) bit – [INTSTA<2>].

```
INTSTAbits.T0CKIE = 0;
```

11.4.2.2 EXTERNAL INTERRUPT ON INT PIN

There are two ways to disable the external interrupt on the INT pin:

Library Call

To disable the external interrupt on the INT pin through a library call, you must include the `int16.h` header file and call the `CloseRA0INT` function.

```
#include <int16.h>

CloseRA0INT();
```


© 2002 Microchip Technology Inc.

Modifying Register Bits

To disable the external interrupt on the INT pin, you must clear the INTSTA register's External Interrupt on RA0/INT Pin Enable (INTE) bit – [INTSTA<0>].

```
INTSTAbits.INTE = 0;
```

11.4.2.3 TMR0 OVERFLOW INTERRUPT

There are two ways to disable the TMR0 overflow interrupt:

Library Call

To disable the TMR0 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `CloseTimer0` function.

```
#include <timers16.h>

CloseTimer0();
```

Modifying Register Bits

To disable the TMR0 overflow interrupt, you must clear the INTSTA register's TMR0 Overflow Interrupt Enable (T0IE) bit – [INTSTA<1>].

```
INTSTAbits.T0IE = 0;
```

11.4.2.4 TMR1 OVERFLOW INTERRUPT

There are two ways to disable the TMR1 overflow interrupt:

Library Call

To disable the TMR1 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `CloseTimer1` function

```
#include <timers16.h>

CloseTimer1();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the TMR1 overflow interrupt for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) TMR1 Interrupt Enable (TMR1IE) bit – [PIE1<4>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.TMR1IE = 0;  //Disable all
                      //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the TMR1 overflow interrupt for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) TMR1 Interrupt Enable (TMR1IE) bit – [PIE<4>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.TMR1IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.5 TMR2 OVERFLOW INTERRUPT

There are two ways to disable the TMR2 overflow interrupt:

Library Call

- To disable the TMR2 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `CloseTimer2` function.

```
#include <timers16.h>

CloseTimer2();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the TMR2 overflow interrupt for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) TMR2 Interrupt Enable (TMR2IE) bit – [PIE1<5>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.TMR2IE = 0; //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the TMR2 overflow interrupt for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) TMR2 Interrupt Enable (TMR2IE) bit – [PIE<5>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.TMR2IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```


© 2002 Microchip Technology Inc.

11.4.2.6 TMR3 OVERFLOW INTERRUPT

There are two ways to disable the TMR3 overflow interrupt:

Library Call

To disable the TMR3 Overflow Interrupt through a library call, you must include the `timers16.h` header file and call the `CloseTimer3` function.

```
#include <timers16.h>

CloseTimer3();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the TMR3 overflow interrupt for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) TMR3 Interrupt Enable (TMR3IE) bit – [PIE1<6>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.TMR3IE = 0; //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the TMR3 overflow interrupt for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) TMR3 Interrupt Enable (TMR3IE) bit – [PIE<6>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.TMR3IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.7 PORTB INTERRUPT ON CHANGE

There are two ways to disable the PORTB Interrupt on Change:

Library Call

To disable the PORTB interrupt on change through a library call, you must include the `port16.h` header file and call the `ClosePORTB` function.

```
#include <port16.h>

ClosePORTB();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the PORTB interrupt on change for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) PORTB Interrupt on Change Enable (RBIE) bit – [PIE1<7>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.RBIE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the PORTB interrupt on change for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) PORTB Interrupt on Change Enable (RBIE) bit – [PIE<7>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.RBIE = 0;    //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.8 CAPTURE1 INTERRUPT

There are two ways to disable the Capture1 Interrupt:

Library Call

To disable the Capture1 Interrupt through a library call, you must include the `captur16.h` header file and call the `CloseCapture1` function.

```
#include <captur16.h>

CloseCapture1();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the Capture1 interrupt on PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) Capture1 Interrupt Enable (CA1IE) bit – [PIE1<2>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.CA1IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```


© 2002 Microchip Technology Inc.

PIC17C4X DEVICES

To disable the Capture1 interrupt on PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) Capture1 Interrupt Enable (CA1IE) bit – [PIE<2>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.CA1IE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.9 CAPTURE2 INTERRUPT

There are two ways to disable the Capture2 Interrupt:

Library Call

To disable the Capture2 Interrupt through a library call, you must include the `captur16.h` header file and call the `CloseCapture2` function.

```
#include <captur16.h>

CloseCapture2();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the Capture2 interrupt on PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) Capture2 Interrupt Enable (CA2IE) bit – [PIE1<3>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.CA2IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the Capture2 interrupt on PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) Capture2 Interrupt Enable (CA2IE) bit – [PIE<3>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.CA2IE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.10 CAPTURE3 INTERRUPT

> **Note:** Only pertains to PIC17C7XX devices. Does not apply to PIC17C4X devices.

There are two ways to disable the Capture3 Interrupt:

Library Call

To disable the Capture3 Interrupt through a library call, you must include the `captur16.h` header file and call the `CloseCapture3` function

```
#include <captur16.h>

CloseCapture3();
```

Modifying Register Bits

To disable the Capture3 interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) Capture3 Interrupt Enable (CA3IE) bit – [PIE2<2>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.CA3IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.11 CAPTURE4 INTERRUPT

> **Note:** Only pertains to PIC17C7XX devices. Does not apply to PIC17C4X devices.

There are two ways to disable the Capture4 Interrupt:

Library Call

To disable the Capture4 Interrupt through a library call, you must include the `captur16.h` header file and call the `CloseCapture4` function.

```
#include <captur16.h>

CloseCapture4();
```

Modifying Register Bits

To disable the Capture4 interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) Capture4 Interrupt Enable (CA4IE) bit – [PIE2<3>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.CA4IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```


© 2002 Microchip Technology Inc.

11.4.2.12 USART1 TRANSMIT INTERRUPT

There are two ways to disable the USART1 Transmit Interrupt:

Library Call

To disable the USART1 Transmit Interrupt through a library call, you must include the `usart16.h` header file and call the `CloseUSART1` function.

```
#include <usart16.h>

CloseUSART1();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the USART1 transmit interrupt for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) USART1 Transmit Interrupt Enable (TX1IE) bit – [PIE1<1>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.TX1IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the USART1 transmit interrupt for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) USART1 Transmit Interrupt Enable (TX1IE) bit – [PIE<1>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.TX1IE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.13 USART1 RECEIVE INTERRUPT

There are two ways to disable the USART1 Receive Interrupt:

Library Call

To disable the USART1 Receive Interrupt through a library call, you must include the `usart16.h` header file and call the `CloseUSART1` function.

```
#include <usart16.h>

CloseUSART1();
```

Modifying Register Bits

PIC17C7XX DEVICES

To disable the USART1 receive interrupt for PIC17C7XX devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register1's (PIE1's) USART1 Receive Interrupt Enable (RC1IE) bit – [PIE1<0>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE1bits.RC1IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

PIC17C4X DEVICES

To disable the USART1 receive interrupt for PIC17C4X devices, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register's (PIE) USART1 Receive Interrupt Enable (RC1IE) bit – [PIE<0>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIEbits.RC1IE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.14 USART2 TRANSMIT INTERRUPT

Note: PIC17C7XX devices only.

There are two ways to disable the USART2 Transmit Interrupt:

Library Call

To disable the USART2 Transmit Interrupt through a library call, you must include the `usart16.h` header file and call the `CloseUSART2` function.

```
#include <usart16.h>

CloseUSART2();
```

Modifying Register Bits

To disable the USART2 transmit interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) USART2 Transmit Interrupt Enable (TX2IE) bit – [PIE2<1>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

Note: Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.TX2IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```


© 2002 Microchip Technology Inc.

11.4.2.15 USART2 RECEIVE INTERRUPT

> **Note:** PIC17C7XX devices only.

There are two ways to disable the USART2 Receive Interrupt:

Library Call

To disable the USART2 Receive Interrupt through a library call, you must include the `usart16.h` header file and call the `CloseUSART2` function.

```
#include <usart16.h>

CloseUSART2();
```

Modifying Register Bits

To disable the USART2 receive interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) USART2 Receive Interrupt Enable (RC2IE) bit – [PIE2<0>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.RC2IE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.16 SYNCHRONOUS SERIAL PORT INTERRUPT

> **Note:** Only pertains to PIC17C7XX devices. Does not apply to PIC17C4X devices.

To disable the Synchronous Serial Port Interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) Synchronous Serial Port Interrupt Enable (SSPIE) bit – [PIE2<7>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.SSPIE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.17 BUS COLLISION INTERRUPT

> **Note:** Only pertains to PIC17C7XX devices. Does not apply to PIC17C4X devices.

To disable the Bus Collision Interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) Bus Collision Interrupt Enable (BCLIE) bit – [PIE2<6>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.BCLIE = 0;  //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```

11.4.2.18 A/D MODULE INTERRUPT

> **Note:** PIC17C7XX devices only.

There are two ways to disable the A/D Module Interrupt:

Library Call

To disable the A/D Module Interrupt through a library call, you must include the `adc16.h` header file and call the `CloseADC` function.

```
#include <adc16.h>

CloseADC();
```

Modifying Register Bits

To disable the A/D Module Interrupt, you must clear either of the following enable bits:

- The Peripheral Interrupt Enable Register2's (PIE2's) A/D Module Interrupt Enable (ADIE) bit – [PIE2<5>].
- The INTSTA register's Peripheral Interrupt Enable (PEIE) bit – [INTSTA<3>].

> **Note:** Clearing the PEIE bit disables all peripheral interrupts.

```
PIE2bits.ADIE = 0;   //Disable all
                     //peripheral interrupts
INTSTAbits.PEIE = 0;
```


© 2002 Microchip Technology Inc.

Chapter 12. Implementation-Defined Behavior

12.1 INTRODUCTION

This section describes the behavior of MPLAB C17 where the ANSI standard X3.159-1989 describes the behavior as *implementation defined.* The text below in italic font is taken directly from the ANSI standard with the appropriate section in parentheses.

12.2 HIGHLIGHTS

This chapter covers ANSI implementation issues for the following categories:

- Identifiers
- Characters
- Integers
- Floating Point
- Arrays and Pointers
- Registers
- Structures and Unions
- Bit-Fields
- Enumerations
- Switch Statements
- Preprocessor Directives

12.3 IDENTIFIERS

The number of significant initial characters (beyond 31) in an identifier without external linkage (3.1.2)

The number of significant initial characters (beyond 6) in an identifier with external linkage (3.1.2)

Whether case distinctions are significant in an identifier with external linkage (3.1.2)

All MPLAB C17 identifiers have 31 significant characters. Case distinctions are significant in an identifier with external linkage.

12.4 CHARACTERS

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (3.1.3.4)

The value of the integer character constant is the 8-bit value of the first character. Wide characters are not supported.

Whether a 'plain' char has the same range of values as signed char or unsigned char (3.2.1.1)

A 'plain' `char` has the same range of values as a `signed char`.

12.5 INTEGERS

A 'char', a 'short int', or and 'int' bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an 'int' or 'unsigned int' may be used. If an 'int' can represent all values of the original type, the value is converted to an 'int'; otherwise, it is converted to an 'unsigned int'. These are called the "integral promotions." All other arithmetic types are unchanged by the integral promotions. The integral promotions preserve value including sign. (3.2.1.1)

The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (3.2.1.2)

When converting from a larger integer type to a smaller integer type, the high order bits of the value are discarded and the remaining bits are interpreted according to the type of the smaller integer type. When converting from an unsigned integer to a signed integer of equal size, the bits of the unsigned integer are simply reinterpreted according to the rules for a signed integer of that size.

The results of bitwise operations on signed integers (3.3)

The bitwise operators are applied to the signed integer as if it were an unsigned integer of the same type (i.e., the sign bit is treated as any other bit).

The sign of the remainder on integer division (3.3.5)

The remainder has the same sign as the quotient.

The result of a right shift of a negative-valued signed integral type (3.3.7)

The value is shifted as if it were an unsigned integral type of the same size (i.e., the sign bit is not propagated).

12.6 FLOATING POINT

The representations and sets of values of the various types of floating point numbers (3.1.2.5)

The direction of truncation when an integral number is converted to a floating point number that cannot exactly represent the original value (3.2.1.3)

The direction of truncation or rounding when a floating point number is converted to a narrower floating point number (3.2.1.4)

The rounding to the nearest method is used.

© 2002 Microchip Technology Inc.

12.7 ARRAYS AND POINTERS

The type of integer required to hold the maximum size of an array – that is, the type of the size of operator, size_t (3.3.3.4, 4.1.1)

`size_t` is defined as an `unsigned int`.

The result of casting a pointer to an integer, or vice-versa (3.3.4)

The integer will contain the binary value used to represent the pointer. If the pointer is larger than the integer, the representation will be truncated to fit in the integer.

The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff_t (3.3.6, 4.1.1)

`ptrdiff_t` is defined as an `unsigned int`.

12.8 REGISTERS

The extent to which objects can actually be placed in registers by use of the register storage class specifier (3.5.1)

The `register` storage class specifier is ignored.

12.9 STRUCTURES AND UNIONS

A member of a union object is accessed using a member of a different type (3.3.2.3)

The value of the member is the bits residing at the location for the member interpreted as the type of the member being accessed.

The padding and alignment of members of structures (3.5.2.1)

Members of structures and unions are aligned on byte boundaries.

12.10 BIT-FIELDS

Whether a 'plain' int bit-field is treated as a signed int or as an unsigned int bit-field (3.5.2.1)

A 'plain' `int` bit-field is treated as an `unsigned int` bit-field.

The order of allocation of bit-fields within a unit (3.5.2.1)

Bit-fields are allocated from least significant bit to most significant bit in order of occurrence.

Whether a bit-field can straddle a storage-unit boundary (3.5.2.1)

A bit-field cannot straddle a storage unit boundary.

12.11 ENUMERATIONS

The integer type chosen to represent the values of an enumeration type (3.5.2.2)

`signed int` is used to represent the values of an enumeration type.

12.12 SWITCH STATEMENTS

The maximum number of case values in a switch statement (3.6.4.2)

The maximum number of values is limited only by target memory.

12.13 PREPROCESSING DIRECTIVES

The method for locating includable source files (3.8.2)

Includable source files specified via the `#include <filename>` mechanism are searched for in the path specified in the MCC_INCLUDE environment variable. The MCC_INCLUDE environment variable contains a semi-colon delimited list of directories to search.

The support for quoted names for includable source files (3.8.2)

Includable source files specified via the `#include "filename"` mechanism are searched for in the current directory and then in the path specified in the MCC_INCLUDE environment variable. The MCC_INCLUDE environment variable contains a semi-colon delimited list of directories to search.

The behavior on each recognized #pragma directive (3.8.6)

Each `#pragma` directive is listed in 2.6 "Statement Differences".

© 2002 Microchip Technology Inc.

Chapter 13. MPLAB C17 Diagnostics

13.1 INTRODUCTION

This appendix lists diagnostic messages generated by the MPLAB C17 compiler.

13.2 HIGHLIGHTS

The following diagnostic types apply to MPLAB C17:

- Errors
- Warnings

13.3 ERRORS

1000: argument count mismatch in function call

To call a function, the number of arguments passed must match exactly the number of parameters declared for the function.

1001: type mismatch in argument %d

The type of an argument to a function call must be compatible with the declared type of the corresponding parameter

1002: arithmetic type expected in expression

The operator requires that its operand be of arithmetic type

1003: arithmetic or pointer to object type required

1004: array must have integral constant size

1005: object of pointer type required for [ ] operator

The array access operator, '[ ]', requires that one operand be a pointer and the other be an integer, that is, for 'x[y]' the expression '*(x+y)' must be valid. 'x[y]' is functionally equivalent to '*(x+y)'.

1006: '–>' requires pointer to struct or union

The member access operator '–>' requires operands of pointer to struct/union.

1007: call of non-function

The operand of the '()' function call post-fix operator must be of type 'pointer to function.' Most commonly, this is a function identifier. Common causes include missing scope parentheses.

1008: cannot modify 'const' qualified object

An object qualified with 'const' is declared to be read-only data and modifications to it are therefore not allowed.

1009: cannot return an object of array type

1010: unable to locate include file '%s'

The compiler was unable to locate the '%s' file. Common causes include misspelled file '%s' and misconfigured include path.

1011: unable to open include file '%s'

The compiler was unable to open the '%s'd file. Common causes include misspelled file'%s' and insufficient access rights

1012: ')' expected in macro definition

A closing parenthesis is missing in the definition of a macro.

1013: constant expression required

1014: ')' expected

A closing parenthesis is missing.

1015: '%s'

source code '#error' directive message

1016: divide by zero in constant expression

The compiler cannot process a constant expression which contains a divide by (or modulus by) zero.

1017: divide by constant zero in expression

The compiler cannot process an expression which contains a divide by (or modulus by) constant zero.

1018: '.' requires struct or union

The member access operator '.' requires operands of struct/union.

1019: duplicate case label value %d

1020: duplicate declaration for symbol '%s'

1021: duplicate label '%s'

1022: #elif in #else clause not allowed

1023: #elif without #if

1024: #else without #if

1025: #endif without #if

1026: extra 'default' statement in switch

A switch statement can only have a single 'default' label. Common causes include a missing '}' to close an inner switch.

1027: extraneous input following '%s'

1028: 'high' and 'low' are not valid in this context

1029: identifier expected

1030: member access on incomplete structure type '%s'

1031: initializer count mismatch for '%s'

1032: initializer list required for '%s'

1034: type mismatch in initializer

1035: value expected in initializer

1036: in-line assembly must be within a function body

1037: integer constant expected

1038: integer type required

Bitwise operators require that both operands be of integer type. Common causes include a missing '*' or '[ ]' operator.

© 2002 Microchip Technology Inc.

1040: invalid character constant

1041: invalid expression in assembly statement

1042: invalid member of structure ‘%s’

1043: invalid storage class in parameter %d

1044: lvalue required

An expression which designates an object is required. Common causes include missing parentheses and a missing ‘*’ operator.

1045: argument count mismatch invoking macro ‘%s’

1046: identifier expected in macro definition

1047: missing argument %d in macro ‘%s’

1048: misplaced ‘break’ statement

A ‘break’ statement must be inside a ‘while’, ‘do’, ‘for’ or ‘switch’ statement. Common causes include a misplaced ‘}’.

1049: misplaced ‘continue’ statement

A ‘continue’ statement must be inside a ‘while’, ‘do’, ‘for’ or ‘switch’ statement.

1050: missing ‘)’ in macro invocation on line %d

1051: missing #endif

1052: multiple ‘#else’ clauses for ‘#if’ not allowed

1053: cannot use ‘%s’ twice in same declaration

1054: cannot use type twice in same declaration

1055: must have constant operand for 1-bit quantity

1056: must have constant operand for 3-bit quantity

1057: identifier expected following ‘%s’

1058: pointer operand required for ‘*’ operator

The ‘*’ dereference operator requires a pointer to a non-void object as its operand

1059: syntax error: Expecting second parameter

1060: hardware multiply is not supported on the 17c42

1061: pragma error: bank type specified for ROM section

1062: block assignments must be four bytes or smaller

1064: 32-bit integers not supported

1065: invalid assembly instruction

1066: variable length argument lists not supported

1067: number of parameters conflicts with previous definition

1068: old style function definitions not supported

1069: ‘(’ expected in macro invocation

1070: operator %c requires arithmetic operands

1071: operator ‘%s’ requires arithmetic operands

1072: operator %c requires integral operands

1073: parameter ‘%s’ type mismatch

1074: cannot cast a pointer’s location qualifier

1075: error in pragma directive

1076: redundant section modifier '%s' in pragma

1077: definition '%s' does not match prototype

1078: conflicting qualifiers specified

1079: redeclaration of '%s' does not match first

1080: scalar operand required

A conditional statement control expression must be of scalar type, (i.e., an integer or a pointer).

1081: section address permitted only at definition

1082: section pragma not allowed inside a function

1083: section overlay attribute does not match definition

1084: section share attribute does not match definition

1085: section type does not match definition

1086: shift by a negative value

1087: static function '%s' missing definition

1088: conflicting storage classes specified

1089: structure, Union or Enum type mismatch '%s'

1090: switch expression must be 8-bit

1091: symbol '%s' already defined

1092: syntax error

1093: syntax error, expecting string

1094: conflicting types specified

1095: type declarator mismatch

1096: type location mismatch

1097: type mismatch

The type of the return value is not compatible with the declared return type of the function. Common causes include a missing '*' or '[ ]' operator

1098: type mismatch in redeclaration of '%s'

The type of the symbol declared is not compatible with the type of a previous declaration of the same symbol. Common causes include missing qualifiers or misplaced qualifiers.

1099: type qualifier mismatch

1100: type range mismatch

1101: type storage class mismatch

1102: undefined symbol '%s'

A symbol has been referenced before it has been defined. Common causes include a misspelled symbol '%s', a missing header file which declares the symbol, and a reference to a symbol valid only in an inner scope.

1103: unexpected input after '%s'

1104: unknown preprocessor directive '%s'

© 2002 Microchip Technology Inc.

1105: unresolved label ‘%s’

The label has been referenced via a ‘goto’ statement, but has not been defined in the function. Common causes include a misspelled label identifier and a reference to an out of scope label, (i.e., a label defined in another function).

1106: bit field type must be integer

1107: section #pragmas not allowed inside functions

13.4 WARNINGS

2000: no prototype for ‘%s’

2001: shift by zero

2002: shift by more bits than contained in operand

2003: ‘rom’ and ‘volatile’ in same declaration

2004: unused symbol block

2005: redefinition of macro ‘%s’ is not identical

2006: call of function ‘%s’ without prototype

A function call has been made without an in-scope function prototype for the function being called. This can be un-safe, as no type-checking for the function arguments can be performed.

2007: unknown pragma ‘%s’ encountered

NOTES:

© 2002 Microchip Technology Inc.

Section 4 – Appendices

Appendix A. Reference Documents.....125
Appendix B. Example Programs.....127
Appendix C. ASCII Character Set.....129

© 2002 Microchip Technology Inc.

Appendix A. Reference Documents

A.1 INTRODUCTION

This appendix gives references that may be helpful in programming with MPLAB C17.

A.2 HIGHLIGHTS

This appendix lists the following reference types:

- C Standards Information
- General C Information

A.3 C STANDARDS INFORMATION

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability and efficient execution of C language programs on a variety of computing systems.

A.4 GENERAL C INFORMATION

Harbison, Samuel P., and Steele, Guy L., *C A Reference Manual*, Fourth Edition, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, New Jersey 07632.

Presents a concise exposition of C as defined by the ANSI standard. This book is an excellent reference for C programmers.

Kochan, Steven G. *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

Another excellent reference for learning ANSI C, used in colleges and universities.

A best selling authoritative reference for the C programming language.

Van Sickle, Ted. *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

Although this book focuses on Motorola® microcontrollers, the basic principles of programming with C for microcontrollers is useful.

NOTES:

© 2002 Microchip Technology Inc.

Appendix B. Example Programs

B.1 INTRODUCTION

This chapter gives an overview of the MPLAB C17 example programs included with the compiler program and support files.

B.2 HIGHLIGHTS

The contents of this chapter are as following:

- Overview of Example Files
- Example Details

B.3 OVERVIEW OF EXAMPLE FILES

Example files may be found in the `examples` directory after you have installed MPLAB C17. The examples included at the time this document was published are contained in subdirectories as follows:

- General Examples
  - `Example1`
  - `Example2`
  - `Example3`
- Peripheral-Specific Examples
  - `AD`
  - `INT`
  - `LINK`
  - `PORT`
  - `PWM`
  - `TABLE_R/W`
  - `USART`

Additions, deletions or other changes to this list may have occurred. Check the `readme.txt` in the `examples` directory for more information on what examples are available and a brief description of the function of each example.

B.4 EXAMPLE DETAILS

The types of files typically found in an example subdirectory are as follows:

- Source files (.c, .asm) – the main program files.
- Batch files (.bat) – for use with command-line applications.

Additional files will be necessary to build the example into an application.

From `c:\mcc\h:`

- Header files (.h) – include files with device register definitions.

From `c:\mcc\lib:`

- Precompiled object files (.o) – provide "canned" start-up code, initialization code, interrupt service routines (for MPLAB C17) and register definitions, based on device and memory model used.
- Library files (.lib) – include microchip libraries.

From `c:\mcc\lkr:`

- Linker script files (.lkr) – directions for the linker, based on device.

To build the application, follow either the instructions for building on the command line (Chapter 4) or using MPLAB IDE (*MPLAB IDE User's Guide* DS51025).

Note: When linking, you may get the following message:

"Warning – Could not open source file '<filename>'.
This file will not be present in the list file."

This comes from using precompiled libraries, where the source for these libraries is not in the default directory (`c:\mcc\src`).

© 2002 Microchip Technology Inc.

Appendix C. ASCII Character Set

Most Significant Character

Least Significant Character

Hex	0	1	2	3	4	5	6	7
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NOTES:

© 2002 Microchip Technology Inc.

Glossary

A

Absolute Section (MPLINK Linker)

A section with a fixed (absolute) address that can not be changed by the linker.

Access RAM – PIC18 Devices Only

Special general purpose registers on PIC18 devices that allow access regardless of the setting of the bank select bit (BSR).

Alpha Character

Alpha characters are those characters that are letters of the arabic alphabet (a, b, …, z, A, B, …, Z).

ANSI

American National Standards Institute, which is an organization responsible for formulating and approving computer-related standards in the United States.

Alphanumeric

Alphanumeric characters are comprised of alpha characters and decimal digits (0,1, …, 9).

Application

A set of software and hardware usually designed to be a product controlled by a PICmicro® microcontroller.

ASCII

American Standard Code for Information Interchange is character set encoding using 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembler (Assemblers)

A language tool that translates assembly source code into machine code.

Assembly Language (Assemblers)

A programming language that is once removed from machine language. Machine languages consist entirely of numbers and are difficult for humans to read and write. Assembly languages enable a programmer to use names (mnemonics) instead of numbers.

Assigned Section (MPLINK™ Linker)

A section which has been assigned to a target memory block in the linker command file. The linker allocates an assigned section into its specified target memory block.

Asynchronous Stimulus (Simulators)

Data generated to simulate external inputs to a simulator device.

B

Breakpoint – Hardware (MPLAB® ICE 2000, MPLAB ICD, MPLAB ICD 2)

An event whose execution will cause a halt.

Breakpoint – Software (Debuggers)

An address where execution of the firmware will halt. Usually achieved by a special break opcode.

Build (MPLAB® IDE v5.xx/v6.xx)

The compilation and linking of all the source files for an application.

C

C (Compilers)

A high level programming language that may be used to develop applications for microcontrollers, especially high-end device families.

Calibration Memory

A special function register or registers used to hold values for calibration of a PICmicro® microcontroller on-board RC oscillator or other device peripherals.

COFF (MPLAB ASM30, Linkers)

Common Object File Format. An object file format that contains machine code and debugging information.

Command Line Interface

Command line interface refers to executing a program on the command line with options.

Compiler (Compilers)

A language tool that translates source code into assembly code.

Configuration Bits

Special-purpose bits programmed to set PICmicro® microcontroller modes of operation. A configuration bit may or may not be preprogrammed.

Control Directives (Assemblers)

Control directives in an assembler permit code to be conditionally assembled.

Cross Reference File (Linkers)

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives (Assemblers)

Data directives are those that control the assembler's allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

On a PICmicro MCU device, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

© 2002 Microchip Technology Inc.

Directives

Directives provide control of the language tool's operation.

Download

Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

DSC

See Digital Signal Controller.

DSP

See Digital Signal Processing.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Emulation (MPLAB ICE 2000)

The process of executing software loaded into emulation memory as if it were firmware residing on a microcontroller device.

Emulation Memory (MPLAB ICE 2000)

Program memory contained within the emulator.

Emulator (MPLAB ICE 2000)

Hardware that performs emulation.

Emulator System (MPLAB ICE 2000)

The MPLAB ICE 2000 emulator system includes the pod, processor module, device adapter, cables and MPLAB IDE software.

Event (MPLAB IDE v5.xx/v6.xx)

A description of a bus cycle which may include address, data, pass count, external input, cycle type (fetch, R/W) and time stamp. Events are used to describe triggers and breakpoints.

Executable Code

Software that is ready to be loaded for execution.

Export (MPLAB IDE v5.xx/v6.xx)

Send data out of the MPLAB IDE in a standardized format.

Expressions

Expressions are used in the operand field of the source line and may contain constants, symbols, or combinations of constants and/or symbols separated by arithmetic or logical operators. Each constant or symbol may be preceded by a plus or minus to indicate a positive or negative expression.

Extended Microcontroller Mode – PIC17 and PIC18 Devices Only

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC17 or PIC18 device.

External Input Line (MPLAB ICE 2000)

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

External Label (Linkers)

A label that has external linkage.

External Linkage (Linkers)

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

External RAM – PIC17 and PIC18 Devices Only

Off-chip Read/Write memory.

External Symbol (Linkers)

A symbol for an identifier which has external linkage.

External Symbol Definition (Linkers)

A symbol for a function or variable defined in the current module.

External Symbol Reference (Linkers)

A symbol which references a function or variable defined outside the current module.

External Symbol Resolution (Linkers)

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

F

File Registers

On-chip general purpose and special function registers.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PICmicro microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

G

GPR

General Purpose Register. The portion of PICmicro MCU data memory (RAM) available for general use, e.g., program-specific variables.

© 2002 Microchip Technology Inc.

H

Halt (MPLAB IDE v5.xx/v6.xx)

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

HEX Code

Executable instructions assembled or compiled from source code into hexadecimal format code. HEX code is contained in a HEX file.

HEX File

An ASCII file containing hexadecimal addresses and values (HEX code) suitable for programming a device.

High Level Language (Language Tools)

A language for writing programs that is of a higher level of abstraction from the processor than assembly code. High level languages (such as C) employ a compiler to translate statements into machine instructions that the target processor can execute.

I

ICD

In-Circuit Debugger. MPLAB ICD and MPLAB ICD 2 are Microchip's in-circuit debuggers for PIC16F87X and PIC18FXXXX devices, respectively. These ICDs work with MPLAB IDE.

ICE

In-Circuit Emulator. MPLAB ICE 2000 is Microchip's in-circuit emulator that works with MPLAB IDE. PICMASTER (Obsolete product) and ICEPIC (Third Party product) are other ICE devices.

IDE

Integrated Development Environment. A software application that is used for firmware development. The MPLAB IDE integrates a project manager, an editor, language tools, debug tools, programmers and an assortment of other tools within one Windows® application. A user developing an application can write code, compile, debug and test an application without leaving the MPLAB IDE desktop.

Identifier

A function or variable name.

Import (MPLAB IDE v5.xx/v6.xx)

Bring data into the MPLAB IDE from an outside source, such as from a HEX file.

Initialized Data (Language Tools)

Data which is defined with an initial value. In C,

```
int myVar=5
```

defines a variable which will reside in an initialized data section.

Instructions (Language Tools)

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Instruction Set (Language Tools)

The collection of machine language instructions that a particular processor understands.

Internal Linkage (Linkers)

A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications.

Interrupt

An asynchronous event that suspends normal processing and temporarily diverts the flow of control through an "interrupt handler" routine.

Interrupts may be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call or trap instruction).

In general the computer responds to an interrupt by storing the information about the current state of the running program; storing information to identify the source of the interrupt; and invoking a first-level interrupt handler. This is usually a kernel level privileged process that can discover the precise cause of the interrupt (e.g. if several devices share one interrupt) and what must be done to keep operating system tables (such as the process table) updated. This first-level handler may then call another handler, e.g. one associated with the particular device which generated the interrupt.

Interrupt Handler

A routine which is executed when an interrupt occurs. Interrupt handlers typically deal with low-level events in the hardware of a computer system such as a character arriving at a serial port or a tick of a real-time clock. Special care is required when writing an interrupt handler to ensure that either the interrupt which triggered the handler's execution is masked out (inhibited) until the handler is done, or the handler is written in a re-entrant fashion so that multiple concurrent invocations will not interfere with each other.

If interrupts are masked then the handler must execute as quickly as possible so that important events are not missed. This is often arranged by splitting the processing associated with the event into "upper" and "lower" halves. The lower part is the interrupt handler which masks out further interrupts as required, checks that the appropriate event has occurred (this may be necessary if several events share the same interrupt), services the interrupt, e.g. by reading a character from a UART and writing it to a queue, and re-enabling interrupts.

The upper half executes as part of a user process. It waits until the interrupt handler has run. Normally the operating system is responsible for reactivating a process which is waiting for some low-level event. It detects this by a shared flag or by inspecting a shared queue or by some other synchronization mechanism. It is important that the upper and lower halves do not interfere if an interrupt occurs during the execution of upper half code. This is usually ensured by disabling interrupts during critical sections of code such as removing a character from a queue.

Interrupt Request

The name of an input found on many processors which causes the processor to suspend normal instruction execution temporarily and to start executing an interrupt handler routine. Such an input may be either "level sensitive" – the interrupt condition will persist as long as the input is active or "edge triggered" – an interrupt is signaled by a low-to-high or high-to-low transition on the input. Some processors have several interrupt request inputs allowing different priority interrupts.

© 2002 Microchip Technology Inc.

Interrupt Service Routine

User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

IRQ

See Interrupt Request.

ISO

See International Organization for Standardization.

ISR

See Interrupt Service Routine.

L

Librarian (Librarians)

A language tool that creates and manipulates libraries.

Library (Librarians)

A library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the librarian to combine the object files into one library file. A library can be linked with object modules and other libraries to create executable code.

Linker (Linkers)

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files (Linkers)

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives (Assemblers)

Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

Listing File (Assemblers)

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

Logic Probes (MPLAB ICE 2000)

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V and a common ground.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings. The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated. Also called machine code.

Macro (Assemblers)

A collection of assembler instructions that are included in the assembly code when the macro name is encountered in the source code. Macros must be defined before they are used; forward references to macros are not allowed.

All statements following a `MACRO` directive and prior to an `ENDM` directive are part of the macro definition. Labels used within the macro must be local to the macro so the macro can be called repetitively.

Macro Directives (Assemblers)

Directives that control the execution and data allocation within macro body definitions.

Make Project (MPLAB IDEv5.xx/v6.xx)

A command that rebuilds an application by recompiling only those source files that have changed since the last complete compilation.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also μC.

Memory Models (Compilers)

(C17): Versions of libraries and/or precompiled object files based on a device's memory (RAM/ROM) size and structure.

(C18): A description that specifies the size of pointers that point to program memory.

Microcontroller

A highly integrated chip that contains all the components comprising a controller. Typically this includes a CPU, RAM, some form of ROM, I/O ports and timers. Unlike a general-purpose computer, which also includes all of these components, a microcontroller is designed for a very specific task – to control a particular system. As a result, the parts can be simplified and reduced, which cuts down on production costs.

Microcontroller Mode – PIC17 and PIC18 Devices Only

One of the possible program memory configurations of the PIC17 and PIC18 families of microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode – PIC17 and PIC18 Devices Only

One of the possible program memory configurations of the PIC17 and PIC18 families of microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

Instructions that are translated directly into machine code. Mnemonics are used to perform arithmetic and logical operations on data residing in program or data memory of a microcontroller. They can also move data in and out of registers and memory as well as change the flow of program execution. Also referred to as Opcodes.

© 2002 Microchip Technology Inc.

MPASM Assembler

Microchip Technology's relocatable macro assembler. MPASM assembler is a command-line or Windows-based PC application that provides a platform for developing assembly language code for Microchip's PICmicro microcontroller (MCU) families, KEELOQ® devices and Microchip memory devices. Generically, MPASM assembler will refer to the entire development platform including the macro assembler and utility functions.

MPASM assembler will translate source code into either object or executable code. The object code created by the assembler may be turned into executable code through the use of the MPLINK linker.

MPLAB C1X

Refers to both the MPLAB C17 and MPLAB C18 C compilers from Microchip. MPLAB C17 is the C compiler for PIC17 devices and MPLAB C18 is the C compiler for PIC18 and PIC18FXXXX devices.

MPLAB ICD and MPLAB ICD 2

Microchip's in-circuit debuggers, for PIC16F87X and PIC18FXXX devices, respectively. The ICDs work with MPLAB IDE. The main component of each ICD is the module. A complete system consists of a module, header, demo board, cables and MPLAB IDE Software.

MPLAB ICE 2000

Microchip's in-circuit emulator that works with MPLAB IDE.

MPLAB IDE

The name of the main executable program that supports the IDE.

(IDE5): MPLAB IDE v5.xx has a built-in project manager, editor and simulator (MPLAB SIM) and support for an emulator or debugger. The MPLAB IDE software resides on the PC host. The executable (`mplab.exe`) calls many other files. MPLAB IDE v5.xx and lower is a 16-bit application.

(IDE6): MPLAB IDE v6.xx has a built-in project manager, editor and support for debug and programming tools. The MPLAB IDE software resides on the PC host. The executable calls many other files. MPLAB IDE v6.xx and higher is a 32-bit application.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE in support of PICmicro MCU devices.

MPLIB Object Librarian

The MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C1X C compilers.

The MPLIB librarian will combine multiple object files into one library file. Then the librarian can be used to manipulate the object files within the created library.

MPLINK Object Linker

The MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip MPLAB C17 or C18 C compilers. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though it does not have to be.

The MPLINK linker will combine object files and libraries to create a single executable file.

MPSIM Simulator

The DOS version of Microchip's MPLAB SIM simulator.

MRU

Most Recently Used. Refers to files and windows available to be selected from MPLAB IDE main pull down menus.

N

Nesting Depth

The maximum level to which macros can include other macros.

Node (MPLAB IDE v5.xx)

MPLAB IDE project component.

Non Real-Time

Refers to the processor at a breakpoint or executing single step instructions or MPLAB IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

No Operation. An instruction that has no effect when executed except to advance the program counter.

O

Object Code

The machine code generated by a source code language processor such as an assembler or compiler. A file of object code may be immediately executable or it may require linking with other object code files, e.g. libraries, to produce a complete executable program.

Object File

A module which may contain relocatable code or data and references to external code or data. Typically, multiple object modules are linked to form a single executable output. Special directives are required in the source code when generating an object file. The object file contains object code.

Object File Directives

Directives that are used only when creating an object file.

Off-Chip Memory – PIC17 and PIC18 Devices Only

Off-chip memory refers to the memory selection option for the PIC17 or PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the Emulator. The Memory tab accessed from *Options > Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. See Mnemonics.

Operators

Arithmetic symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence.

© 2002 Microchip Technology Inc.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter (MPLAB IDE v5.xx/v6.xx)

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or Program Counter.

PC Host

Any IBM® or compatible Personal Computer running Windows® 3.1x or Windows 95/98, Windows NT®, or Windows 2000.

PICmicro® MCUs

PICmicro® microcontrollers (MCUs) refers to all Microchip microcontroller families.

PICSTART Plus Programmer

A device programmer from Microchip. Programs 8, 14, 28 and 40 pin PICmicro® microcontrollers. Must be used with MPLAB® IDE Software.

Pod (MPLAB ICE 2000)

The external emulator box that contains emulation memory, trace memory, event and cycle timers and trace/breakpoint logic.

Power-on-Reset Emulation (MPLAB ICE 2000)

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

Pragma (Compilers)

A directive that has meaning to a specific compiler.

Precedence

The concept that some elements of an expression are evaluated before others; (i.e., * and / before + and -). In language tools, operators of the same precedence are evaluated from left to right. Use parentheses to alter the order of evaluation.

Program Counter

A register that specifies the current execution address for emulation and simulation.

Program Memory

The memory area in a microcontroller where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

Programmer

A device used to program electrically programmable semiconductor devices such as microcontrollers.

Project (MPLAB IDE v5.xx/v6.xx)

A set of source files and instructions to build the object and executable code for an application.

PRO MATE II Programmer

A device programmer from Microchip. Programs all PICmicro microcontrollers and most memory and Keeloq devices. Can be used with MPLAB IDE or stand-alone.

Prototype System

A term referring to a user's target application, or target board.

PWM Signals

Pulse Width Modulation Signals. Certain PICmicro MCU devices have a PWM peripheral.

Q

Qualifier

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

R

Radix

The number base, hexadecimal, or decimal, used in specifying an address and for entering data in the *Window > Modify* command.

RAM

Random Access Memory (Data Memory).

Raw Data

The binary representation of code or data associated with a section.

Real-Time

When released from the halt state in the emulator or MPLAB ICD mode, the processor runs in real-time mode and behaves exactly as the normal chip would behave. In real-time mode, the real-time trace buffer of MPLAB ICE is enabled and constantly captures all selected cycles, and all break logic is enabled. In the emulator or MPLAB ICD, the processor executes in real-time until a valid breakpoint causes a halt, or until the user halts the emulator.

In the simulator real-time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. The assembler currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

© 2002 Microchip Technology Inc.

Relocatable Section (Linkers)

A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation (Linkers)

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all identifier symbol definitions within the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory).

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

S

Section (Linkers)

An portion of code or data which has a name, size and address.

SFR

See Special Function Registers.

Shared Section (MPLINK Linker)

A section which resides in a shared (non-banked) region of data RAM.

Shell (MPASM Assembler)

The MPASM assembler shell is a prompted input interface to the macro assembler. There are two MPASM assembler shells: one for the DOS version and one for the Windows® version.

Simulator

A software program that models the operation of PICmicro microcontrollers.

Single Step (MPLAB IDE v5.xx/v6.xx)

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables and status displays so you can analyze and debug instruction execution.

You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew (MPLAB ICE 2000)

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed Opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the Opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2))

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code – Assembly

Source code consists of PICmicro MCU instructions and MPASM assembler directives and macros that will be translated into machine code by an assembler.

Source Code – C

A program written in the high level language called "C" which will be converted into PICmicro MCU machine code by a compiler. Machine code is suitable for use by a PICmicro MCU or Microchip development system product like MPLAB IDE.

Source File – Assembly

The ASCII text file of PICmicro MCU instructions and MPASM assembler directives and macros (source code) that will be translated into machine code by an assembler. It is an ASCII file that can be created using any ASCII text editor.

Source File – C

The ASCII text file containing C source code that will be translated into machine code by a compiler. It is an ASCII file that can be created using any ASCII text editor.

Special Function Registers

Registers that control I/O processor functions, I/O status, timers, or other modes or peripherals.

Stack – Hardware

An area in PICmicro MCU memory where function arguments, return values, local variables and return addresses are stored; (i.e., a "Push-Down" list of calling routines). Each time a PICmicro MCU executes a `CALL` or responds to an interrupt, the software pushes the return address to the stack. A return command pops the address from the stack and puts it in the program counter.

The PIC18 family also has a hardware stack to store register values for "fast" interrupts.

Stack – Software

The compiler uses a software stack for storing local variables and for passing arguments to and returning values from functions.

Static RAM or SRAM

Static Random Access Memory. Program memory you can Read/Write on the target board that does not need refreshing frequently.

Status Bar (MPLAB IDE v5.xx/v6.xx)

The Status Bar is located on the bottom of the MPLAB IDE window and indicates such current information as cursor position, development mode and device and active tool bar.

Step Into (MPLAB IDE v5.xx/v6.xx)

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a CALL instruction into a subroutine.

© 2002 Microchip Technology Inc.

Step Over (MPLAB IDE v5.xx/v6.xx)

Step Over allows you to debug code without stepping into subroutines. When stepping over a CALL instruction, the next breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached.

The Step Over command is the same as Single Step except for its handling of CALL instructions.

Stimulus (Simulators)

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch (Simulators)

A counter for measuring execution cycles.

Symbol (MPLAB IDE v5.xx/v6.xx)

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc.

Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels.

System Button

The system button is another name for the system window control. Clicking on the system button pops up the system menu.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.” In some MPLAB IDE windows, additional modes or functions can be found.

T

Target (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

Refers to user hardware.

Target Application (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

Firmware residing on the target board.

Target Board (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

The circuitry and programmable device that makes up the target application.

Target Processor (MPLAB ICE 2000, MPLAB ICD, MPLAB ICD 2)

The microcontroller device on the target application board.

Template (Editor)

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar (MPLAB IDE v5.xx/v6.xx)

A row or column of icons that you can click on to execute MPLAB IDE functions.

Trace (Debuggers)

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE's trace window.

Trace Memory (Debuggers)

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trigger Output (MPLAB ICE 2000)

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs (Compilers)

These are three-character sequences, all starting with `??`, that are defined by ISO C to stand for single characters

The nine trigraphs and their replacements are

Trigraph:	??(	??)	??<	??>	??=	??/	??'	??!	??-
Replacement:	[	]	{	}	#	\	^	|	~

U

Unassigned Section (MPLINK Linker)

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

Data which is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

Upload

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

W

Warning

An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

Watchdog Timer (WDT)

A timer on a PICmicro microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using configuration bits.

Watch Variable (MPLAB IDE v5.xx/v6.xx)

A variable that you may monitor during a debugging session in a watch window.

Watch Window (MPLAB IDE v5.xx/v6.xx)

Watch windows contain a list of watch variables that are updated at each breakpoint.

© 2002 Microchip Technology Inc.

Index

Symbols

#include 28, 82
#pragma interrupt 28
#pragma list / nolist 29
#pragma sectiontype 29, 65
#pragma varlocate 30, 94
.asm 39, 57
.c 22, 39, 57
.cod 39, 57
.err 23, 60
.h 60
.hex 39, 42, 57, 61
.lib 39, 57
.lkr 39, 57
.lst 39, 57
.map 39, 57, 61
.o 23, 39, 57
.out 39, 57, 61

A

ACSII Character Set 153
Add Project Files 44
Address Spaces, ROM and RAM 34, 65
ALUSTA 73, 87
ANSI C vs. MPLAB C17 25
ANSI Compatibility 135
ANSI-89 extension 78
Architecture 20
Arrays
    ANSI C 138
    Initialization 33
asm (_asm) 27, 89
Assembler, Internal 89
Assembly Language, Mixing with C 90
Assembly, Inline 91
Attributes
    Overlay 66, 70
    Shared 66, 71
AUTOEXEC.BAT 24

B

Banked/Paged Data 69
bin directory 24, 59
Bit-Fields 35
    ANSI C 139
BSR 73, 87

C

C Programming References 149
C, Mixing with Assembly Language 90
c0l17.asm 68
c0s17.asm 67
Call Conventions
    Functions 72
    Mixing C and ASM 90
    Software Stack 72
char 78
Characters, ANSI C 136
ClrWdt( ) 79
code 65, 70
Code and Data Sections 65
Code File 39, 57
Code, Locating 70
Code, Start Up 39, 47, 57, 60, 67
COFF File 39, 57, 61
Command Line
    Multiple File Compile 61
    Option Descriptions 58
    Overview 55
    Single File Compile 59
Comments 89
Compiler Versions 23
Constants 31
Contents, Section 66
Controlling What Goes on the Stack 72
Customer Support 10

D

Data in Program Memory 71
Data Representation 77
Data Type 77
    Floating Point 78
    Integer 78
Data, Locating 70
Development Mode 41
doc directory 24
Documentation
    Conventions 7
    Layout 5
    Numbering Conventions 8
    Updates 7

E

Edit Project 42
Enable/Disable Interrupts 87, 97
endasm (_endasm) 27, 89
endian 77
Enumerations, ANSI C 139
Environment Variable *See MCC_INCLUDE*
Error File 23, 60
Errors 141
Example Code 151
examples directory 24
Executable
    Directory 24, 59
    Files 23, 37, 39, 57, 61
External Declaration 79, 80

F

far 27, 69, 82
Floating Point 78
    ANSI C 137
FSRx 23, 75
Function Call Conventions 72
Functions 26, 32

G

Glossary 155
Going Forward 53, 62

H

h directory 24, 42, 59
Header
    Directory 24, 42, 59
    File 60
Hex File 39, 42, 57, 61

I

idata 65, 67, 70
Identifiers, ANSI C 135
IEEE 754 78
Include
    Current Search Path 28, 42
    Directory 59
Initialization 67
    Arrays 33
    Data 39, 47, 57, 60, 68
    Stack 69
Inline Assembly Language 91
Install _TMR0 74
Install Language Tool 50
Install/Uninstall the Compiler 24
Install_INT 80
Install_PIV 80
Install_T0CKI 80
Install_TMR0 80
Installation Requirements 23
int 78
Integers 78
    ANSI C 136
Internet Address 9
Interrupts 83
    Disable 121
    Enable 97
    Enable/Disable 87
    Handler Code 39, 47, 57, 60
    ISR, Writing 84
    Latency 86
    Nested 73, 87
    Saving FSR 75
    Support Macros 73, 86
    Vector, Writing 85
ISR *See Interrupts*

K

Keyword Differences 27

L

Latency, Interrupt 86
lib directory 24, 47, 59
Librarian *See MPLIB Librarian*
Libraries 8, 19, 53, 62, 93
Library
    Directory 24, 47, 59
    Files 39, 57
Linker
    Directory 24, 59
    Script 39, 48, 57
Linker Script 82, 94
Linker *See MPLINK Linker*
list 29
Listing File 39, 57
little endian 77
lkr directory 24, 59
Locating Code 70
Locating Data 70
long 78

M

main( ), branching to 69
Make Project 50
Map File 39, 57, 61
MCC_INCLUDE 24, 28, 59
mcc17 ..... 23, 37, 58
mcc17d 23, 37
Memory
    Models 47, 69
    Requirements 23
Microchip Internet Web Site 9
MPLAB C17 Description 19, 164
MPLAB C17 Libraries 8
MPLAB C17 vs. ANSI C 25
MPLAB ICE 19
MPLAB IDE 19, 24, 40

© 2002 Microchip Technology Inc.

MPLAB Projects 37
MPLAB SIM 19
MPLIB Librarian 21, 39, 57
MPLINK Linker 21, 39, 43, 51, 57, 89

N

near 27, 69
Nesting Interrupts 73, 87
New Project 41
Node Properties 43
Nop( ) 79

O

Object Files 23
Object Files, Precompiled 39, 47, 57
Oddities of Standard Functions 35
Operators 33
Optimization Tips 93
Overlay 70
overlay 66

P

Paged/Banked Data 69
PCLATH 73, 87
PICmicro MCU 170
Pointers 35
    ANSI C 138
pragma interrupt 28
pragma list / nolist 29
pragma sectiontype 29, 65
pragma *See also #pragma*
pragma varlocate 30, 94
Precompiled Object Files 47
Preprocessor Directives, ANSI C 139
Processor Header File 26, 79, 82
PRODH 74, 75
PRODH, PRODL 23
PRODL 74, 75
Product Support 11
Program Components, Basic 26
Program Memory, Data in 71
Project Nodes 44
Project Window 52
Projects, MPLAB IDE 37

R

RAM
    Address Spaces 34
    Pointers 34, 35
    ram 27, 33, 69
Ranges, Integer Types 78
README File 8
References 8
Register Definitions 39, 47, 57, 60, 80
Registers, ANSI C 138
Reserved Resources 23
Reset( ) 79
Rlcf( ) 79
Rlncf( ) 79
ROM
    Address Spaces 34
    Pointers 34, 35
    rom 27, 33, 69
    String 34
romdata 65, 70
Rrcf( ) 79

S

Sections
    Allocation 65
    Attributes 66
    Code and Data 65
    Contents 66
    Default Names 67
SFRs 80
SFRs, Using 82
Shared 71
shared 66
short 78
signed char 78
signed int 78
signed long 78
signed short 78
Simulator *See MPLAB SIM*
Sizes, Integer Types 78
Sleep( ) 79
Software Stack Call Conventions 72
Source Code 22, 39, 57
    Directory 24
Special Function Registers 80
src directory 24
Stack, Software 71
    Control What Goes On 72
    Initialization 69
    Size and Location 71
Standard Functions, Oddities 35
Start Up Code 39, 47, 57, 60, 67
STARTUP( ) (_ _STARTUP( )) 68
Statement Differences 28
Static Locals And Parameters 93
Static Strings 34
Storage Classes 32
Strings 35
Structures 35
    ANSI C 138
Support
    Customer 10
    Product 11
Swapf( ) 79
switch 33
    ANSI C 139

System Requirements, Host Computer 23

T

TBLPTR 75
TBLPTRH, TBLPTRL, TBLAT 23
Troubleshooting 9, 50

U

udata 65, 66, 70
Uninstall the Compiler 24
Unions, ANSI C 138
unsigned char 78
unsigned int 78
unsigned long 78
unsigned short 78
USE_INITDATA 68
USE_STARTUP 68
Using SFR's 82

V

Variables 31
Versions, Compiler 23
volatile 81, 82

W

Warnings 145
Watchdog Timer (WDT) 79
WREG 33, 73, 75, 84, 87
Writing an Interrupt Service Routine 84
Writing Efficient Code 93
Writing the Interrupt Vector 85
WWW Address 9

© 2002 Microchip Technology Inc.

NOTES:

NOTES:

© 2002 Microchip Technology Inc.

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com

Rocky Mountain
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966 Fax: 480-792-4338

Atlanta
3780 Mansell Road, Suite 130
Alpharetta, GA 30022
Tel: 770-640-0034 Fax: 770-640-0307

Boston
2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848 Fax: 978-692-3821

Chicago
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Kokomo
2767 S. Albright Road
Kokomo, Indiana 46902
Tel: 765-864-8360 Fax: 765-864-8387

Los Angeles
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

Toronto
6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699 Fax: 905-673-6509

ASIA/PACIFIC

Australia
Microchip Technology Australia Pty Ltd
Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733 Fax: 61-2-9868-6755

China - Beijing
Microchip Technology Consulting (Shanghai) Co., Ltd., Beijing Liaison Office
Unit 915
Bei Hai Wan Tai Bldg.
No. 6 Chaoyangmen Beidajie
Beijing, 100027, No. China
Tel: 86-10-85282100 Fax: 86-10-85282104

China - Chengdu
Microchip Technology Consulting (Shanghai) Co., Ltd., Chengdu Liaison Office
Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200 Fax: 86-28-86766599

China - Fuzhou
Microchip Technology Consulting (Shanghai) Co., Ltd., Fuzhou Liaison Office
Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506 Fax: 86-591-7503521

China - Hong Kong SAR
Microchip Technology Hongkong Ltd.
Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200 Fax: 852-2401-3431

China - Shanghai
Microchip Technology Consulting (Shanghai) Co., Ltd.
Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

China - Shenzhen
Microchip Technology Consulting (Shanghai) Co., Ltd., Shenzhen Liaison Office
Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380 Fax: 86-755-82966626

China - Qingdao
Rm. B503, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India
Microchip Technology Inc.
India Liaison Office
Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaugnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan
Microchip Technology Japan K.K.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan
Microchip Technology (Barbados) Inc.,
Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria
Microchip Technology Austria GmbH
Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark
Microchip Technology Nordic ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Microchip Technology GmbH
Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-039-65791-1 Fax: 39-039-6899883

United Kingdom
Microchip Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5869 Fax: 44-118 921-5820

12/05/02

© 2002 Microchip Technology Inc.