
MPLAB[®]-CXX COMPILER USER'S GUIDE

Information contained in this publication regarding device applications and the like is intended by way of suggestion only. 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. Use of Microchip's products as critical components in life support systems is not authorized except with express written approval by Microchip.

© 2000 Microchip Technology Incorporated. All rights reserved.

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

Microsoft, the Microsoft Internet Explorer, Windows, and MS-DOS are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Netscape is a registered trademark of Netscape Corporation in the United States and/or other countries.

Microwire is a registered trademark of National Semiconductor Corporation.

I²C is a trademark of Philips Corporation.

SPI is a registered trademark of Motorola Corporation

All product/company trademarks mentioned herein are the property of their respective companies.

MPLAB[®]-CXX Compiler User's Guide



Table of Contents

General Information

Introduction	1
Highlights	1
About This Guide	1
Warranty Registration	4
Recommended Reading	4
Troubleshooting	5
The Microchip Internet Web Site	6
Development Systems Customer Notification Service	7
Customer Support	9

MPLAB[®]-CXX Compiler User's Guide

Part 1 – Getting Started with MPLAB-CXX

Chapter 1. MPLAB-CXX Preview

1.1 Introduction	13
1.2 Highlights	13
1.3 What MPLAB-CXX Is	13
1.4 What MPLAB-CXX Does	14
1.5 ANSI Compatibility	14
1.6 Tool Compatibility	14

Chapter 2. MPLAB-CXX Installation

2.1 Introduction	15
2.2 Highlights	15
2.3 Host Computer System Requirements	15
2.4 Compiler Versions	15
2.5 Installation	16

Chapter 3. MPLAB-CXX Overview

3.1 Introduction	19
3.2 Highlights	19
3.3 Overview of Compilers	19
3.4 Compiler Input/Output Files	21
3.5 Compiler Resource Requirements	22

Chapter 4. Using MPLAB-CXX without MPLAB IDE

4.1 Introduction	23
4.2 Highlights	23
4.3 MPLAB-CXX – Command Line Overview	23
4.4 Using MPLAB-C17 on the Command Line	26
4.5 Using MPLAB-C18 on the Command Line	31
4.6 Going Forward	36

Table Of Contents

Chapter 5. Using MPLAB-CXX with MPLAB IDE

5.1 Introduction	37
5.2 Highlights	37
5.3 MPLAB-CXX – MPLAB Projects Overview	37
5.4 Using MPLAB-C17 with MPLAB IDE – A Tutorial	40
5.5 Using MPLAB-C18 with MPLAB IDE – A Tutorial	54
5.6 Going Forward	68

MPLAB[®]-CXX Compiler User's Guide

Part 2 – Using MPLAB-CXX

Chapter 6. MPLAB-CXX and C

6.1 Introduction	71
6.2 Highlights	71
6.3 C vs. MPLAB-CXX	71
6.4 Components of a Basic MPLAB-CXX Program	72
6.5 C Keywords	73

Chapter 7. MPLAB-CXX Fundamentals

7.1 Introduction	75
7.2 Highlights	75
7.3 Preprocessor Directives	76
7.4 Comments	85
7.5 Constants	86
7.6 Variables	88
7.7 Functions	96
7.8 Operators	99
7.9 Program Control Statements	106
7.10 Arrays and Strings	112
7.11 Pointers	115
7.12 Structures and Unions	118

Chapter 8. MPLAB-CXX and PICmicro MCU Programming

8.1 Introduction	123
8.2 Highlights	123
8.3 PICmicro MCU Programming Specifics	123
8.4 MPLAB-C17 and PICmicro MCU Programming	126
8.5 MPLAB-C18 and PICmicro MCU Programming	135

Table Of Contents

Chapter 9. Mixing Assembly Language and C Modules

9.1 Introduction	139
9.2 Highlights	139
9.3 Calling Conventions	139
9.4 Mixing Assembly Language and C Variables and Functions ...	140
9.5 Calling an Assembly Function in C – MPLAB-C17	141
9.6 Using the File Selection Registers (FSR's)	142

Chapter 10. ANSI Implementation Issues

10.1 Introduction	145
10.2 Highlights	145
10.3 Identifiers	145
10.4 Characters	146
10.5 Integers	146
10.6 Floating Point	147
10.7 Arrays and Pointers	148
10.8 Registers	148
10.9 Structures and Unions	148
10.10 Bit-Fields	149
10.11 Enumerations	149
10.12 Switch Statement	149
10.13 Preprocessing Directives	149

Chapter 11. Examples

11.1 Introduction	151
11.2 Highlights	151
11.3 Overview of Example Files	151
11.4 Example Details	152

MPLAB[®]-CXX Compiler User's Guide

Appendices

Appendix A. ASCII Character Set

A.1 Introduction	155
A.2 ASCII Character Set	155

Appendix B. PIC17CXXX Instruction Set

B.1 Introduction	157
B.2 Highlights	157
B.3 Key to PICmicro MCU Family Instruction Sets	157
B.4 PIC17CXXX Instruction Set	158

Appendix C. PIC18CXXX Instruction Set

C.1 Introduction	163
C.2 Highlights	163
C.3 Key to Enhanced 16-Bit Core Instruction Set	163
C.4 PIC18CXXX Instruction Set	164

Appendix D. MPLAB-C17 Errors

D.1 Introduction	169
D.2 Highlights	169
D.3 Errors	169
D.4 Warnings	173

Appendix E. MPLAB-C18 Errors

E.1 Introduction	175
E.2 Highlights	175
E.3 Errors	175
E.4 Warnings	182

Appendix F. References

F.1 Introduction	185
F.2 Highlights	185
F.3 C Standards Information	185
F.4 General C Information	185

Table Of Contents

Glossary 187

- Introduction 187
- Highlights 187
- Terms 187

Index 203

Worldwide Sales and Service 210

MPLAB[®]-CXX Compiler User's Guide

General Information

Introduction

This first chapter contains general information that will be useful to know before using MPLAB-C17 or MPLAB-C18.

Highlights

The information you will garner from this chapter:

- About this Guide
- Recommended Reading
- Warranty Registration
- Troubleshooting
- The Microchip Internet Web Site
- Development Systems Customer Notification Service
- Customer Support

About This Guide

Document Layout

This document describes how to use MPLAB-CXX 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).

The User's Guide layout is as follows:

Part 1 - Getting Started with MPLAB-CXX

- **Chapter 1: MPLAB-CXX Preview** – describes what MPLAB-C17 and MPLAB-C18 are and what they can do.
- **Chapter 2: MPLAB-CXX Installation** – discusses compiler versions, PC requirements, and installation procedures.
- **Chapter 3: MPLAB-CXX Overview** – gives an overview of compiler operation, input/output files, and resource requirements.
- **Chapter 4: Using MPLAB-CXX without MPLAB IDE** – describes how to use MPLAB-C17 or MPLAB-C18 as stand-alone compilers.
- **Chapter 5: Using MPLAB-CXX with MPLAB IDE** – describes how to use MPLAB-C17 or MPLAB-C18 with MPLAB IDE.

MPLAB[®]-CXX Compiler User's Guide

Part 2 - Using MPLAB-CXX

- **Chapter 6: MPLAB-CXX and C** – compares MPLAB-CXX and C.
- **Chapter 7: MPLAB-CXX Fundamentals** – describes the MPLAB-CXX programming language including functions, statements, operators, variables, and other elements.
- **Chapter 8: MPLAB-CXX and PICmicro[®] MCU Programming** – describes how to use MPLAB-CXX in conjunction with device programming.
- **Chapter 9: Mixing C with Assembly Language Modules** – provides guidelines to using C with MPASM assembly language modules.
- **Chapter 10: ANSI Implementation Issues** – details MPLAB-CXX specific parameters described as implementation defined in the ANSI standard.
- **Chapter 11: Examples** – discusses the MPLAB-C17 examples included in the `examples` directory.

Appendices

- **Appendix A: ASCII Character Set** – contains the ASCII character set.
- **Appendix B: PIC17CXXX Instruction Set** – gives the instruction set for the PIC17CXXX device family.
- **Appendix C: PIC18CXXX Instruction Set** – gives the instruction set for the PIC18CXXX device family.
- **Appendix D: MPLAB-C17 Errors** – lists errors generated by MPLAB-C17.
- **Appendix E: MPLAB-C18 Errors** – lists errors generated by MPLAB-C18.
- **Appendix F: References** – gives references that may be helpful in programming with MPLAB-CXX.

- **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 number for Microchip Technology Inc. sales and service locations throughout the world.

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 (Helvetica font):		
Underlined, italic text with right arrow	A menu selection from the menu bar	<u>File</u> > <i>Save</i>
Bold characters	A window or dialog button to click	OK, Cancel
Characters in angle brackets < >	A key on the keyboard	<Tab>, <Ctrl-C>
Documents (Helvetica font):		
Italic characters	Referenced books	<i>MPLAB User's Guide</i>

Updates

All documentation becomes dated, and this user's guide is no exception. Since MPLAB IDE, MPLAB-C17, MPLAB-C18 and other Microchip tools are constantly evolving to meet customer needs, some MPLAB IDE dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site to obtain the latest documentation available.

MPLAB[®]-CXX Compiler User's Guide

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 and MPLAB-C18. 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.

MPLAB-CXX Reference Guide – Libraries and Precompiled Object Files (DS51224)

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

README.C17, README.C18

For the latest information on using MPLAB-C17 or MPLAB-C18, read the README.C17 or README.C18 file (ASCII text) included with the software. These README files contain update information that may not be included in this document.

README.XXX

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

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 assembler (MPASM), the linker (MPLINK) and the librarian (MPLIB).

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

Our web site (<http://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.

MPLAB[®]-CXX Compiler User's Guide

The Microchip Internet Web Site

Microchip provides on-line support on the Microchip World Wide Web (WWW) site.

The web site is used by Microchip as a means to make files and information easily available to customers. To view the site, the user must have access to the Internet and a web browser, such as Netscape[®] Communicator or Microsoft[®] Internet Explorer[®]. Files are also available for FTP download from our FTP site.

Connecting to the Microchip Internet Web Site

The Microchip web site is available by using your favorite Internet browser to attach to:

<http://www.microchip.com>

The file transfer site is available by using an FTP program/client to connect 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, distributors and factory representatives. Other data available for consideration is:

- Latest Microchip Press Releases
- Technical Support Section with Frequently Asked Questions
- 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

Development Systems Customer Notification Service

Microchip provides a customer notification service to help our customers keep current on Microchip products with the least amount of effort. Once you subscribe to one of our list servers, you will receive email notification whenever we change, update, revise or have errata related to that product family or development tool. See the Microchip WWW page for other Microchip list servers.

The Development Systems list names are:

- Compilers
- Emulators
- Programmers
- MPLAB IDE
- Otools (Other Tools)

Once you have determined the names of the lists that you are interested in, you can subscribe by sending a message to:

```
listserv@mail.microchip.com
```

with the following as the body:

```
subscribe <listname> yourname
```

Here is an example:

```
subscribe mplab John Doe
```

To UNSUBSCRIBE from these lists, send a message to:

```
listserv@mail.microchip.com
```

with the following as the body:

```
unsubscribe <listname> yourname
```

Here is an example:

```
unsubscribe mplab John Doe
```

The following sections provide descriptions of the available Development Systems lists.

Compilers

The latest information on Microchip C compilers, Linkers and Assemblers. These include MPLAB-C17, MPLAB-C18, MPLINK, MPASM as well as the Librarian, MPLIB for MPLINK.

To SUBSCRIBE to this list, send a message to:

```
listserv@mail.microchip.com
```

with the following as the body:

```
subscribe compilers yourname
```

MPLAB[®]-CXX Compiler User's Guide

Emulators

The latest information on Microchip In-Circuit Emulators. These include MPLAB-ICE and PICMASTER.

To SUBSCRIBE to this list, send a message to:

`listserv@mail.microchip.com`

with the following as the body:

`subscribe emulators yourname`

Programmers

The latest information on Microchip PICmicro MCU device programmers. These include PRO MATE[®] II and PICSTART[®] Plus.

To SUBSCRIBE to this list, send a message to:

`listserv@mail.microchip.com`

with the following as the body:

`subscribe programmers yourname`

MPLAB IDE

The latest information on Microchip MPLAB IDE, the Windows Integrated Development Environment for development systems tools. This list is focused on MPLAB IDE, MPLAB-SIM, MPLAB Project Manager and general editing and debugging features. For specific information on MPLAB IDE compilers, linkers and assemblers, subscribe to the COMPILERS list. For specific information on MPLAB IDE emulators, subscribe to the EMULATORS list. For specific information on MPLAB IDE device programmers, please subscribe to the PROGRAMMERS list.

To SUBSCRIBE to this list, send a message to:

`listserv@mail.microchip.com`

with the following as the body:

`subscribe mplab yourname`

Otools (Other Tools)

The latest information on other development system tools provided by Microchip. For specific information on MPLAB IDE and its integrated tools refer to the other mail lists.

To SUBSCRIBE to this list, send a message to:

`listserv@mail.microchip.com`

with the following as the body:

`subscribe otools yourname`

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Corporate Applications Engineer (CAE)
- Hot line

Customers should call their distributor, representative, or field application engineer (FAE) for support. Local sales offices are also available to help customers. See the back cover for a listing of sales offices and locations.

Corporate applications engineers (CAEs) may be contacted at (480) 786-7627.

In addition, there is a Systems Information and Upgrade Line. This line provides system users 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 any currently available upgrade kits.

The Hot Line Numbers are:

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

1-480-786-7302 for the rest of the world.

MPLAB[®]-CXX Compiler User's Guide

NOTES:



MPLAB[®]-CXX COMPILER USER'S GUIDE

Part 1 – Getting Started with MPLAB-CXX

Chapter 1. MPLAB-CXX Preview	13
Chapter 2. MPLAB-CXX Installation.....	15
Chapter 3. MPLAB-CXX Overview.....	19
Chapter 4. Using MPLAB-CXX without MPLAB IDE	23
Chapter 5. Using MPLAB-CXX with MPLAB IDE.....	37

Part
1

Getting started
with MPLAB-CXX

MPLAB[®]-CXX Compiler User's Guide

Chapter 1. MPLAB-CXX Preview

1.1 Introduction

This chapter will give you a preview of MPLAB-C17 and MPLAB-C18.

1.2 Highlights

This chapter covers the following topics:

- What MPLAB-CXX Is
- What MPLAB-CXX Does
- ANSI Compatibility
- Tool Compatibility

1.3 What MPLAB-CXX Is

MPLAB-CXX is a term used to refer to both the MPLAB-C17 and MPLAB-C18 compilers. The MPLAB-C17 and MPLAB-C18 compilers are full-featured ANSI C compilers for the Microchip Technology PIC17CXXX and PIC18CXXX PICmicro 8-bit microcontrollers, respectively. The compilers are fully compatible with Microchip's MPLAB Integrated Development Environment (IDE), 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 and MPLAB-C18 have implemented extensions to the C language to provide specific support for Microchip's PICmicro MCU peripherals. The C libraries include: A/D converter, Input Capture, Interrupt Support Macros, SPI[®], Timers, USART, I²C[®], I/O Port, Pulse Width Modulation, Reset, External LCD, Software SPI, Software I²C, Software USART, Character Classification, Relay, Memory/String Manipulation, and Number/Text Conversion.

MPLAB[®]-CXX Compiler User's Guide

1.4 What MPLAB-CXX Does

The MPLAB-C17 and MPLAB-C18 compilers provide a solution for developing C code for the PIC17CXXX and PIC18CXXX microcontrollers. These compilers have the following features:

- ANSI compatibility
- Integration with MPLAB IDE for easy-to-use project management and source-level debugging
- Generation of relocatable object modules for enhanced code reuse
- Compatibility with object modules generated with MPASM, allowing complete freedom in mixing assembly and C in a single project
- Transparent read/write access to external memory
- Strong support for inline assembly when total control is absolutely necessary
- Efficient code generator engine with multi-level optimization
- Extensive library support, including PWM, SPI, I²C, USART, UART, string manipulation, and math libraries
- Full user-level control over data and code memory allocation

1.5 ANSI Compatibility

MPLAB-C17 and MPLAB-C18 are free-standing ANSI C implementations except where specifically noted in this user's guide. These compilers deviate from the ANSI standard only where the standard conflicts with efficient PICmicro MCU support. See Chapter 10 and the `README` files for details.

1.6 Tool Compatibility

MPLAB-C17 and MPLAB-C18 are compatible with all Microchip development systems currently in production. This includes MPLAB-SIM (PICmicro MCU discrete-event simulator), MPLAB-ICE (PICmicro MCU Universal In-Circuit Emulator), PRO MATE II (the Microchip Universal Programmer), and PICSTART Plus (the Microchip low-cost development programmer).

Chapter 2. MPLAB-CXX Installation

2.1 Introduction

This chapter discusses how to install MPLAB-C17 or MPLAB-C18 on your PC.

If you are going to use MPLAB-CXX with MPLAB, install MPLAB IDE first.

2.2 Highlights

This chapter includes:

- Host Computer System Requirements
- Compiler Versions
- Installation

2.3 Host Computer System Requirements

MPLAB-C17 requires:

- PC compatible 386 or better class system
- MS-DOS[®]/PC-DOS version 5.0 or greater, or Windows[®] 3.x, or Windows 95/98, or Windows NT[®]
- 16 MB memory (32 MB recommended)

MPLAB-C18 requires:

- PC compatible 486 or better class system
- Windows 95/98 or Windows NT
- 16 MB memory (32 MB recommended)

2.4 Compiler Versions

There are two versions of MPLAB-C17:

- a DOS-extended version, `mcc17d.exe`
- a Windows 32-bit version (Windows 9x/NT native), `mcc17.exe`

and one version of MPLAB-C18:

- a Windows 32-bit version (Windows 9x/NT native), `mcc18.exe`

All compilers are command line programs. `mcc17d.exe` is used with DOS or Windows 3.x. `mcc17.exe` and `mcc18.exe` are used with Windows 95/98 or Windows NT. You can use all versions with MPLAB IDE, though the Windows 9x/Windows NT native versions are recommended.

MPLAB[®]-CXX Compiler User's Guide

2.5 Installation

If you are going to use MPLAB-CXX with MPLAB, install MPLAB IDE first.

To install MPLAB-CXX, enter Windows, run the file `SETUP . EXE` on the CD-ROM, and follow the prompts. For illustration, the setup for MPLAB-C17 v2.30 is described here.

The first dialog you should see is the Welcome dialog. This dialog displays the version of MPLAB-C17 it will install and give you the opportunity to close any open programs before installing. When you are ready to proceed, click **Next**.

The next dialog is a display of the latest `README . C17`. Check out any new features and information here. When you are ready to proceed, click **Next**.

The Choose Destination Location dialog will appear next. The default destination directory for installing the files is `c : \mcc`, where `c :` is your master hard drive. If you wish to install in a different directory, click **Browse...** and find or make another directory for installation. Click **OK** to return to the Choose Destination Location dialog. Then click **Next** to continue.

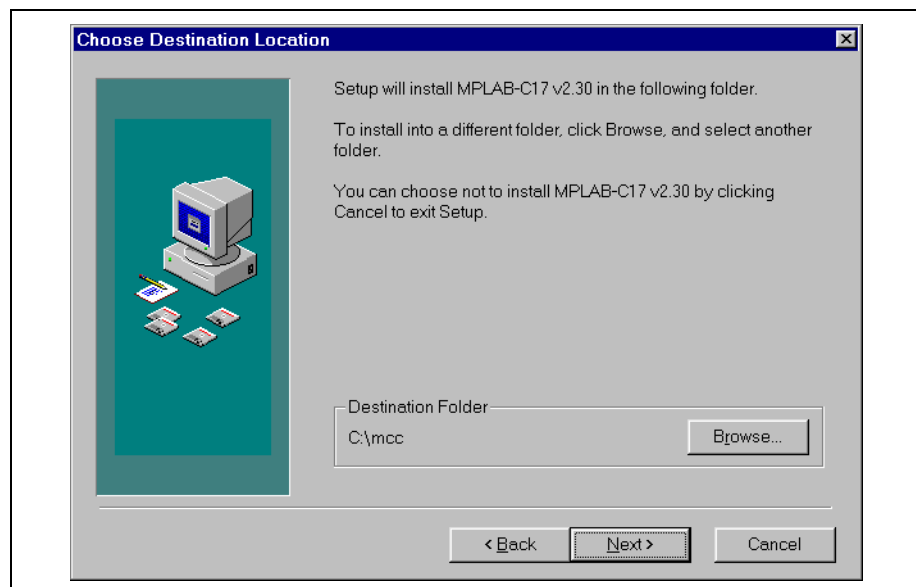


Figure 2.1: Choose Destination Location

The next dialog is Select Components. It is recommended that you install all components. Not doing so may cause certain MPLAB-C17 items not to function properly. Once you are more familiar with the compiler, you may go back and re-install it with fewer components, but for people new to MPLAB-C17, it is best to have all components installed. Click **Next** to proceed.

MPLAB-CXX Installation

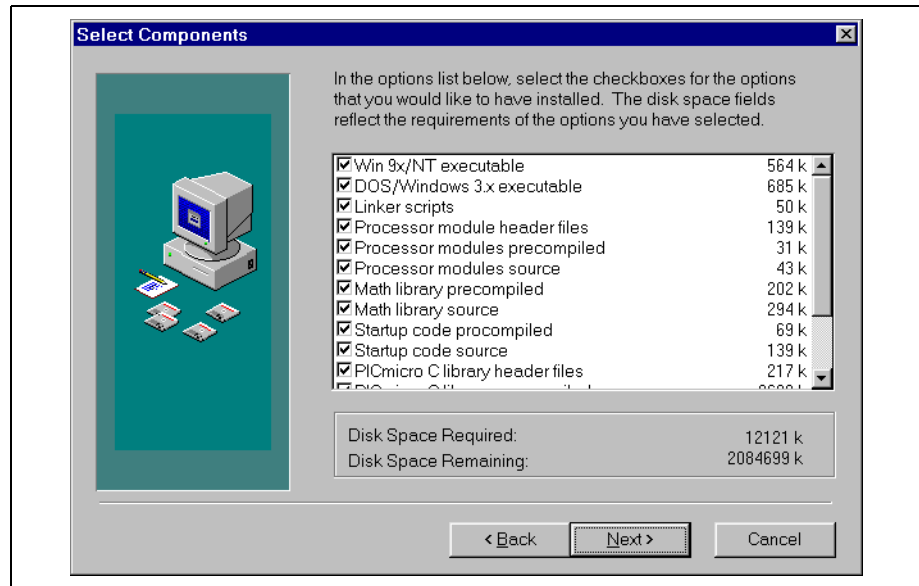


Figure 2.2: Select Components

The Select Program Manager dialog should now be visible. Choose the default group of Microchip MPLAB-C17 or pick another group from the list. Click **Next** to continue.

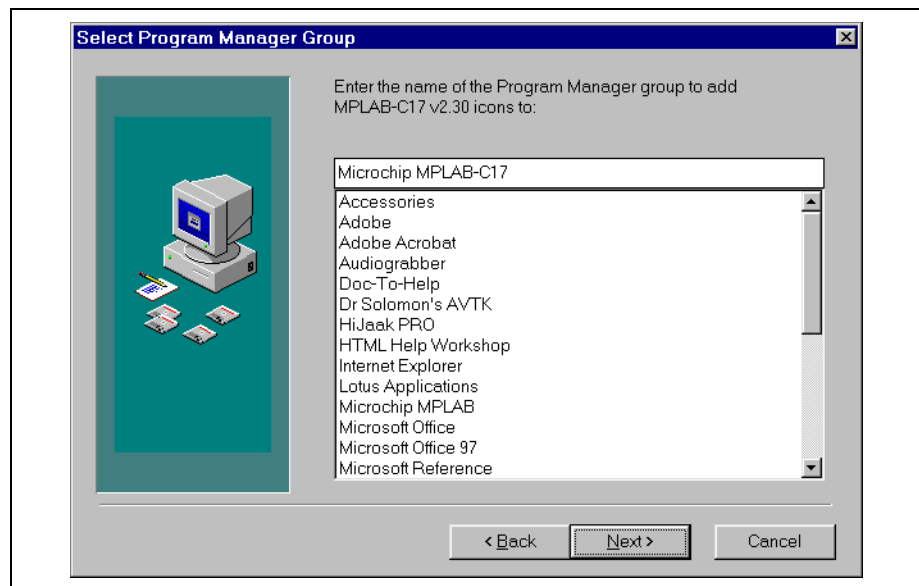


Figure 2.3: Select Program Manager Group

The next dialog is the Start Installation dialog. To complete the installation, click **Next**. To quit the installation at this point, click **Cancel**.

MPLAB[®]-CXX Compiler User's Guide

The install program will use or create the directory you chose in the Choose Destination Location dialog. Then it will install 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, 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.

Chapter 3. MPLAB-CXX Overview

3.1 Introduction

This chapter presents an overview of the compiler operation.

3.2 Highlights

This chapter includes:

- Overview of Compilers
- Compiler Input/Output Files
- Compiler Resource Requirements

3.3 Overview of Compilers

MPLAB-C17 and MPLAB-C18 generate 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 3.1, or it can be linked with other separately assembled or compiled modules as shown in Figure 3.2. Related modules can also be grouped and stored together in a library using Microchip's MPLIB Librarian (Figure 3.3). Required libraries can be specified at link time, and only the modules which are needed will be included in the final executable.

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

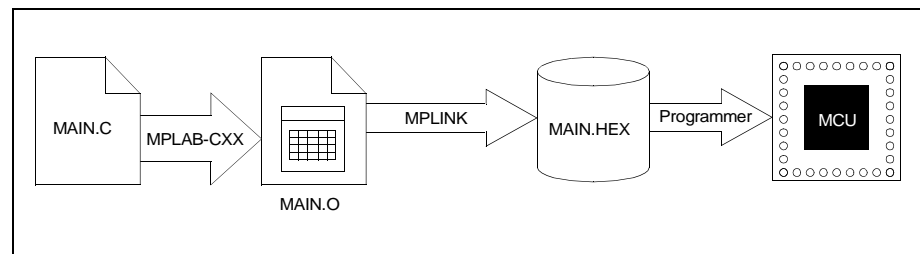


Figure 3.1: Generating Executable Code From One Object Module

MPLAB[®]-CXX Compiler User's Guide

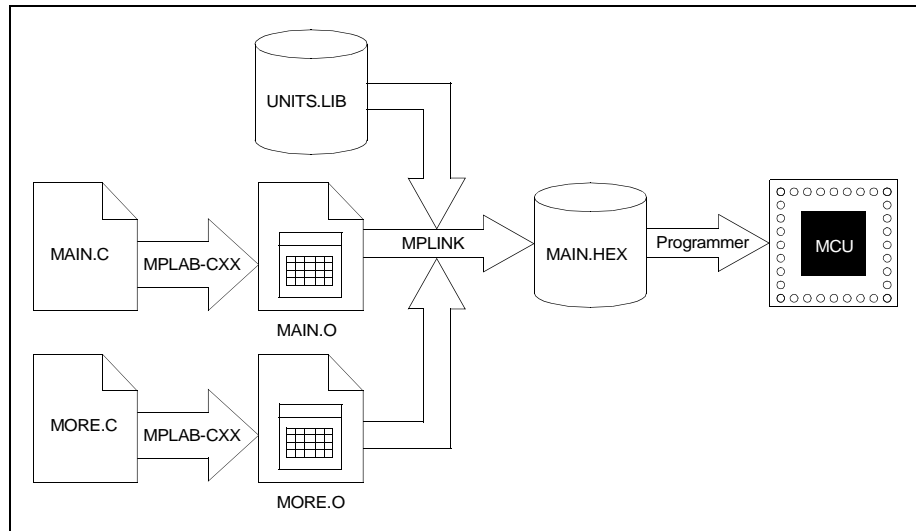


Figure 3.2: Generating Executable Code From Object Modules

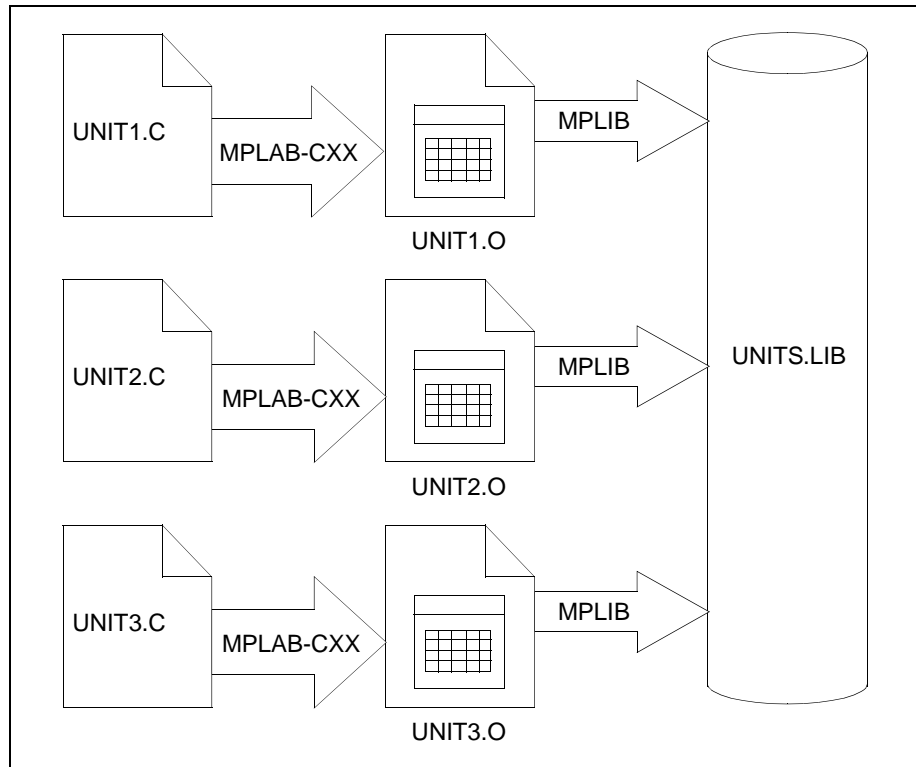


Figure 3.3: Creating a Reusable Object Library

3.4 Compiler Input/Output Files

These are the default file extensions used by MPLAB-CXX.

Table 3.1: MPLAB-CXX Default Extensions

Extension	Purpose
.c	Default source file extension input to MPLAB-C17/C18: <source_name>.c
.err	Output extension from MPLAB-C17/C18 for error files: <source_name>.err
.o	Output extension from MPLAB-C17/C18 for object files: <source_name>.o

3.4.1 Source Code Format (.c)

The source code file may 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 F.

3.4.2 Error File Format (.err)

MPLAB-CXX by default generates an error file. This file can be useful when debugging your code. The MPLAB Source Level Debugger will automatically open 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.

3.4.3 Object File Format (.o)

Object files are the relocatable code produced from source files.

3.5 Compiler Resource Requirements

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, FSR2**: 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.

Chapter 4. Using MPLAB-CXX without MPLAB IDE

4.1 Introduction

This chapter discusses how to use MPLAB-CXX from the command line (without MPLAB IDE).

4.2 Highlights

This chapter includes:

- MPLAB-CXX – Command Line Overview
- Using MPLAB-C17 on the Command Line
- Using MPLAB-C18 on the Command Line
- Going Forward

4.3 MPLAB-CXX – 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 Windows NT (`mcc17.exe`). To use MPLAB-C17 with MPLAB IDE, see Chapter 5.

MPLAB-C18 (`mcc18.exe`) can be invoked directly on the command line in console mode in Windows 95/98 or Windows NT. To use MPLAB-C18 with MPLAB IDE, see the next chapter.

MPLAB-CXX may be used alone to compile individual C source files into object files. Or, it may be used in conjunction with MPLINK to create hex files.

Figure 4.1 shows a generic use of the MPLAB-CXX compiler tool.

MPLAB[®]-CXX Compiler User's Guide

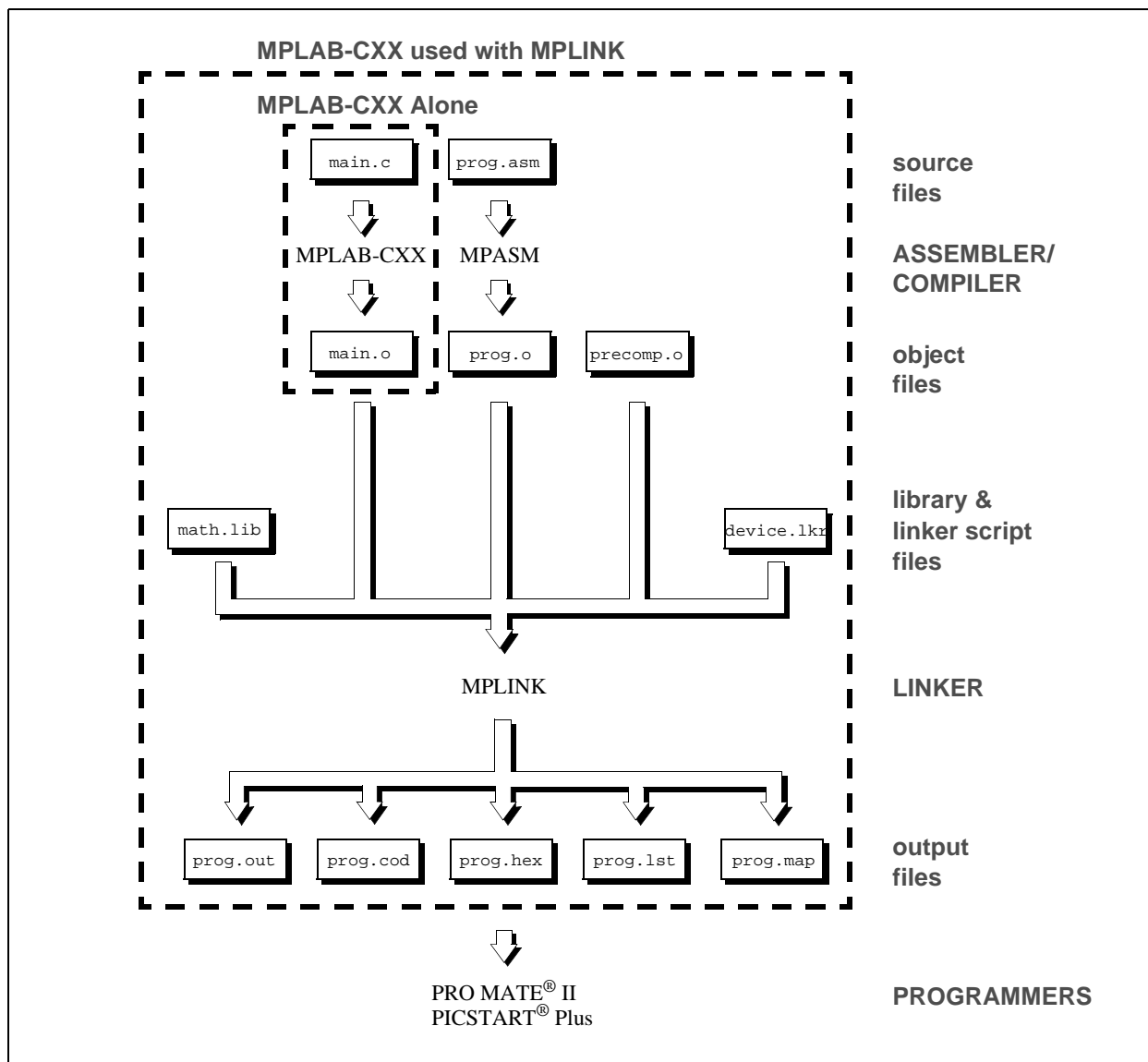


Figure 4.1: MPLAB-CXX – Used Alone and With MPLINK

Using MPLAB-CXX without MPLAB IDE

In this diagram, MPLAB-CXX 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) 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), producing the object file `prog.o` for input into MPLINK. See the *MPASM User's Guide with MPLINK and MPLIB* 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.

MPLAB-C17:

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

MPLAB-C18:

- Standard C libraries
- Processor-specific libraries (peripheral and SFR definitions)

Precompiled object files are often device and/or memory model dependent. For more information on available precompiled object files, see the *MPLAB-CXX Reference Guide*.

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

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

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

- **COFF file (.out)**. Intermediate file used by MPLINK 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.

MPLAB[®]-CXX Compiler User's Guide

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-MPLAB-C17

Option	Default	Description
<code>-?, -h</code>	—	Help screen.
<code>-ipath</code>	—	Add the semicolon delimited path, <i>path</i> , to the search path for include files.
<code>-fo=filename</code>	—	Use <i>filename</i> as the name of the output object file.
<code>-fe=filename</code>	—	Use <i>filename</i> as the name of the output error file.
<code>-O</code>	—	Optimize for smallest code. Equivalent to: <code>-Or -Oc -Op</code>
<code>-Oc [+ -]</code>	Enabled	With this optimization on, the compiler will intelligently determine the level of stack support to include for each function.
<code>-Or [+ -]</code>	Enabled	With this optimization on, the compiler will run an optimization pass to remove extraneous bank select and <code>MOVLW</code> instructions.

Using MPLAB-CXX without MPLAB IDE

Table 4.1: Command Line Option Descriptions-MPLAB-C17 (Continued)

Option	Default	Description
-Ol [+ -]	Enabled	When this optimization is on, the default storage class for local variables and function parameters is 'static'.
-Op [+ -]	Disable	When this optimization is on, far pointers to RAM are assumed to not point to SFRs. This simplifies setting the bank for access.
-m{s m c l}	s	Select the memory model (see Section 8.3.1). 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)
-p=processor	17C44	Select to compile for the designated processor.
-dmacro[=text]	—	Define a macro. Equivalent to placing the following at the head of the file: #define macro text
-w{1 2 3}	2	Set compiler message level. 1 display errors only 2 display errors and warnings 3 display errors, warnings, and messages
-nw#	—	Suppress message number #. Error messages cannot be suppressed.
-q	—	Suppress the sign-on banner (Quiet mode).

4.4.2 Compiling a Single File on the Command Line

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.

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.

MPLAB[®]-CXX Compiler User's Guide

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 generates 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. 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, `c0117.o`, the data initialization file, `idata17.o`, an interrupt handler file, `int756a1.o`, and the processor definition file, `p17c756a.o`, to reference any special function registers. See the *MPLAB-XX Reference Guide* 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.

Using MPLAB-CXX without MPLAB IDE

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 ex1.map -o ex1.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*.

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 ex1.map -o ex1.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);
}
```

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:

```
mccl7 ex1.c -p=17c756a
mccl7 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 ex1.map -o ex1.out
```

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

Using MPLAB-CXX without MPLAB IDE

4.5 Using MPLAB-C18 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

4.5.1 Command Line Interface

The command line interface of MPLAB-C18 is as follows:

```
mcc18 [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:

```
mcc18 -p=18c452 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-C18 can be specified with either `/` or `-`, though the `-` is shown in the table.

Table 4.2: Command Line Option Descriptions-MPLAB-C18

Option	Default	Description
<code>-?, -h</code>	—	Help screen.
<code>-ipath</code>	—	Add the semicolon delimited path, <i>path</i> , to the search path for include files.
<code>-fo=filename</code>	—	Use <i>filename</i> as the name of the output object file.
<code>-fe=filename</code>	—	Use <i>filename</i> as the name of the output error file.
<code>-k</code>	—	Set plain char type to unsigned char.
<code>-ls</code>	—	Large stack (can span multiple banks).
<code>-O</code>	—	Optimize for smallest code. Equivalent to: <code>-Oi -Om -Ob</code>

MPLAB[®]-CXX Compiler User's Guide

Table 4.2: Command Line Option Descriptions-MPLAB-C18 (Continued)

Option	Default	Description
-Oi[- +]	Disabled	Disable/Enable integer promotions.
-Om[- +]	Enabled	Disable/Enable duplicate string merging.
-On[0 1 2]	2	Set banking optimizer level.
-m{s l}	s	Select the memory model (see Section 8.3.1). s: small model (near rom) l: large model (far rom)
-Ou[- +]	Enabled	Disable/Enable unreachable code removal
-Ob[- +]	Enabled	Disable/Enable branch optimizations
-p= <i>processor or family</i>	18C452	Select to compile for the designated processor or family (PIC18CXX)
-dmacro [=text]	—	Define a macro. Equivalent to placing the following at the head of the file: #define macro text
-w{1 2 3}	2	Set compiler message level. 1 display errors only 2 display errors and warnings 3 display errors, warnings, and messages
-nw#	—	Suppress message number #. Error messages cannot be suppressed.
-q	—	Suppress the sign-on banner (Quiet mode).
-lfsr	—	Enables LFSR workaround.
--help-message-list	—	Display a list of all diagnostic messages.
--help-message-all	—	Display help for all diagnostic messages.
--help-message n	—	Display help on diagnostic number n.

Using MPLAB-CXX without MPLAB IDE

4.5.2 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 <p18c452.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 `p18c452.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:

```
mcc18 ex1.c -p=18c452
```

This tells the compiler to compile the program `ex1.c` for the PIC18C452. The compiler generates 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. These messages are also displayed on the screen.

MPLAB[®]-CXX Compiler User's Guide

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, two library files will be necessary. These files are general C libraries, `c1ib.lib`, and the processor-specific libraries, `p18c???.lib`, where '???' is the target processor specific number. See the *MPLAB-XX Reference Guide* for more information on these precompiled library files.

For MPLAB-C18, these library files are listed in the included linker scripts, so there is no need to specifically call them out when linking. However, the path to these files still needs to be specified (`c:\mcc\lib`) for the linker.

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 -k c:\mcc\lkr 18c452.lkr -m
ex1.map -o ex1.out
```

The file `ex1.o` is linked with the library files found in the `c:\mcc\lib` directory specified by the `-l` directive. Specific linker information (including the names of library files) is provided by the linker script file, `18c452.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 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' Guide with MPLINK and MPLIB*.

Summary:

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

```
mcc18 ex1.c -p=18c452
```

- Link to generate `ex1.hex`:

```
mplink ex1.o -l c:\mcc\lib -k c:\mcc\lkr 18c452.lkr -m
ex1.map -o ex1.out
```

Using MPLAB-CXX without MPLAB IDE

4.5.3 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 <p18c452.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);
}
```

File 2

```
/* add.c */
#include <p18c452.h>
unsigned char Add(unsigned char a, unsigned char b)
{ return a+b; }
```

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

```
mccl8 ex1.c -p=18c452
mccl8 add.c -p=18c452
```

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

```
mplink ex1.o add.o -l c:\mcc\lib -k c:\mcc\lkr
18c452.lkr -m ex1.map -o ex1.out
```

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

4.6 Going Forward

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.

You should now know what MPLAB-CXX is and does, how to install it and how to use it on a command line (For information on how to use MPLAB-CXX with MPLAB IDE, see Chapter 5.) In Part 2, you will learn about C programming in general and specifically for PICmicro MCUs, and mixing assembly and C modules in your project. The appendices contain information useful for debugging, such as PICmicro MCU instruction sets and compiler error message definitions.

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-CXX Reference Guide*.

Chapter 5. Using MPLAB-CXX with MPLAB IDE

5.1 Introduction

This chapter discusses how to use MPLAB-CXX with MPLAB IDE.

5.2 Highlights

This chapter includes:

- MPLAB-CXX - MPLAB Projects Overview
- Using MPLAB-C17 with MPLAB – A Tutorial
- Using MPLAB-C18 with MPLAB – A Tutorial
- Going Forward

5.3 MPLAB-CXX – MPLAB Projects Overview

MPLAB-CXX may be used with MPLAB IDE running under Windows 3.x (`mcc17d.exe`), or with MPLAB IDE running under Windows 9x or Windows NT (`mcc17.exe` or `mcc18.exe`).

MPLAB-CXX 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* for more information on MPLAB IDE and MPLAB IDE Projects.

Figure 5.1 shows a generic MPLAB Project using the MPLAB-CXX compiler tool.

MPLAB[®]-CXX Compiler User's Guide

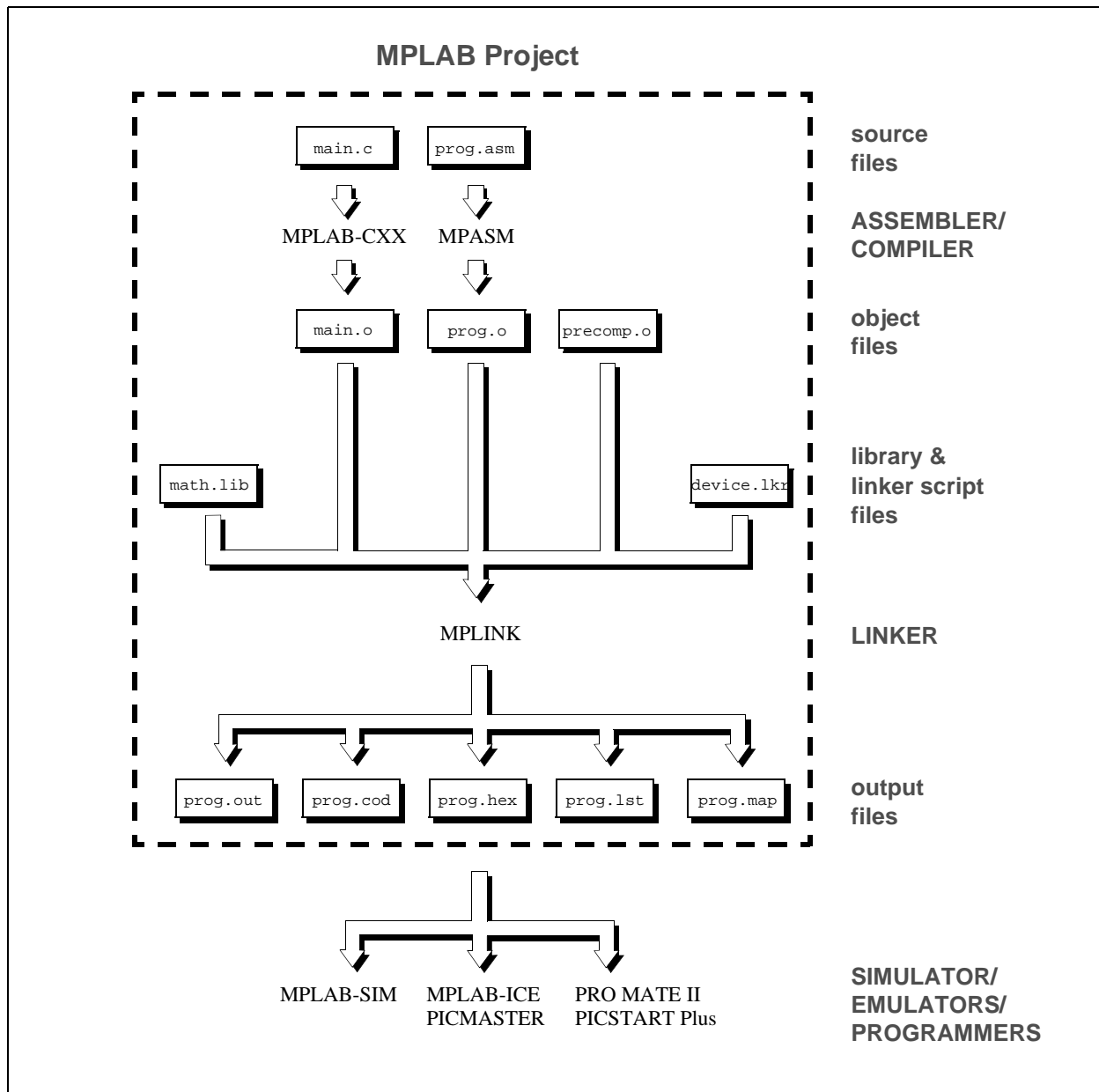


Figure 5.1: An MPLAB Project with MPLAB-CXX – Files and Associated Tools

Using MPLAB-CXX with MPLAB IDE

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

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

MPLAB-C17:

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

MPLAB-C18:

- Standard C libraries
- Processor-specific libraries (peripheral and SFR definitions)

Precompiled object files are often device and/or memory model dependent. For more information on available precompiled object files, see the *MPLAB-CXX Reference Guide*.

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

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). See the *MPASM User's Guide with MPLINK and MPLIB* for more information on linker script files and using the linker.

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

- **COFF file (.out)**. Intermediate file used by MPLINK 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.

MPLAB[®]-CXX Compiler User's Guide

5.4 Using MPLAB-C17 with MPLAB IDE – A Tutorial

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
- Add the source file, setting the language tool to MPLAB-C17
- Add precompiled nodes (object files)
- Add a linker script node
- Build the project

5.4.1 Overview

Figure 5.2 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 according to directions in the linker script file (`p17c756s.lkr`) to produce the main output file, `ex1.hex`.

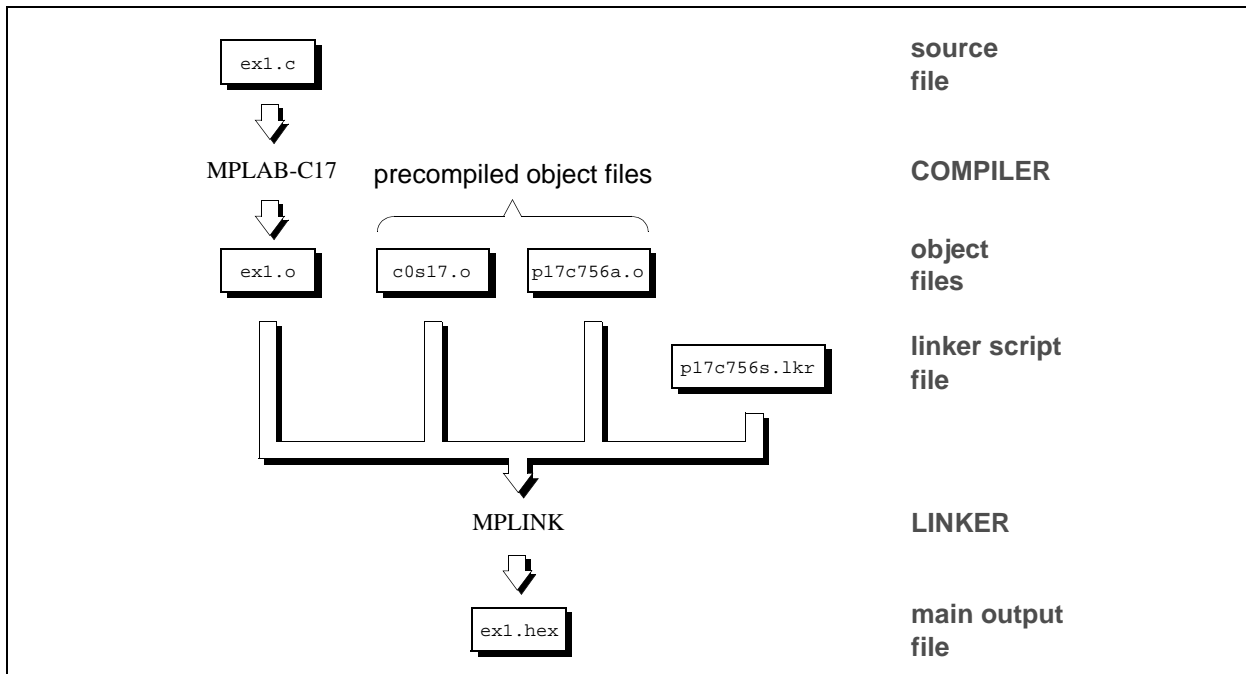


Figure 5.2: An MPLAB Project with MPLAB-C17

Using MPLAB-CXX with MPLAB IDE

5.4.2 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 <pl7c756a.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; }
```

5.4.3 Set Development Mode

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

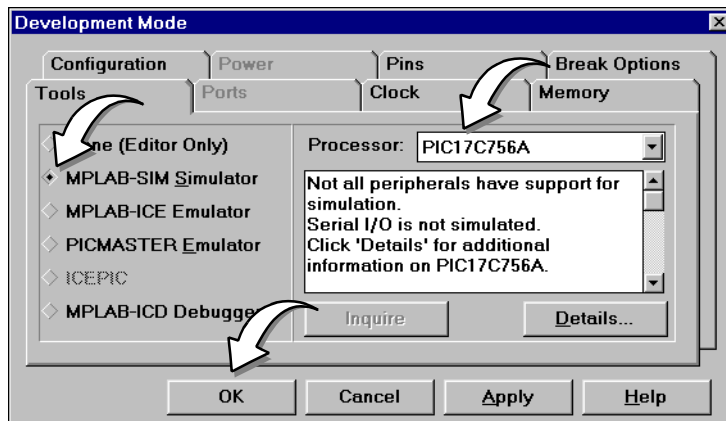


Figure 5.3: Development Mode – PIC17C756A

MPLAB[®]-CXX Compiler User's Guide

5.4.4 New Project

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

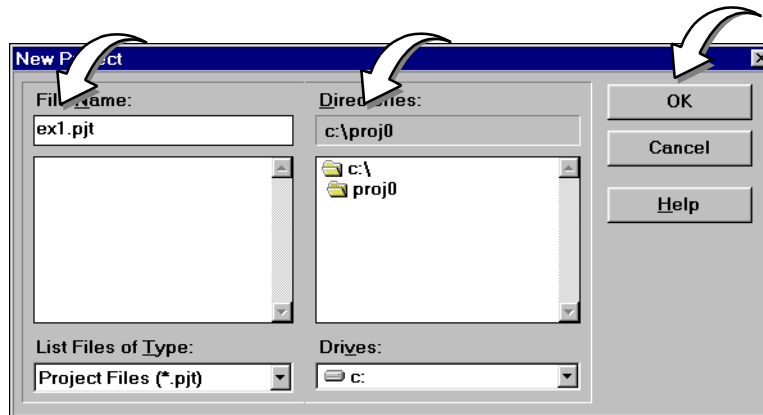


Figure 5.4: New Project – ex1.pjt

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

Using MPLAB-CXX with MPLAB IDE

5.4.5 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**.

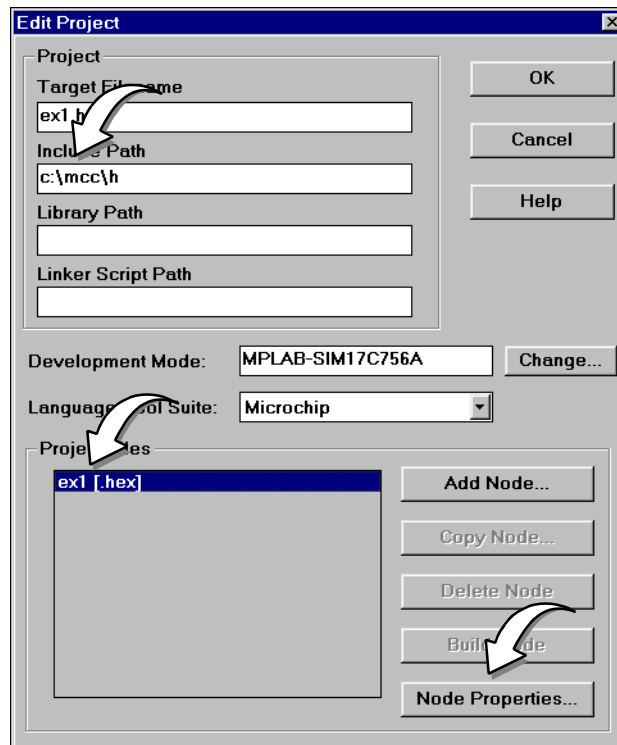


Figure 5.5: Edit Project – ex1.pjt

MPLAB[®]-CXX Compiler User's Guide

5.4.6 Set Node Properties

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

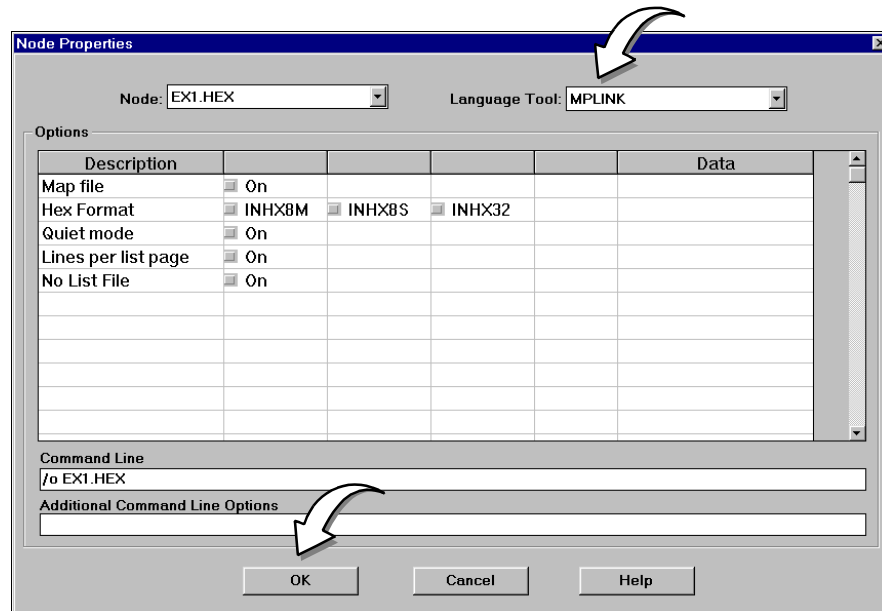


Figure 5.6: Node Properties – ex1.hex

The Node Properties dialog shows the command line switches for the tool, in this case MPLINK. 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 with MPLINK and MPLIB User's Guide* (DS33014) for more information on these command line switches.

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

5.4.7 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.

Using MPLAB-CXX with MPLAB IDE

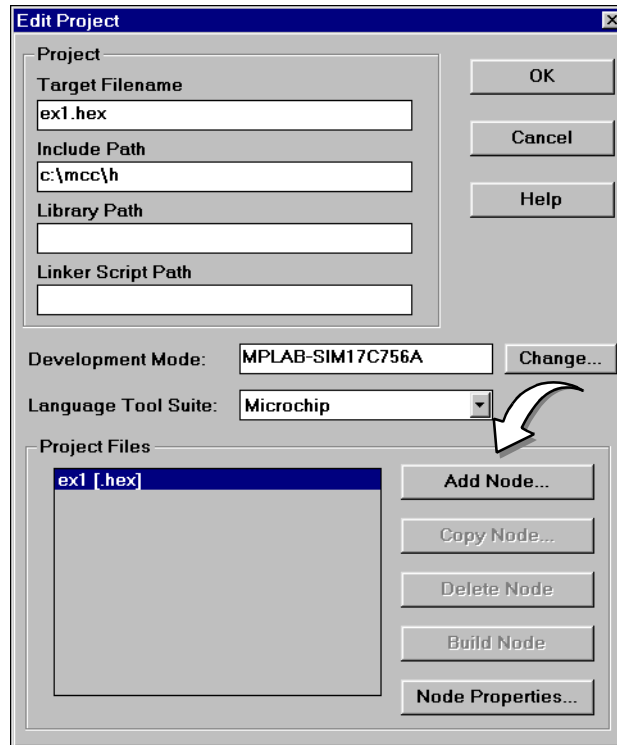


Figure 5.7: Edit Project – Add Node ex1.c

5.4.8 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**.

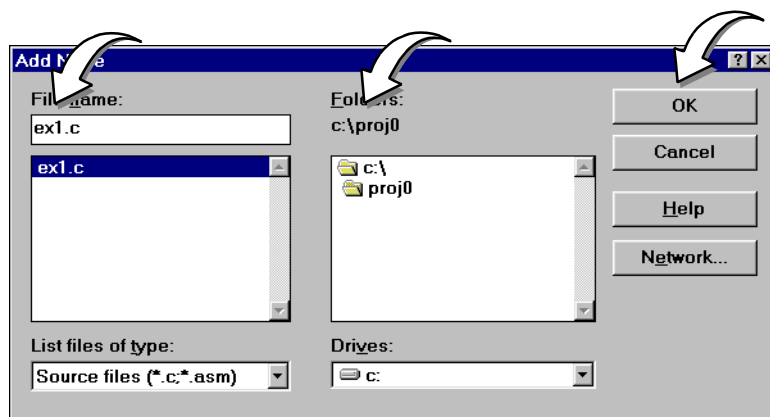


Figure 5.8: Add Node – ex1.c

MPLAB[®]-CXX Compiler User's Guide

The Edit Project dialog should now look like Figure 5.9. Click on `ex1 [.c]` in the Project Files section of the dialog and then click on **Node Properties**.

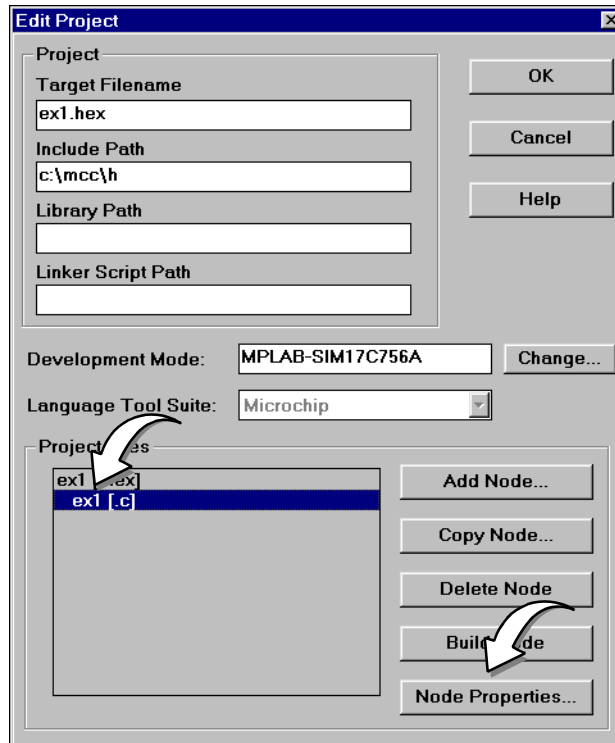


Figure 5.9: Edit Project – ex1.c Added

Using MPLAB-CXX with MPLAB IDE

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.

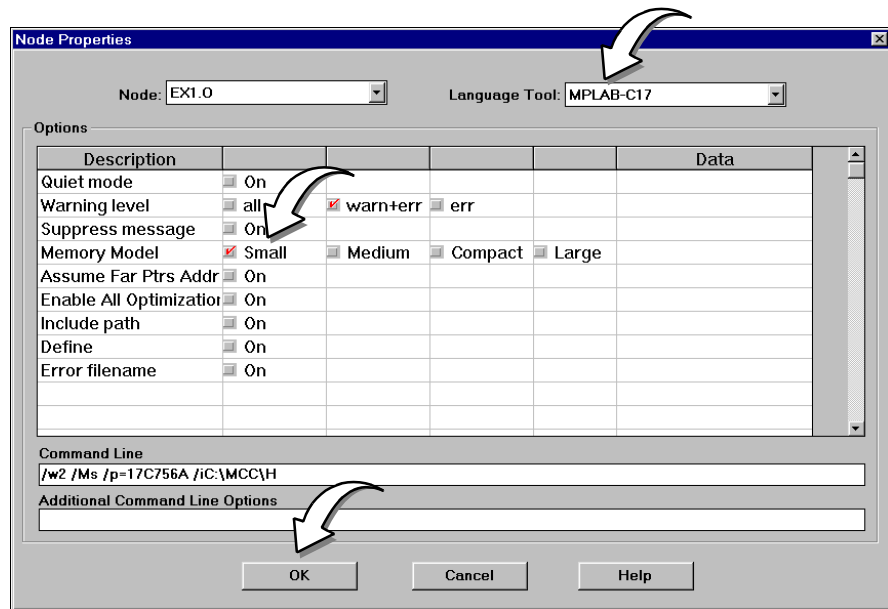


Figure 5.10: Node Properties - ex1.o

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`.

5.4.9 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.

- `c0117.o` – Start Up Code
(source in `c:\mcc\src\startup`)
- `idata17.o` – Code to Initialize Data
(source in `c:\mcc\src\startup`)
- `int756a1.o` – Interrupt Service Routines
(source in `c:\mcc\src\startup`)
- `p17c756a.o` – PIC17C756A Register Definitions
(source in `c:\mcc\src\proc`)

MPLAB[®]-CXX Compiler User's Guide

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 (`int756a1.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)'.
(*Note: The original text contains a typo 'p17c756a.o' which has been corrected to 'p17c756a.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.

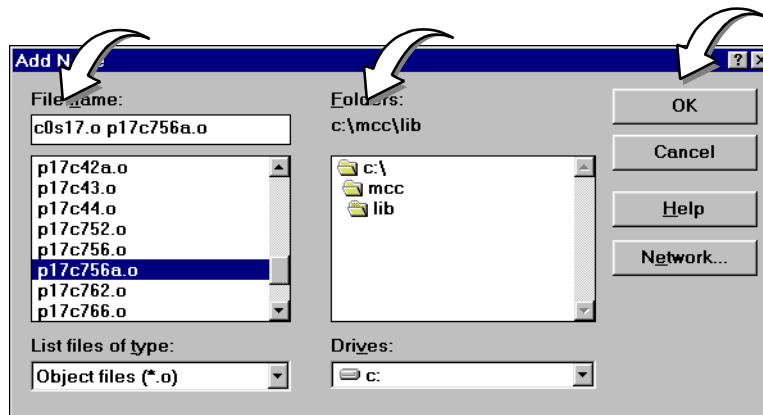


Figure 5.11: Add Node – Object Files

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-CXX Reference Guide – Libraries and Precompiled Object Files*.

Using MPLAB-CXX with MPLAB IDE

5.4.10 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.

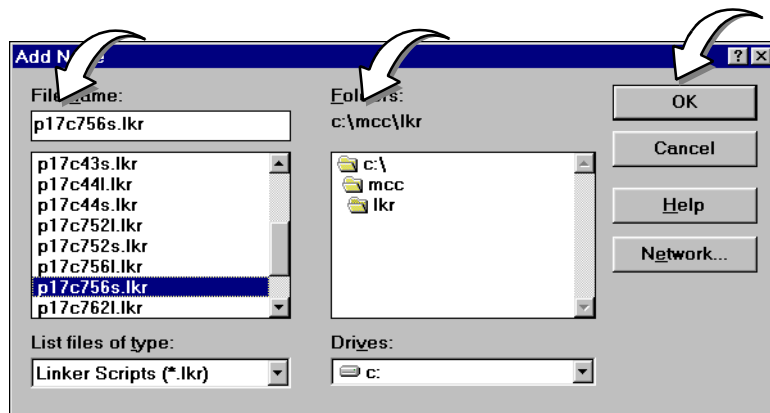


Figure 5.12: Add Node – p17c756s.lkr

MPLAB[®]-CXX Compiler User's Guide

5.4.11 Finish Project Edit

The Edit Project window should now look like this:

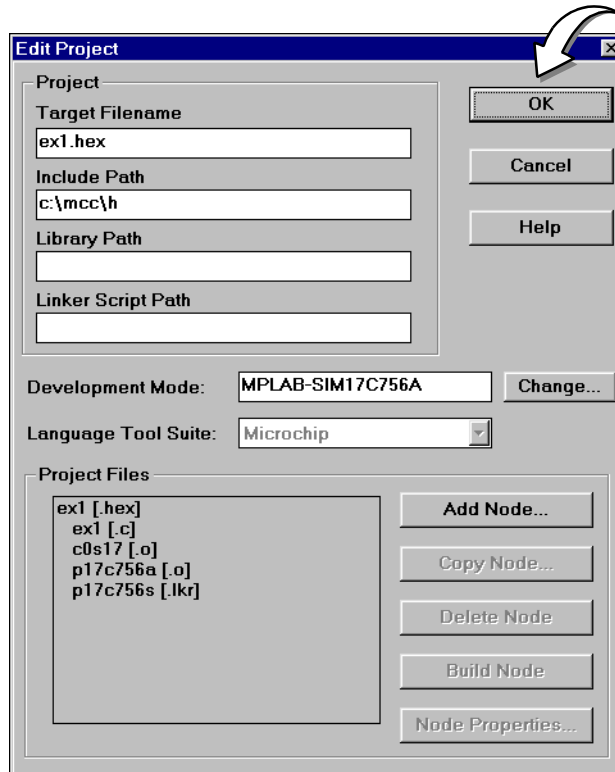


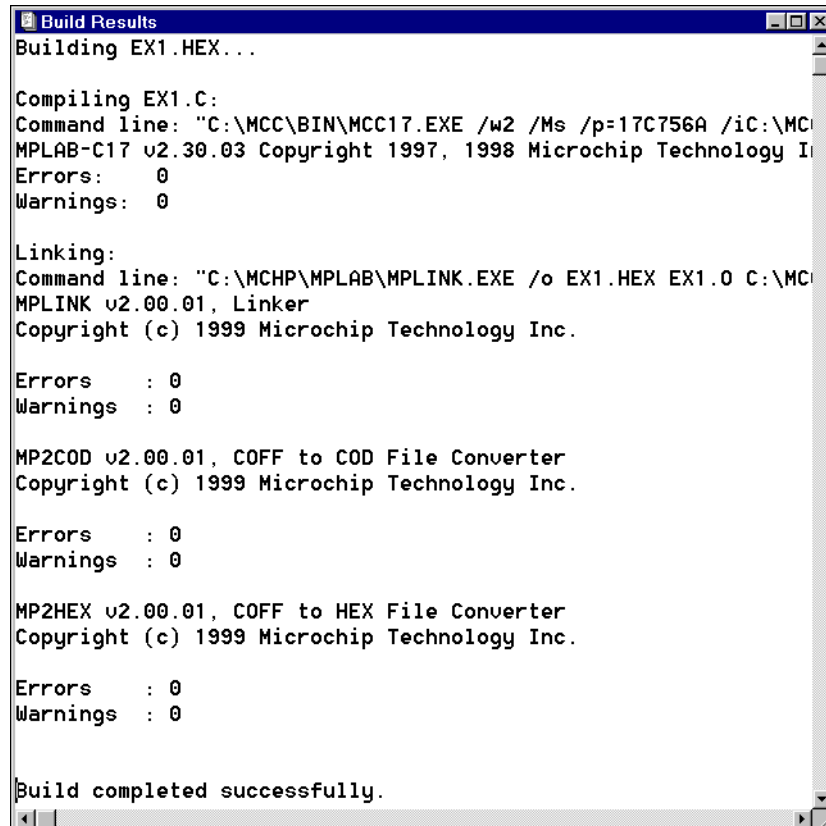
Figure 5.13: Edit Project – ex1.hex

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

Using MPLAB-CXX with MPLAB IDE

5.4.12 Make Project

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



```
Build Results
Building EX1.HEX...

Compiling EX1.C:
Command line: "C:\MCC\BIN\MCC17.EXE /w2 /Ms /p=17C756A /iC:\MC
MPLAB-C17 v2.30.03 Copyright 1997, 1998 Microchip Technology I
Errors:    0
Warnings:  0

Linking:
Command line: "C:\MCHP\MPLAB\MPLINK.EXE /o EX1.HEX EX1.0 C:\MC
MPLINK v2.00.01, Linker
Copyright (c) 1999 Microchip Technology Inc.

Errors    : 0
Warnings  : 0

MP2COD v2.00.01, COFF to COD File Converter
Copyright (c) 1999 Microchip Technology Inc.

Errors    : 0
Warnings  : 0

MP2HEX v2.00.01, COFF to HEX File Converter
Copyright (c) 1999 Microchip Technology Inc.

Errors    : 0
Warnings  : 0

Build completed successfully.
```

Figure 5.14: Build Results – ex1.hex

5.4.13 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 5.15). 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.

MPLAB[®]-CXX Compiler User's Guide

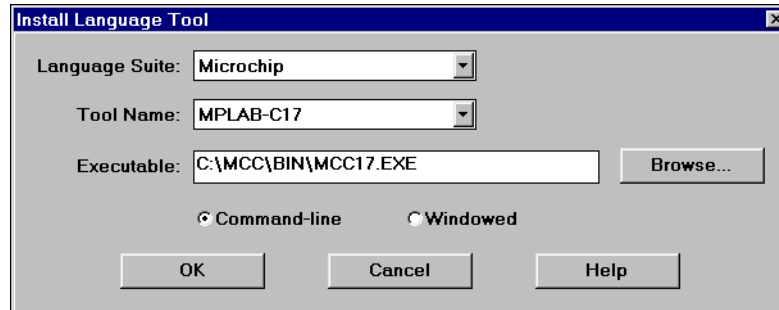


Figure 5.15: Install Language Tool – MPLAB-C17

2. Select *Project > Install Language Tool...* and check that MPLINK is pointing to the `mplink.exe` executable (Figure 5.16). Your executable path may be different from the figure.

The Command-line option should be selected.

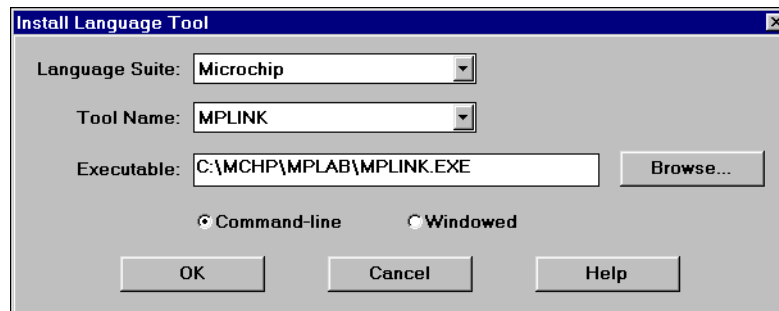


Figure 5.16: Install Language Tool – MPLINK

3. Check the Node Properties for the Project Files `ex1.hex` and `ex1.c`. They should be mapped to the Language Tools MPLINK and MPLAB-C17 respectively.
4. Check the names of the files added to the project against the ones listed in Figure 5.13. 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`).

Using MPLAB-CXX with MPLAB IDE

5.4.14 Project Window

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

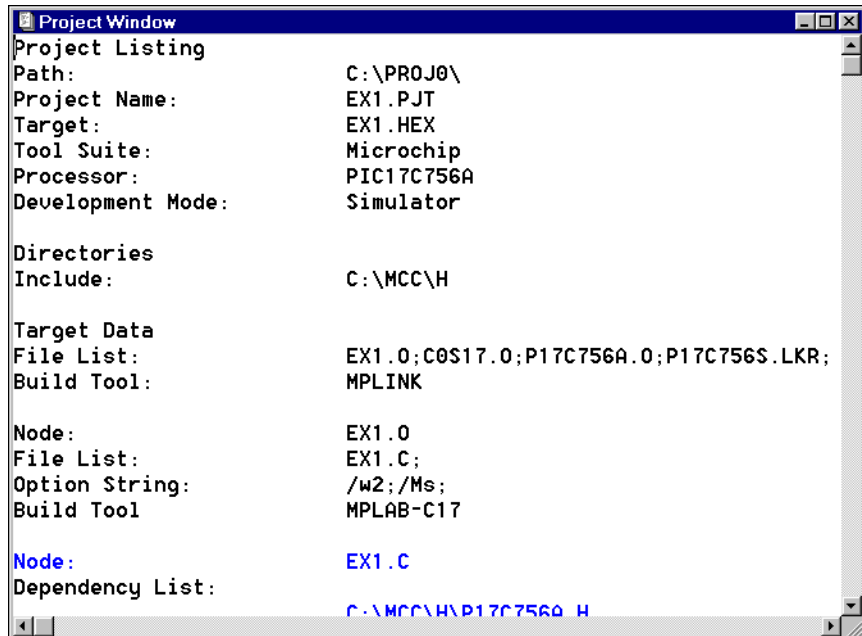


Figure 5.17: Project Window – ex1.pjt

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*.

MPLAB[®]-CXX Compiler User's Guide

5.5 Using MPLAB-C18 with MPLAB IDE – A Tutorial

This section will guide you, step by step, in using MPLAB IDE and MPLAB Projects with MPLAB-C18.

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
- Add the source file, setting the language tool to MPLAB-C18
- Add precompiled nodes and library files
- Add a linker script node
- Build the project

5.5.1 Overview

Figure 5.2 gives a graphical overview of the MPLAB Project using MPLAB-C18. The source file `ex1.c` is associated with the compiler (MPLAB-C18) to produce the object file `ex1.o`. This file and other precompiled object files are linked via MPLINK according to directions in the linker script file (`18c452.lkr`) to produce the main output file, `ex1.hex`.

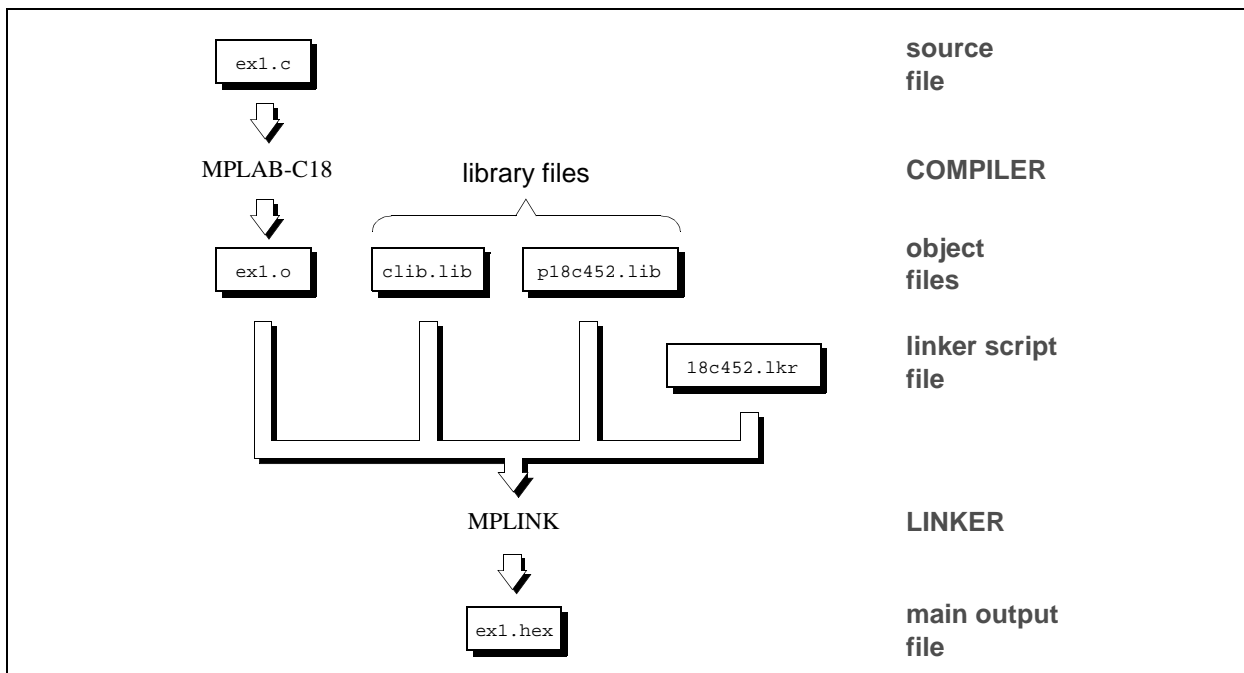


Figure 5.18: An MPLAB Project with MPLAB-C18

Using MPLAB-CXX with MPLAB IDE

5.5.2 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 <pl8c452.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; }
```

5.5.3 Set Development Mode

Set *Options > Development Mode* to MPLAB-SIM simulator and select the PIC18C452 PICmicro MCU for this example. Click **Reset**.

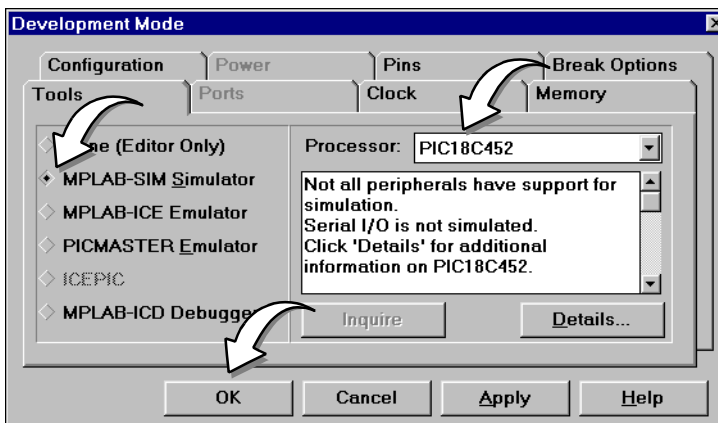


Figure 5.19: Development Mode – PIC18C452

MPLAB[®]-CXX Compiler User's Guide

5.5.4 New Project

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

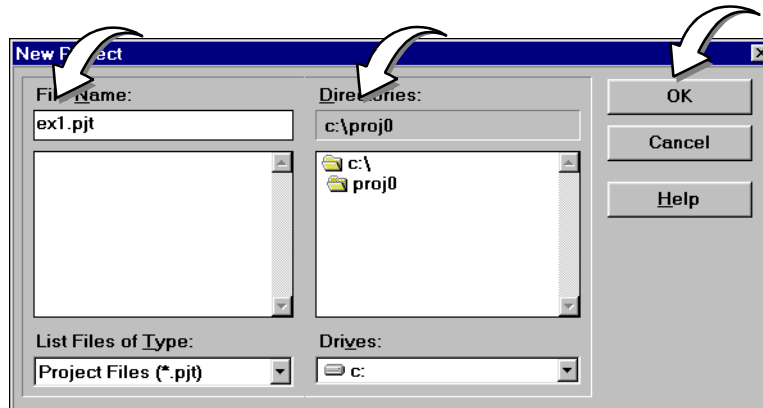


Figure 5.20: New Project – ex1.pjt

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

Using MPLAB-CXX with MPLAB IDE

5.5.5 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**.

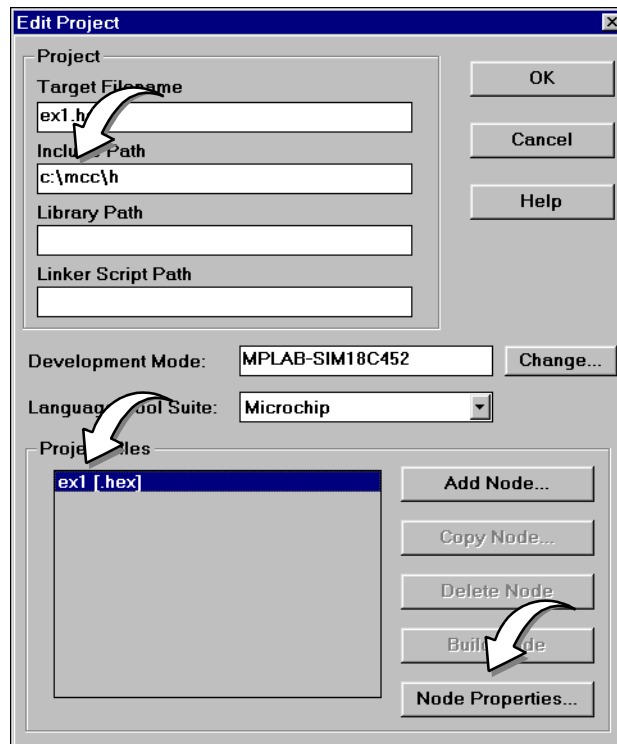


Figure 5.21: Edit Project – ex1.pjt

MPLAB[®]-CXX Compiler User's Guide

5.5.6 Set Node Properties

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

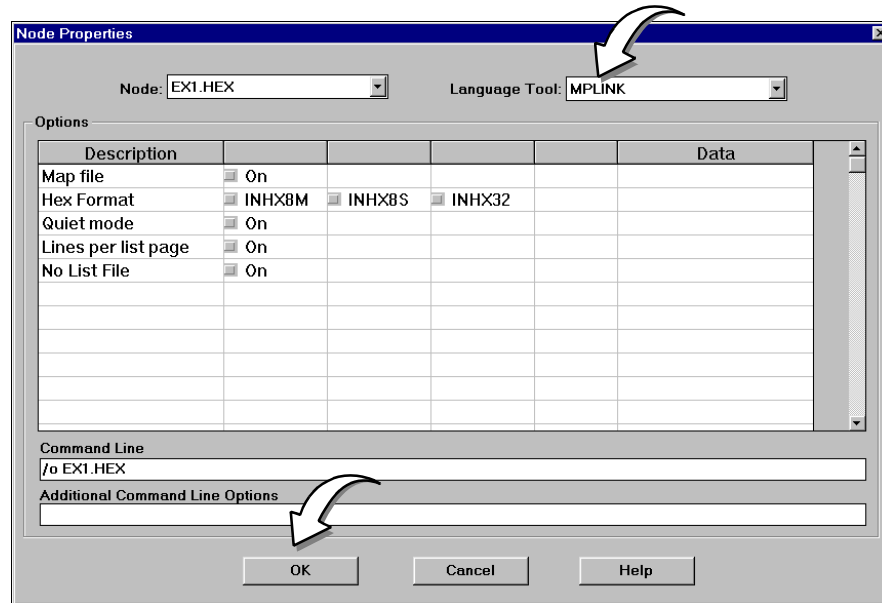


Figure 5.22: Node Properties – ex1.hex

The Node Properties dialog shows the command line switches for the tool, in this case MPLINK. 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 with MPLINK and MPLIB User's Guide* (DS33014) for more information on these command line switches.

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

5.5.7 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.

Using MPLAB-CXX with MPLAB IDE

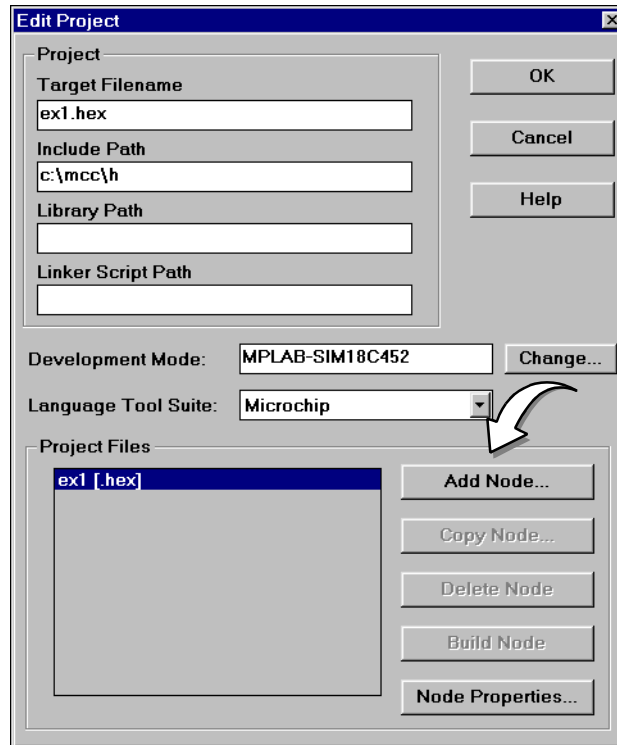


Figure 5.23: Edit Project – Add Node ex1.c

5.5.8 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**.

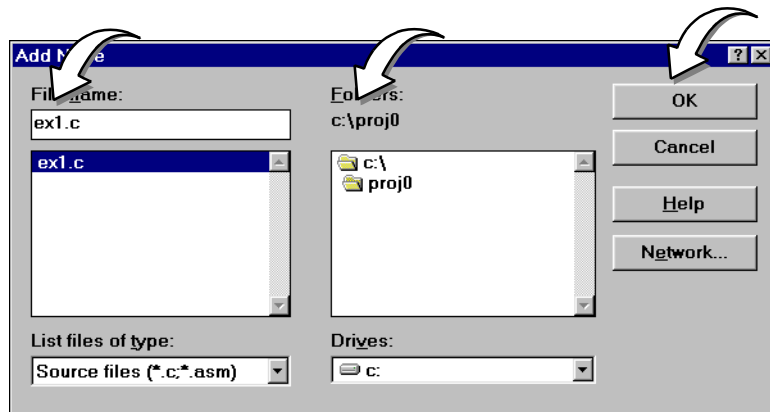


Figure 5.24: Add Node – ex1.c

MPLAB[®]-CXX Compiler User's Guide

The Edit Project dialog should now look like Figure 5.25. Click on `ex1 [.c]` in the Project Files section of the dialog and then click on **Node Properties**.

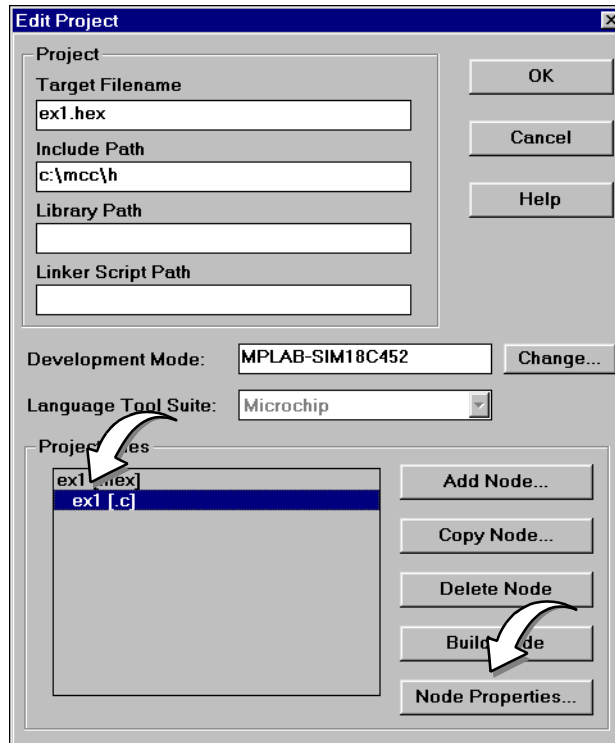


Figure 5.25: Edit Project – ex1.c Added

Using MPLAB-CXX with MPLAB IDE

In the Node Properties dialog, verify that the language tool is set to MPLAB-C18.

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 bank selection.

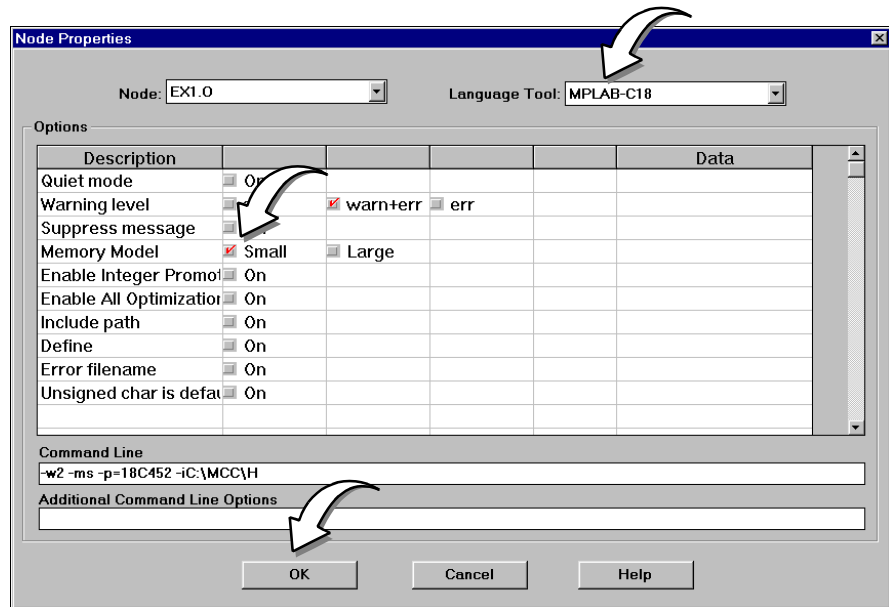


Figure 5.26: Node Properties – ex1.o

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`.

5.5.9 Add Precompiled Object Files

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

- `c1ib.lib` – Standard C Libraries, Start Up and Initialization Code
- `p18c452.lib` – PIC18C452 Processor-Specific Routines, including peripheral access and special function register definitions

For MPLAB-C18, these precompiled object files are listed in the included linker scripts, so there is no need to specifically call them out when linking. However, the path to these files still needs to be specified for the linker.

MPLAB[®]-CXX Compiler User's Guide

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

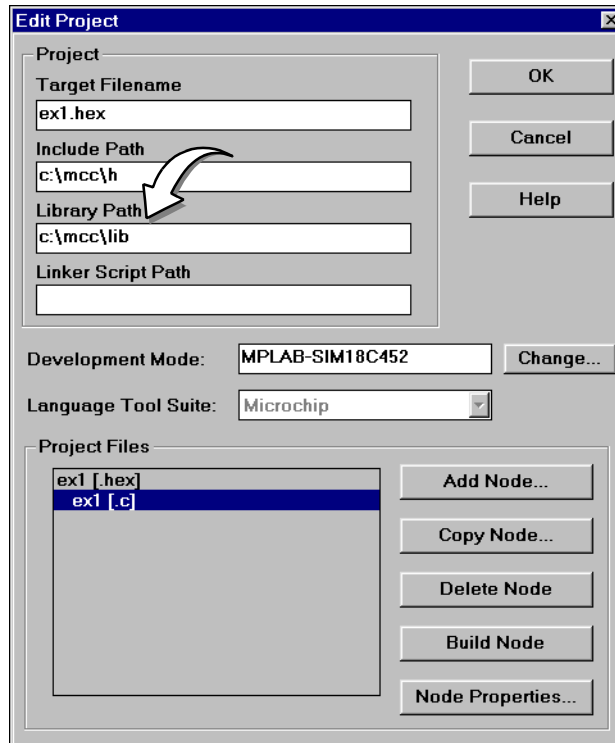


Figure 5.27: Edit Project – Library Path Added

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

- In the Edit Project window, click **Add Node**.
- In the Add Node dialog, select the Folder `c:\mcc\lib`.
- Select the desired library from the “File name” list. Make sure ‘List files of type:’ specifies ‘Libraries (*.lib)’.
- Click **OK**.

Library files do not have node properties as they are already compiled.

For more information on libraries and precompiled object files, please refer to the *MPLAB-CXX Reference Guide – Libraries and Precompiled Object Files*.

Using MPLAB-CXX with MPLAB IDE

5.5.10 Select Linker Script

Use the **Add Node** button from the Edit Project dialog to add the linker script file `18c452.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.

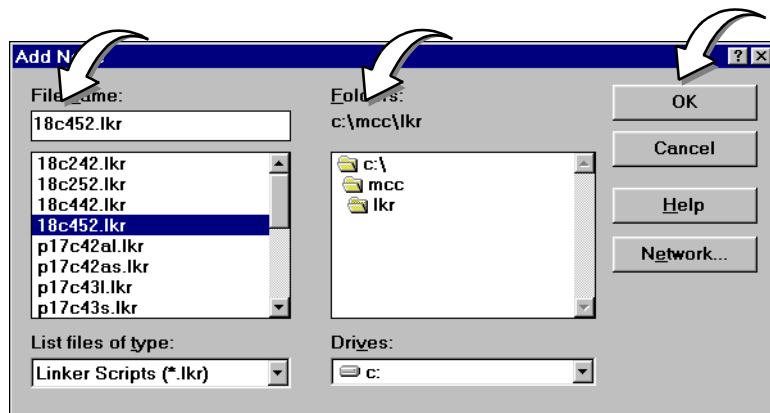


Figure 5.28: Add Node – p18c452.lkr

MPLAB[®]-CXX Compiler User's Guide

5.5.11 Finish Project Edit

The Edit Project window should now look like this:

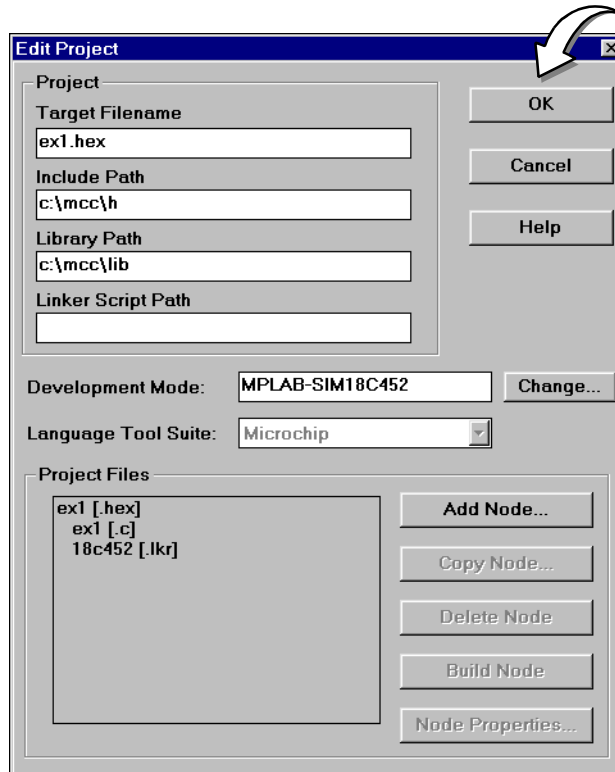


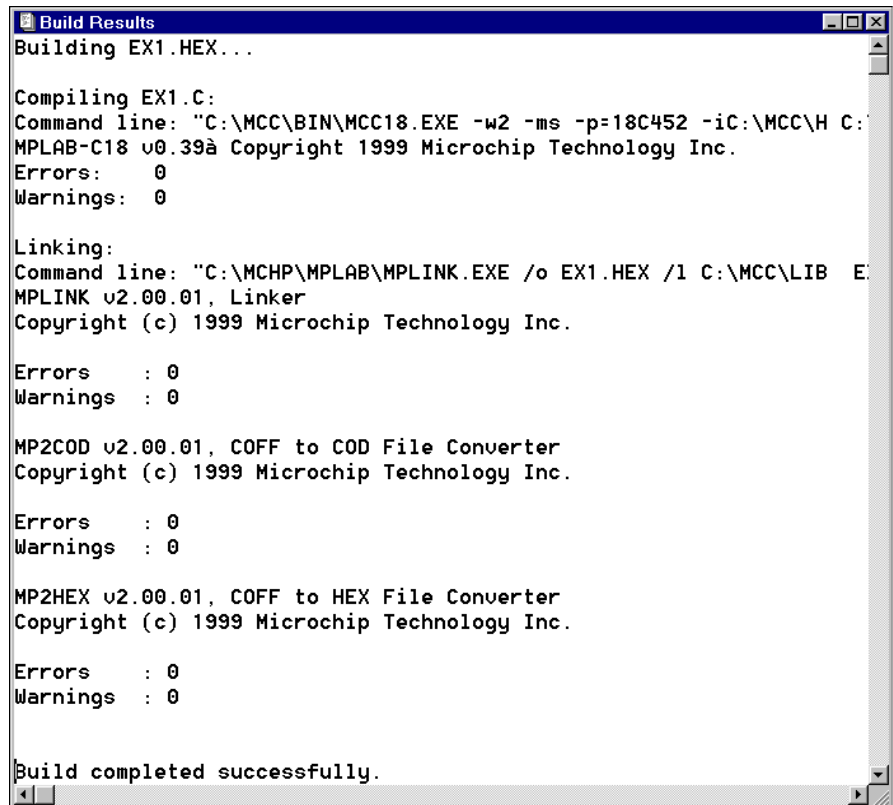
Figure 5.29: Edit Project – ex1.hex

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

Using MPLAB-CXX with MPLAB IDE

5.5.12 Make Project

Select *Project > Make Project* from the menu to compile the application using MPLAB-C18 and MPLINK. A Build Results window is created that shows the command lines sent to each tool. It should look like this:



```
Build Results
Building EX1.HEX...

Compiling EX1.C:
Command line: "C:\MCC\BIN\MCC18.EXE -w2 -ms -p=18C452 -iC:\MCC\H C:
MPLAB-C18 v0.39â Copyright 1999 Microchip Technology Inc.
Errors:      0
Warnings:    0

Linking:
Command line: "C:\MCHP\MPLAB\MPLINK.EXE /o EX1.HEX /1 C:\MCC\LIB E
MPLINK v2.00.01, Linker
Copyright (c) 1999 Microchip Technology Inc.

Errors      : 0
Warnings    : 0

MP2COD v2.00.01, COFF to COD File Converter
Copyright (c) 1999 Microchip Technology Inc.

Errors      : 0
Warnings    : 0

MP2HEX v2.00.01, COFF to HEX File Converter
Copyright (c) 1999 Microchip Technology Inc.

Errors      : 0
Warnings    : 0

Build completed successfully.
```

Figure 5.30: Build Results – ex1.hex

5.5.13 Troubleshooting

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

1. Select *Project > Install Language Tool...* and check that MPLAB-C18 references the `mcc18.exe` executable (Figure 5.15). Your executable path may be different from the figure.

The Command-line option should be selected.

MPLAB[®]-CXX Compiler User's Guide

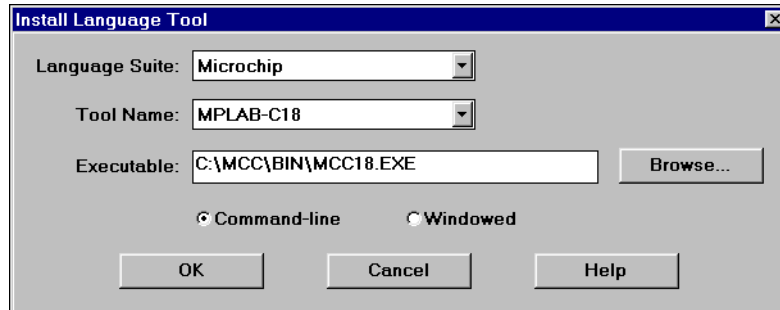


Figure 5.31: Install Language Tool – MPLAB-C18

2. Select *Project > Install Language Tool...* and check that MPLINK is pointing to the `mplink.exe` executable (Figure 5.32). Your executable path may be different from the figure.

The Command-line option should be selected.

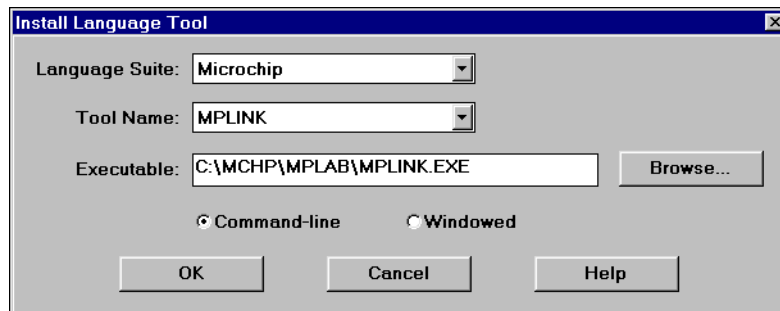


Figure 5.32: Install Language Tool – MPLINK

3. Check the Node Properties for the Project Files `ex1.hex` and `ex1.c`. They should be mapped to the Language Tools MPLINK and MPLAB-C18 respectively.
4. Check the names of the files added to the project against the ones listed in Figure 5.29. 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`).

Using MPLAB-CXX with MPLAB IDE

5.5.14 Project Window

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

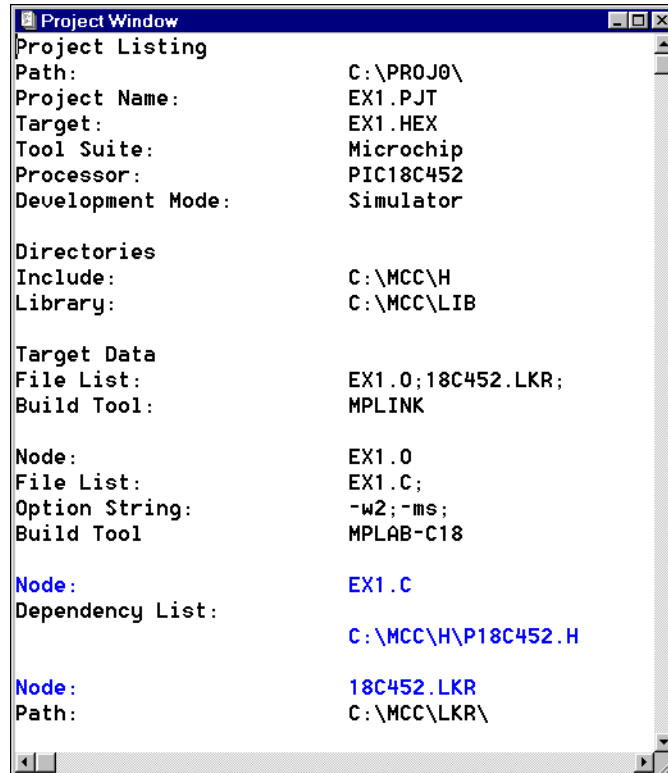


Figure 5.33: Project Window – ex1.pjt

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*.

5.6 Going Forward

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*.

You should now know what MPLAB-CXX is and does, how to install it and how to use it with MPLAB Projects (For information on using MPLAB-CXX on a command line, see the previous chapter.) In Part 2, you will learn about C programming in general and specifically for PICmicro MCUs, and mixing assembly and C modules in your project. The appendices contain information useful for debugging, such as PICmicro MCU instruction sets and compiler error message definitions.

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-CXX Reference Guide*.



MPLAB[®]-CXX COMPILER USER'S GUIDE

Part 2 – Using MPLAB-CXX

Chapter 6. MPLAB-CXX and C.....	71
Chapter 7. MPLAB-CXX Fundamentals	75
Chapter 8. MPLAB-CXX and PICmicro MCU Programming.....	123
Chapter 9. Mixing Assembly Language and C Modules	139
Chapter 10. ANSI Implementation Issues.....	145
Chapter 11. Examples	151

MPLAB[®]-CXX Compiler User's Guide

Chapter 6. MPLAB-CXX and C

6.1 Introduction

This chapter compares MPLAB-CXX with C programming. A basic understanding of C programming is assumed. Programmers who are unfamiliar with the C language can refer to Appendix F for a list of C programming references.

For those familiar with C but not experienced with programming microcontrollers, various points are highlighted and deviations from ANSI C are described. Also, device data sheets describing the operation of various PICmicro MCU devices are available from any sales office listed on the last page of this document, or from our web site (<http://www.microchip.com>).

6.2 Highlights

This chapter covers the following topics:

- MPLAB-CXX vs. C
- Components of a Basic MPLAB-CXX Program
- C Keywords

6.3 C vs. MPLAB-CXX

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, it is left up to the programmer to determine memory allocation, I/O operation through a peripheral, etc. Libraries and precompiled object files are available with MPLAB-CXX to aid the programmer in this endeavor.

An MPLAB-CXX 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

MPLAB[®]-CXX Compiler User's Guide

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.

6.4 Components of a Basic MPLAB-CXX Program

The following is the shell for a basic MPLAB-CXX 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 */
}
```

`p17cxx.h` includes proper processor-specific header file based on the processor selected on the command line.

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

The next line is a declaration of a function that will be used in the main routine (`function1`). Functions are discussed further in Section 7.7. 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.

6.5 C Keywords

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-CXX keywords, where MPLAB-CXX keywords are shown in bold.

Table 6.1: ANSI C and MPLAB-CXX Keywords

Keywords	Defined	Keywords	Defined
<code>_asm</code>	Section 8.3.7	<code>int</code>	Section 7.6.1
<code>_endasm</code>	Section 8.3.7	<code>long*</code>	Section 7.6.1
<code>auto</code>	Section 7.6.1	<code>near</code>	Section 7.6.1
<code>break</code>	Section 7.9.7	<code>ram</code>	Section 7.6.1
<code>case</code>	Section 7.9.6	<code>register**</code>	Section 7.6.1
<code>char</code>	Section 7.6.1	<code>return</code>	Section 7.7.4
<code>const</code>	Section 7.6.1	<code>rom</code>	Section 7.6.1
<code>continue</code>	Section 7.9.8	<code>short</code>	Section 7.6.1
<code>default</code>	Section 7.9.6	<code>signed</code>	Section 7.6.1
<code>do</code>	Section 7.9.5	<code>sizeof</code>	Section 7.8.8
<code>double*</code>	Section 7.6.1	<code>static</code>	Section 7.6.3
<code>else</code>	Section 7.9.2	<code>struct</code>	Section 7.12.2
<code>enum</code>	Section 7.6.4	<code>switch</code>	Section 7.9.6
<code>extern</code>	Section 7.6.3	<code>typedef</code>	Section 7.6.5
<code>far</code>	Section 7.6.1	<code>union</code>	Section 7.12.3
<code>float*</code>	Section 7.6.1	<code>unsigned</code>	Section 7.6.1
<code>for</code>	Section 7.9.3	<code>void</code>	Section 7.6.1
<code>goto</code>	Section 7.9.9	<code>volatile</code>	Section 7.6.3
<code>if</code>	Section 7.9.1	<code>while</code>	Section 7.9.4
<p>* <code>double</code>, <code>float</code> and <code>long</code> are not supported by MPLAB-C17. <code>double</code> and <code>float</code> are the same in MPLAB-C18.</p> <p>** has no effect in MPLAB-CXX</p>			

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Chapter 7. MPLAB-CXX Fundamentals

7.1 Introduction

MPLAB-CXX Fundamentals describes the C programming language as used with PICmicro MCU devices. A basic understanding of C programming is assumed. Programmers who are unfamiliar with the C language can refer to Appendix F for a list of C programming references.

For those familiar with C but not experienced with programming microcontrollers, various points are highlighted and deviations from ANSI C are described. Also, device data sheets describing the operation of various PICmicro MCU devices are available from any sales office listed on the last page of this document, or from our website (<http://www.microchip.com>).

7.2 Highlights

This chapter covers the following topics:

- Preprocessor Directives
- Comments
- Constants
- Variables
- Functions
- Operators
- Program Control Statements
- Arrays and Strings
- Pointers
- Structures and Unions

7.3 Preprocessor Directives

Preprocessor directives give instructions on how to compile the source code. Preprocessor directives generally do not translate directly into executable code.

Preprocessor directives begin with the '#' character. This section discusses the following preprocessor directives:

- `#define`
- `#else`
- `#elif`
- `#endif`
- `#error`
- `#if`
- `#ifdef`
- `#ifndef`
- `#include`
- `#line`
- `#pragma text`
- `#undef`

7.3.1 `#define`

7.3.1.1 Description

The `#define` directive defines string constants that are substituted into a source line before the source line is evaluated. These can improve source code readability and maintainability. Common uses are to define constants that are used in many places and provide short cuts to more complex expressions.

7.3.1.2 Syntax

define-directive:

```
#define identifier pp-token-list new-line  
#define identifier lparen parameter-list )  
pp-token-list new-line  
#define identifier lparen ) pp-token-list new-line
```

lparen:
(¹

1. No whitespace may separate *lparen* and the macro name.

```
parameter-list:  
    identifier  
    parameter-list , identifier
```

7.3.1.3 Example

```
#define MAX_COUNT 100  
#define VERSION "v1.0"  
#define PERIMETER( x, y ) 2*x + 2*y
```

7.3.2 #else

7.3.2.1 Description

Refer to #if, #ifdef, and #ifndef for a description of the #else directive.

7.3.3 #elif

7.3.3.1 Description

Refer to #if, #ifdef, and #ifndef for a description of the #elif directive.

7.3.4 #endif

7.3.4.1 Description

Refer to #if, #ifdef, and #ifndef for a description of the #endif directive.

7.3.5 #error

7.3.5.1 Description

The #error directive generates a user-defined error message at compile time. One use of #error is to detect cases where the source code generates constants that are out of range. No code is generated as a result of using this directive.

7.3.5.2 Syntax

```
error-directive:  
    #error pp-token-list new-line
```

7.3.5.3 Example

```
#define MAX_COUNT 100
#define ELEMENT_SIZE 3
#if (MAX_COUNT * ELEMENT_SIZE) > 256
    #error "Data size too large."
#endif
```

7.3.6 #if

7.3.6.1 Description

The `#if` directive is useful for conditionally compiling code based on the evaluation of an expression. `#if` must be terminated by `#endif`. The `#elif` is used to test a new expression. The directive `#else` is also available to provide an alternative compilation. The `defined()` operator acts similarly to `#ifdef` when combined with `#if`.

7.3.6.2 Syntax

```
if-directive:
    #if constant-expression new-line
```

7.3.6.3 Example

```
#define MAX_COUNT 100
#define ELEMENT_SIZE 3
#if defined(MAX_COUNT) && defined(ELEMENT_SIZE)
#if (MAX_COUNT * ELEMENT_SIZE) > 256
    #error "Data size too large."
#else
    #define DATA_SIZE MAX_COUNT * ELEMENT_SIZE
#endif
#endif
```

7.3.7 #ifdef

7.3.7.1 Description

The `#ifdef` directive is similar to the `#if` directive, except that instead of evaluating an expression, it checks to see if the specified symbol has been defined. Like the `#if` directive, `#ifdef` must be terminated by `#endif`, and can optionally be used with `#else`.

7.3.7.2 Syntax

```
ifdef-directive:
    #ifdef identifier new-line
```


7.3.7.3 Example

```
#ifdef DEBUG
    Count = MAX_COUNT;
#endif
```

7.3.8 #ifndef

7.3.8.1 Description

The `#ifndef` directive is similar to the `#ifdef` directive, except that it checks to see if the specified symbol has not been defined. Like the `#if` directive, `#ifndef` must be terminated by `#endif`, and can optionally be used with `#else`.

7.3.8.2 Syntax

```
ifndef-directive:
    #ifndef identifier new-line
```

7.3.8.3 Example

```
#ifndef DEBUG
#define Debug(x)
#else
#define Debug(x) x
#endif
```

7.3.9 #include

7.3.9.1 Description

`#include` inserts the full text from another file at this point in the source code. The inserted file may contain any number of valid C statements.

7.3.9.2 Syntax

```
include-directive:
    #include " filename " new-line
    #include < filename > new-line
    #include pp-token-list new-line
```

When “filename” is used, MPLAB-CXX 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’.

MPLAB[®]-CXX Compiler User's Guide

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

7.3.9.3 Example

```
#include <p17cxx.h>
#include "header.h"
```

7.3.10 #line

7.3.10.1 Description

The line directive causes the compiler to renumber the source text so that the following line has the specified line number.

7.3.10.2 Syntax

```
line-directive:
    #line digit-sequence new-line
    #line digit-sequence " filename " new-line
    #line pp-token-list new-line
```

7.3.10.3 Example

```
#line 34          /* This line is line 34 */
#line 55 "main.c" /* This line is line 55 of main.c */
```

7.3.11 #pragma interrupt fname – MPLAB-C17

7.3.11.1 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.

7.3.11.2 Syntax

```
interrupt-directive:
    #pragma interrupt function-name [section-name]
        save=symbol-list new-line
symbol-list:
    symbol-name
    symbol-list , symbol-name
```

7.3.12 #pragma interrupt fname – MPLAB-C18 #pragma interruptlow fname – MPLAB-C18

7.3.12.1 Description

The `interrupt` pragma declares a function to be an interrupt function. MPLAB-CXX will save and restore a basic context of `W`, `BSR`, and `STATUS` by default, and the `save=` clause allows additional arbitrary symbols to be saved and restored by the function. The function will terminate with a `RETFIE` instruction.

Any temporary locations required during the evaluation of expressions in the function will be allocated from a private memory section and will not be overlaid with the temporary locations of any other function, including other interrupt functions. Optionally, a name can be specified for the section into which temporary locations are allocated.

Both high and low priority interrupts are supported. The `interrupt` pragma declares a high priority interrupt, and the `interruptlow` pragma declares a low priority interrupt. A high priority interrupt uses the shadow registers to save and restore `W`, `BSR`, and `STATUS`, while a low priority interrupt uses the software stack to save and restore `W`, `BSR`, and `STATUS`. As a consequence of this, a high priority interrupt terminates with a fast `RETFIE`, while a low priority interrupt terminates with a slow `RETFIE`.

Two `MOVFF` instructions are required for each byte of context to be saved and restored via the software stack; therefore, in order to save and restore the `W`, `BSR`, and `STATUS` registers, a low priority interrupt has an additional 12-word overhead beyond the requirements of a high priority interrupt.

MPLAB-CXX does not automatically place an interrupt function at the interrupt vector. An absolute code section may be used to locate the interrupt function. More commonly, a `GOTO` instruction is placed at the interrupt vector for transferring control to the interrupt function proper.

7.3.12.2 Syntax

```
interrupt-directive:  
    #pragma interrupt function-name [section-name]  
        save=symbol-list new-line  
    #pragma interruptlow function-name [section-name]  
        save=symbol-list new-line  
  
symbol-list:  
    symbol-name  
    symbol-list , symbol-name
```

7.3.12.3 Example

Declare a low-priority interrupt function `myInterrupt` not located at the interrupt vector. Save several additional symbols of context.

```
void myInterrupt (void);
```

MPLAB[®]-CXX Compiler User's Guide

```
#pragma code lowVector=0x18
void atLowVector (void)
{
    _asm GOTO myInterrupt _endasm
}
#prama code /* return to default code section */

#pragma interruptlow myInterrupt save=FSR0, PROD
void myInterrupt (void)
{
    *globalCharPointer += PORTB;
}
```

7.3.13 #pragma list / #pragma nolist

7.3.13.1 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.

7.3.13.2 Syntax

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

7.3.14 #pragma sectiontype

7.3.14.1 Description

Sections

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 section of the *MPASM User's Guide with MPLINK and MPLIB*.

The section declaration family of pragmas changes the section into which MPLAB-CXX 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-CXX 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.

For pragma optimization tips, see Section 8.5.7.

7.3.14.2 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:
    access1
    shared2
    overlay
location:
    = address
    gpr bank-number2
    sfr bank-number2
```

7.3.14.3 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 udata section which will be located in access memory (MPLAB-C18 only).

```
#pragma udata access myAccessDataSection
```

Declare a new code section at address 0x8000.

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

7.3.14.4 See also

```
#pragma varlocate
```

1. MPLAB-C18 only
2. MPLAB-C17 only

7.3.15 #pragma varlocate n name #pragma varlocate {gpr | sfr} name – C17 only

7.3.15.1 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, in the case of MPLAB-C17, 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.

7.3.15.2 Syntax

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

7.3.16 #undef

7.3.16.1 Description

The `#undef` directive undefines a string constant. After a string constant has been undefined, any reference to it generates an error unless the string constant is redefined.

7.3.16.2 Syntax

```
undef-directive:  
    #undef identifier new-line
```

7.3.16.3 Example

```
#define MAX_COUNT 10  
.  
.  
.  
#undef MAX_COUNT  
#define MAX_COUNT 20
```

7.4 Comments

7.4.1 Description

Comments are used to document the meaning and operation of the source code. The compiler ignores all comments. A comment can be placed anywhere in a program where white space can occur. Comments can be many lines long and may also be used to temporarily remove a line of code. ANSI C comments cannot be nested.

7.4.2 Syntax

ANSI C comment:

'/*' begins and '*/' terminates a block comment.

C++ comment:

'//' comments to the end of the line.

7.4.3 Example

```
/* This is a block comment.  
   It can have multiple lines  
   between the comment delimiters.  
*/  
// This is a C++ style one-line comment.
```

MPLAB[®]-CXX Compiler User's Guide

7.5 Constants

7.5.1 Description

A constant in C is any literal number, single character, or character string.

7.5.2 Syntax

7.5.2.1 Numeric Constants

By default, literal numbers are evaluated in decimal. Hexadecimal values can be specified by preceding the number by 0x. Octal values can be specified by preceding the number by 0 (zero). Binary values can be specified by preceding the number by 0b.

7.5.2.2 Character Constants

Character constants are denoted by a single character enclosed by single quotes. ANSI C escape sequences, as shown by the following table, are treated as a single character.

Table 7.1: ANSI C Escape Sequences

Escape Character	Description	Hex Value
\a	Bell (alert) character	07
\b	Backspace character	08
\f	Form feed character	0C
\n	New line character	0A
\r	Carriage return character	0D
\t	Horizontal tab character	09
\v	Vertical tab character	0B
\\	Backslash	5C
\?	Question mark character	3F
\'	Single quote (apostrophe)	27
\"	Double quote character	22
\000	Octal number (zero, Octal digit, Octal digit)	
\xHH	Hexadecimal number	

7.5.2.3 String Constants

String constants are denoted by zero or more characters (including ANSI C escape sequences) enclosed in double quotes. A string constant has an implied null (zero) value after the last character.

7.5.3 Example

7.5.3.1 Numeric Constants

Each of the following evaluates to a decimal twelve:

- 12 Decimal
- 0x0C Hexadecimal
- 014 Octal
- 0b1100 Binary

7.5.3.2 Character Constants

Each of the following is a character constant:

- 'a' Lowercase 'a'
- '\n' New Line
- '\0' Zero or null character

7.5.3.3 String Constants

The following is an example of a string constant:

```
"Hello World"
```

7.6 Variables

This section examines how C uses variables to store data.

The topics discussed in this section are:

- Basic Data Types
- Variable Declaration
- Enumeration
- typedef

7.6.1 Basic Data Types

7.6.1.1 Description

Basic data types are listed in Table 7.2, and allowed modifiers are shown in Table 7.3. Table 7.4 shows the size and range of common data types as implemented by MPLAB-CXX.

C represents all negative numbers in the two's complement format. Integral data types are `char`, `int`, and `long`.

Table 7.2: Basic Data Types

MPLAB-C17	MPLAB-C18	Use
<code>void</code>	<code>void</code>	no type
<code>char</code>	<code>char</code>	single character
<code>int</code>	<code>int</code>	integer value
—	<code>float</code>	floating point value
—	<code>double</code>	floating point value

Table 7.3: Data Type Modifiers

Modifier	Applicable Data Type	Use
<code>auto</code>	any	Variable exists only during the execution of the block in which it was defined.
<code>const</code>	any	Declares data that will not be modified.**
<code>far</code>	any	Paging/banking of data required
<code>extern</code>	any	Declares data that is allocated elsewhere.
<code>long</code>	<code>int</code>	MPLAB-C18: Extended accuracy integer value.

* Not supported by MPLAB-C17

** Does not imply program memory

MPLAB-CXX Fundamentals

Table 7.3: Data Type Modifiers (Continued)

Modifier	Applicable Data Type	Use
near	any	MPLAB-C17: No paging/banking of data required. MPLAB-C18: Data Memory (RAM): Denotes access RAM. MPLAB-C18: Program Memory (ROM): Denotes that the object is located at an address < 64K.
register	any	No effect in MPLAB-CXX
short	int	Declares a short integer.
signed	char, int, long*	Declares a signed variable.
static	any	If the variable is declared outside a function, it can be referenced only within the current file. If the variable is declared inside a function/block, then it can be referenced only within the function/block. The variable can be initialized only with a constant expression which is evaluated at the start of program execution. Values are retained through function calls, and the default initial value is 0.
unsigned	char, int, long*	Declares an unsigned variable.
rom, ram	any	Locate object in program/data memory.

* Not supported by MPLAB-C17

** Does not imply program memory

Table 7.4: Data Type Ranges

Type	Bit Width	Range
void	N/A	none
char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
short	16	-32,768 to 32,767
unsigned short	16	0 to 65,535
short long*	24	-8,388,608 to 8,388,607
unsigned short long*	24	0 to 16,777,215
long*	32	-2,147,483,648 to 2,147,483,647

* Not supported by MPLAB-C17.

MPLAB-C18 Only:
A plain char may be unsigned by default via the -k command line option.

MPLAB[®]-CXX Compiler User's Guide

Table 7.4: Data Type Ranges

Type	Bit Width	Range
unsigned long*	32	0 to 4,294,967,295
float*	32	1.7549435E-38 to 6.80564693E+38
double*	32	1.7549435E-38 to 6.80564693E+38

* Not supported by MPLAB-C17.

7.6.2 Variable Declaration

7.6.2.1 Description

A variable is a name for a specific memory location. In C, all variables must be declared before they are used. A variable's declaration defines its storage class.

Variables can be declared in two places: at the start of a compound statement or outside all functions. The variables are called local and global, respectively.

7.6.2.2 Syntax

```
declaration:
    declaration-specifiers declarator-list ;
declarator-list:
    declarator
    declarator-list , declarator
declaration-specifiers:
    declaration-specifier
    declaration-specifiers declaration-specifier
declaration-specifier:
    type-name
    extern
    static
    ram
    rom
    const
    volatile
    near
    far
type-name:
    basic-type-name
    tag-type-name
basic-type-name:
    int
    short
    char
    unsigned
```

```
long
float
double
tag-type-name:
    enumerated-type-name
    struct-or-union-type-name
```

Local variables (declared inside a compound statement) can only be used by statements within the block where they are declared. The value of a local variable cannot be accessed by functions or statements outside of the function. The most important thing to remember about local variables is that they are created upon entry into the block and destroyed when the block is exited. Local variables must be declared before executable statements.

Global variables can be used by all of the functions in the program. Global variables must be declared before any functions that use them. Most importantly, global variables are not destroyed until the execution of the program is complete.

7.6.2.3 Example

```
unsigned char GlobalCount;
void f2(void)
{
    unsigned char count;
    for(count=0;count<10;count++)
        GlobalCount++;
}
void f1(void)
{
    unsigned char count;
    for(count=0;count<10;count++)
    {
        unsigned char temp;
        f2();
        temp = count *2;
    }
}
void main(void)
{
    GlobalCount = 0;
    f1();
}
```

This program increments GlobalCount to 100. The operation of the program is not affected adversely by the variable named count located in both functions. The variable temp is allocated inside the for() loop and deallocated once the loop exits.

MPLAB[®]-CXX Compiler User's Guide

7.6.3 Storage Class (extern, static, volatile)

7.6.3.1 extern/static

`static` and `extern` behave in the ANSI specified manner. `static` used with a local variable declaration inside of a block causes the variable to maintain its value between entrances to the block. `static` used for a global object (variable or function) declaration outside of all functions limits the scope of the object to the file containing the definition.

`extern` does not allocate space for its object. The compiler assumes the definition appears in an external file. This external reference is resolved at link time.

A global object has external linkage by default.

7.6.3.2 Example1

In file1.c:

```
static unsigned char a;
        unsigned char b;
void main(void)
{
    a = 1;
    b = 2;
    a = new_function();
    return a;
}
```

In file2.c:

```
extern int b;
int new_function(void)
{
    int c;
    c = b;          /* this will not produce an error,
                    because b is extern by default
                    in file1.c and declared extern
                    in file2.c */

    return a;      /* this will produce an undefined
                    variable error because 'a' is
                    only valid within file1.c */
}
```

7.6.3.3 Example2

```
unsigned char hello(void)
{
    static unsigned char i = 0;
    i++;
    return i;
}
```

```
}
void main(void)
{
    unsigned char count;
    for( count = 0; count < 10; count++ )
    {
        unsigned char a;
        a = hello();
    }
}
/* For each call of the function hello, i will be incremented. i is static and will maintain its value between calls to hello. hello is called 10 times, so i will be '10' after the last call. */
```

7.6.3.4 volatile

A volatile variable has a value that can be changed by something other than user code. A typical example is an input port or a timer register. These variables must be declared as 'volatile' so the compiler makes no assumptions on their values while performing optimizations.

7.6.3.5 Example3

```
unsigned char x, y;
volatile unsigned char TMR0;
x = 0x55; /* Compiler's temporary registers
          contain 0x55 */
y = x;    /* and those values can be written to 'y' since
          x is unchanged. */
TMR0 = 0x00;
y = TMR0; /* The compiler must read TMR0 and cannot use
          the 0x00 in its temporary variables since
          TMR0 increments with execution. */
```

7.6.4 Enumeration

7.6.4.1 Description

An enumeration defines a list of named integer constants. The constants defined by an enumeration can be used in the place of any integral value.

7.6.4.2 Syntax

```
enumerated-type-name:
    enum identifier new-line
    enum identifier { enumeration-list } new-line
    enum { enumeration-list } new-line
enumeration-list:
```

MPLAB[®]-CXX Compiler User's Guide

```
enumerated-value
enumeration-list , enumerated-value
enumerated-value:
  identifier
  identifier = constant-expression
```

All enumeration identifiers (such as VALUE_1 in the example) must be unique across all defined enumerations.

Enumerated values can be specified for each enumerated member.

7.6.4.3 Example1

```
enum tag_1 { VALUE_1, VALUE_2, VALUE_3 } enum_1;
/* VALUE_1 is equal to 0 *
 * VALUE_2 is equal to 1 *
 * VALUE_3 is equal to 2 */

char char_1;
enum_1 = 42;      /* this will not produce an error */
char_1 = VALUE_3; /* this will assign char_1 value to 2 */
```

7.6.4.4 Example2

```
enum tag_2 { VALUE_3, VALUE_4, VALUE_5 } enum_2;
/* this definition will cause an error because VALUE_3
already has a value of 2, and cannot also hold a value of
0 */
enum tag_3 { VALUE_6 =2, VALUE_7, VALUE_8=50, VALUE_9 }
enum_3;
/* VALUE_6 is equal to 2 *
 * VALUE_7 is equal to 3 *
 * VALUE_8 is equal to 50 *
 * VALUE_9 is equal to 51 */
enum color_type {red,green,yellow} color;
```

The entries in the enumeration list are assigned constant integer values, starting with zero for the first entry. Each entry is one greater than the previous one. Therefore, in the above example, red is 0, green is 1, and yellow is 2.

The default integer values assigned to the enumeration list can be overridden by specifying a value for a constant. The following example illustrates specifying a value for a constant.

```
enum color_type {red,green=9,yellow} color;
```

This statement assigns 0 to red, 9 to green, and 10 to yellow.

Once an enumeration is defined, the name can be used to create additional variables at other points in the program. For example, the variable mycolor can be created with the color_type enumeration by:

```
enum color_type mycolor;
```


Essentially, enumerations help to document code. Instead of assigning a value to a variable, use an enumeration to clarify the meaning of the value.

7.6.5 typedef

7.6.5.1 Description

The `typedef` statement creates a new name for an existing type. The new name can then be used to declare variables.

Note: Using typedef to Create Portable Programs:

When writing portable code, it is important that the data size be consistent. For example, suppose that 16-bit integers are required. Rather than declaring integers as `int`, declare them as a `typedef` name, such as `myint`. Near the top of the program, declare the `typedef` based on the target machine. When compiling with a tool that uses 16-bit integers, the `typedef` statement should read:

```
typedef int myint;
```

to make all integers declared as `myint` 16-bits.

7.6.5.2 Syntax

The `'typedef'` keyword may be used anywhere the storage class specifiers `'extern'` and `'static'` may be used.

7.6.5.3 Example

```
typedef char string;
typedef unsigned int uint;
void main(void)
{
    string j[10];
    uint i;
    for(i=0;i<10;i++)
        j[i]=i;
}
```

When using a `typedef` statement, remember these two key points:

- A `typedef` does not deactivate the original name or type.
- Several `typedef` statements can be used to create many new names for the same original type.

The `typedef` typically has two purposes:

- Create portable programs
- Document source code

7.7 Functions

Functions are the basic building blocks of the C language. All executable statements must reside within a function.

The topics discussed in this section are:

- Function Declarations
- Function Prototyping
- Passing Arguments to Functions
- Returning Values from Functions

7.7.1 Function Declarations

7.7.1.1 Description

Functions must be declared before they are used. The compiler supports the modern ANSI form of function declarations.

7.7.1.2 Syntax

```
function-definition:  
    function-declarator compound-statement new-line  
function-declarator:  
    declaration-specifiers identifier ( parameter-list )  
parameter-list:  
    parameter  
    parameter parameter-list  
parameter:  
    type-specifier  
    declarator
```

7.7.1.3 Example

```
unsigned char AddOne(unsigned char x)  
{  
    return(x + 1);  
}
```

7.7.2 Function Prototyping

7.7.2.1 Description

A function prototype should be declared before the function is called. A function prototype declares the return type, name, and types of parameters for a function, but no other statements.

7.7.2.2 Syntax

```
function-prototype:  
    function-declarator new-line
```

7.7.2.3 Example

```
unsigned char AddOne(unsigned char x);
```

7.7.3 Passing Arguments to Functions

7.7.3.1 Description

A function argument is a value that is passed to the function when the function is called. C allows zero or more arguments to be passed to a function.

When a function is defined, formal parameters are declared between the parentheses that follow the function name.

Function parameters can have storage class `auto` or `static` (MPLAB-C17 only). `auto` parameters are placed on the software stack, enabling reentrancy, and `static` parameters are allocated globally, enabling direct access and, therefore, smaller code.

For MPLAB-C17 only, 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`.

Note: Overhead of Passing Variables:

MPLAB-CXX 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.

7.7.3.2 Example

The function below calculates the sum of two values that are passed to the function when it is called. When `sum()` is called, the value of each argument is copied into the corresponding parameter variable.

```
void sum( unsigned char a, unsigned char b )  
{  
    int c;  
    c = a+b;  
}  
void main(void)  
{  
    sum(1,10);  
}
```

MPLAB[®]-CXX Compiler User's Guide

```
sum(15,6);  
sum(100,25);  
}
```

Functions pass arguments by value. Any changes made to the formal parameter do not affect the original value in the calling routine.

7.7.4 Returning Values from Functions

7.7.4.1 Description

A function in C can return a value to the calling routine by using the return statement.

For MPLAB-C17, 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.

For MPLAB-C18, the value is always returned on the software stack. On return, *FSR0* points to the return value.

7.7.4.2 Syntax

```
return-statement:  
    return expression new-line  
    return new-line
```

7.7.4.3 Example

```
unsigned char sum(unsigned char a, unsigned char b)  
{  
    return(a + b);  
}  
void main(void)  
{  
    unsigned char c;  
    c = sum(1, 10);  
    c = sum(15, 6);  
    c = sum(100, 25);  
}
```

When a return statement is encountered, the function returns immediately to the calling routine. Any statements after the return are not executed. The return value of a function is not required to be assigned to a variable or to be used in an expression; however, if it is not used, then the value is lost.

7.8 Operators

A C expression is a combination of operators and operands. For the most part, C expressions follow the rules of algebra.

This section discusses many different types of operators including:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Increment and Decrement Operators
- Conditional Operator
- sizeof Operator
- Precedence of Operators

7.8.1 Arithmetic Operators

7.8.1.1 Description

The C language defines five arithmetic operators: addition, subtraction, multiplication, division, and modulus.

7.8.1.2 Syntax

```
arithmetic-expression:  
    postfix-expression  
    arithmetic-expression arithmetic-operator  
                                postfix-expression
```

```
arithmetic-operator:  
    +  addition  
    -  subtraction  
    *  multiplication  
    /  division  
    %  modulus
```

The +, -, *, and / operators may be used with any basic data type.

The modulus operator, %, can only be used with integer data types.

7.8.1.3 Example

```
-b          /* negative b */  
count - 163 /* variable count minus 163 */
```

MPLAB[®]-CXX Compiler User's Guide

7.8.2 Relational Operators

7.8.2.1 Description

The relational operators in C compare two values and return '1' or '0' based on the comparison.

7.8.2.2 Syntax

```
relational-expression:  
    arithmetic-expression  
    relational-expression relational-operator  
                                arithmetic-expression
```

```
relational-operator:  
    >    greater than  
    >=   greater than or equal to  
    <    less than  
    <=   less than or equal to  
    ==   equal to  
    !=   not equal to
```

7.8.2.3 Example

```
count > 0  
value <= MAX  
input != BADVAL
```

7.8.3 Logical Operators

7.8.3.1 Description

The logical operators support the basic logical operations AND, OR, and NOT.

7.8.3.2 Syntax

```
logical-or-expression:  
    logical-and-expression  
    logical-or-expression || logical-and-expression  
logical-and-expression:  
    relational-expression  
    logical-and-expression || relational-expression  
logical-not-expression:  
    !    unary-expression  
    &&   Logical AND  
    ||   Logical OR  
    !    Logical NOT
```

7.8.3.3 Example

```
NotFound && (i <= MAX)
!(Value <= LIMIT)
(('a' <= ch) && (ch <= 'z')) || (('A' <= ch) && (ch <=
'Z'))
```

7.8.4 Bitwise Operators

7.8.4.1 Description

C contains six special operators which perform bit-by-bit operations on numbers. These bitwise operators can only be used on integral data types. The result of using any of these operators is a bitwise operation of the operands.

7.8.4.2 Syntax

```
bitwise-expression:
    postfix-expression
    bitwise-expression bitwise-operator
    postfix-expression

bitwise-not-expression:
    ~ unary-expression

bitwise-operator:
    & bitwise AND
    | bitwise OR
    ^ bitwise XOR
    ~ 1's complement
    >> right shift
    << left shift
```

7.8.4.3 Example

```
Flags & MASK; /* Zero unwanted bits */
Flags ^ 0x07; /* Flip bits 0, 1, and 2 */
Val << 2; /* Multiply Val by 4 */
```

7.8.5 Assignment Operators

7.8.5.1 Description

The most common operation in a program is to assign a value to a variable. C also provides shortcut assignment operators for modifying a variable by performing an operation on itself.

7.8.5.2 Syntax

```
assignment-expression:  
    unary-expression assignment-op expression  
assignment-operator:  
    =  
    +=  
    -=  
    *=  
    /=  
    %=  
    |=  
    ^=  
    >>=  
    <<=
```

7.8.5.3 Example

```
a += b + c;          /* Same as a = a + b + c; */  
a *= b + c;          /* Same as a = a * (b + c); */  
a *= (b + c);        /* Same as a = a * (b + c); */  
r /= s;              /* Same as r = r / s; */  
m *= 5;              /* Same as m = m * 5; */  
Flags |= SETBITS;   /* Set bits in Flags */  
Div2 >>= 1;         /* Divide Div2 by 2 */
```

7.8.6 Increment and Decrement Operators

7.8.6.1 Description

C provides shortcuts for the common operation of incrementing or decrementing a variable. The increment and decrement operators are extremely flexible. They can be used in a statement by themselves, or they can be embedded within a statement with other operators. The position of the operator indicates whether the increment or decrement is to be performed before or after the evaluation of the statement in which it is embedded.

7.8.6.2 Syntax

```
pre-increment-expression:  
    ++ unary-expression  
pre-decrement-expression:  
    -- unary-expression  
post-increment-expression:  
    postfix-expression ++  
post-decrement-expression:  
    postfix-expression --
```


7.8.6.3 Example

```
void main(void)
{
  unsigned char a = 0, b, c;
  a++;                /* same as ++a; */
                    /* a = 1 */
  b = 5 + a++;        /* b = 6, a = 2 */
  c = 6 + --a;        /* c = 7, a = 1 */
}
```

7.8.7 Conditional Operator

7.8.7.1 Description

The conditional operator is a shortcut for executing code based on the evaluation of an expression.

7.8.7.2 Syntax

```
conditional-expression:
  logical-OR-expression ? comma-expression :
                           conditional-expression
```

7.8.7.3 Example

```
c = (a>b) ? a : b; /* c is set to the larger of
                  a and b */
```

7.8.8 sizeof Operator

7.8.8.1 Description

The `sizeof` operator returns the size of the specified item in bytes. The argument to the `sizeof` operator can be a variable, an array name, the name of a basic data type, the name of a derived data type, or an expression.

7.8.8.2 Syntax

```
sizeof-expression:  
    sizeof (type)  
    sizeof a
```

7.8.8.3 Example

```
sizeof (int);    /* number of bytes to store an integer  
                value = 2 */  
sizeof (v);     /* number of bytes to store variable v  
                v is int, value = 2 */  
sizeof (x);     /* number of bytes to store all variables  
                of array x  
                x is int, 10 elements, value = 20 */  
  
sizeof (struct data); /* number of bytes to store  
                       expression  
                       struct has 3 int members,  
                       value = 6 */
```


7.9 Program Control Statements

This section describes the statements that C uses to control the flow of execution in a program, explains how relational and logical operators are used with these control statements, and covers how to execute loops.

Topics discussed in this section include:

- `if` Statement
- `if-else` Statements
- `for` Loop
- `while` Loop
- `do-while` Loop
- `switch` Statement
- `break` Statement
- `continue` Statement
- `goto` Statement

7.9.1 `if` Statement

7.9.1.1 Description

The `if` statement is a conditional statement. The statement associated with the `if` statement is executed based upon the outcome of a condition. If the condition evaluates to nonzero, the statement is executed. Otherwise, it is skipped.

7.9.1.2 Syntax

```
if-statement:  
if ( expression ) statement new-line
```

7.9.1.3 Example

```
if(num > 0) Adjust(num);  
if(count<0)  
{  
    count=0;  
    EndFound = TRUE;  
}
```

7.9.2 if-else Statements

7.9.2.1 Description

The `if-else` statement handles conditions where a program requires one statement to be executed if a condition is nonzero and a different statement if the condition is zero.

7.9.2.2 Syntax

```
if-else-statement:  
    if ( expression ) statement else statement new-line
```

7.9.2.3 Example

```
if(num < 0)  
{  
    num = 0;  
    Valid = 0;  
}  
else  
    Valid = 1;  
if(num == 1)  
    DoCase1();  
else if(num == 2)  
    DoCase2();  
else if(num == 3)  
    DoCase3();  
else  
    DoInvalid();
```

7.9.3 for Statement

7.9.3.1 Description

One of the three loop statements that C provides is the `for` loop. The other two are the `if` and the `do-while` statements.

Use a `for` loop to repeat a statement or set of statements. In general, the first expression is the initialization expression, the second is the loop condition, and the third is the loop expression.

7.9.3.2 Syntax

```
for-statement:  
    for ( expression ; expression ; expression ) statement  
        new-line
```

7.9.3.3 Example

```
unsigned char i;
for(i=0;i<10;i++)
    DoFunc();
for(num=100;num>0;num=num-1)
    { . . . }
for(count=0;count<50;count+=5)
    { . . . }
/* Find Target */
for(i=0; (i<MAX) && (Array[i]<>Target); i++);
```

7.9.4 while Statement

7.9.4.1 Description

Another of the loops in C is the `while` loop. While an expression is nonzero, the `while` loop repeats a statement or block of code. The value of the expression is checked prior to each execution of the statement.

7.9.4.2 Syntax

```
while-statement:
    while ( expression ) statement new-line
```

7.9.4.3 Example

```
X = GetValue()
while (1); /* Loop Forever */
{
    HandleValue(X);
    X = GetValue();
}
```

7.9.5 do-while Statement

7.9.5.1 Description

The final loop in C is the `do` loop. In the `do` loop, the statement is always executed before the expression is evaluated. Thus, the `do` statement always executes at least once. If the `while` expression evaluates to nonzero, control is transferred back to the beginning of the loop.

7.9.5.2 Syntax

```
if-statement:
    do statement while ( expression ) new-line
```

7.9.5.3 Example

```
do
{
  x = GetValue();
  HandleValue(x);
} while (x != 0);
```

7.9.6 switch Statement

7.9.6.1 Description

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 (MPLAB-C17 only).
- The `case` labels must be constant values.

7.9.6.2 Syntax

```
switch-statement:
    switch ( expression ) statement new-line
case-statement:
    case constant-expression : statement new-line
default-statement:
    default : statement new-line
```

The use of the `default` label is good programming practice. It can catch out-of-range data that is not expected.

The `switch` expression is successively tested against a list of constants. When a match is found, execution continues at the labeled `case` statement. If no match is found, the statements associated with the `default` case are executed if a `default` label exists.

7.9.6.3 Example

```
switch(i)
{
  case 1:
    DoCase1();
    break;
  case 2:
    DoCase2();
    break;
  case 3:
    DoCase3();
    break;
  case 4:
```

MPLAB[®]-CXX Compiler User's Guide

```
        DoCase4();
        break;
default:
    DoDefault();
    break;
}
x = 0;
switch(ch)
{
    case 'c':           /* Ignoring case, set x to: */
    case 'C': x++;      /* 1 if ch is A */
    case 'b':           /* 2 if ch is B */
    case 'B': x++;      /* 3 if ch is C */
    case 'a':           /* otherwise, ch is invalid */
    case 'A': x++
        break;
    default :
        BadChar(ch);
        break;
}
```

7.9.7 break Statement

7.9.7.1 Description

The `break` statement exits the innermost enclosing control statement (`for`, `while`, `do`, `switch`) from any point within the body. The `break` statement bypasses normal termination from an expression. If the `break` occurs in a nested loop, control returns to the previous nesting level.

7.9.7.2 Syntax

```
break-statement:
    break new-line
```

7.9.7.3 Example

```
/* Get 100 values. Stop immediately if the value is 0. */
unsigned char i;
for(i = 0; i < 100; i++)
{
    x = GetValue();
    if(x == 0)
        break;
    HandleValue(x);
}
```


7.9.8 continue Statement

7.9.8.1 Description

The `continue` statement allows a program to skip to the end of a `for`, `while`, or `do` statement without exiting the loop.

7.9.8.2 Syntax

```
continue-statement:  
    continue new-line
```

7.9.8.3 Example

```
/* Get 100 values. If the value is 0,  
   ignore it and go on. */  
unsigned char i;  
for (i = 0; i < 100; i++)  
{  
    x = GetValue;  
    if (x == 0)  
        continue;  
    HandleValue(x);  
}
```

7.9.9 goto Statement

7.9.9.1 Description

Execution of a `goto` causes control to be sent directly to the labeled statement. This statement must be located in the same function as the `goto`.

Use of `goto`'s interrupts the normal sequential flow of a program and thus makes it harder to follow and decipher. For this reason, the use of `goto`'s is not considered good programming style, i.e., it is recommended that you do not use them in your program.

7.9.9.2 Syntax

```
goto-statement:  
    goto label new-line
```

7.9.9.3 Example

```
/* Branch on error */  
goto fatal_error;  
:  
fatal_error: error_fn ( "fatal error" );  
:
```

Instead of jumping to the error function, call the function where the error occurred to avoid using a `goto`.

7.10 Arrays and Strings

An array is a list of related variables of the same data type. Strings are arrays of characters with some special rules.

Topics discussed in this section include:

- Arrays
- Strings
- Initializing Arrays

7.10.1 Arrays

7.10.1.1 Description

An array is a list of elements which are all of the same type and can be referenced through the same name. When an array is declared, C defines the first element to be at an index of 0; therefore, if the array has 50 elements, the last element is at an index of 49.

C stores arrays in contiguous memory locations. The first element is at the lowest address.

7.10.1.2 Syntax

```
declarator:  
    declarator array-declarator  
array-declarator:  
    [ constant-expression ]  
    array-declarator [ constant-expression ]
```

7.10.1.3 Example

```
#define SIZE 10  
unsigned char i, num[SIZE];  
for(i = 0; i < SIZE; i++)  
    num[i] = i;
```

To copy the contents of one array into another, copy each individual element from the first array into the second array. The following example shows one method of copying the array `a[]` into `b[]` assuming that each array has 10 elements.

```
for(i=0;i<10;i++)  
    b[i] = a[i];
```

7.10.2 Strings

7.10.2.1 Description

A common one-dimensional array is the string. C does not have a built-in string data type. Instead, a string is defined as a null (0) terminated character array. The size of the character array must include the terminating null. All string constants are automatically null terminated.

7.10.2.2 Example

```
char String[80];
int i;
.
.
.
for(i = 0; (i < 80) && String[i]; i++)
    HandleChar(String[i]);
```

7.10.3 Initializing Arrays

7.10.3.1 Description

C allows initialization of arrays. Standard data type arrays may be initialized in a straight-forward manor. Initializing arrays using the PICmicro MCU `ram` and `rom` qualifiers may require more than a single-line initialization statement.

7.10.3.2 Syntax

```
initialized-declarator:
    declarator = { value-list }
value-list:
    { value-list }
    constant-expression-list
constant-expression-list:
    constant-expression
    constant-expression-list , constant-expression
```

7.10.3.3 Example1

The following example shows a 5 element integer array initialization.

```
int i[5] = {1,2,3,4,5};
```

The element `i[0]` has a value of 1 and the element `i[4]` has a value of 5.

A string (character array) can be initialized in two ways. One method is to make a list of each individual character:

```
char str[4]={ 'a', 'b', 'c', 0};
```

MPLAB[®]-CXX Compiler User's Guide

The second method is to use a string constant:

```
char name[5]="John";
```

A null is automatically appended at the end of "John". When initializing an entire array, the array size may be omitted:

```
char Version[] = "V1.0";
```

7.10.3.4 Example2

Because the PICmicro MCU family of microcontrollers uses separate program memory and data memory address busses in their design, MPLAB-CXX 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-CXX 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-CXX automatically places all string literals in ROM. This type of a string literal is "array of char located in ROM."

For example, when using MPLAB-C18, a string table in ROM 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" };
```

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

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 9 bytes long, and the array is four elements long, so `table2` takes $(9*4+4*2)/2 = 22$ 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-CXX 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 PICXXC756 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.

Method 1: Copy the ROM string to a RAM buffer before sending

```
rom char mystring[] = "Send me to the USART";
void foo( void )
{
    char strbuffer[21];
    str2ram( strbuffer, mystring );
    putsUSART1( strbuffer );
}
```

Method 2: 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 */
```

7.11 Pointers

This section covers one of the most important and powerful features of C, pointers. A pointer is a variable that contains the location of an object.

The topics covered in this section are:

- Introduction to Pointers
- Pointers and Arrays
- Pointer Arithmetic
- Passing Pointers to Functions

MPLAB[®]-CXX Compiler User's Guide

7.11.1 Introduction to Pointers

7.11.1.1 Description

A pointer is an object that holds the location of another object or a NULL constant.

For example, if a pointer variable called Var1 contains the address of a variable called Var2, then Var1 points to Var2. If Var2 is a variable at address 100 in memory, then Var1 would contain the value 100.

For MPLAB-C17:

RAM pointers are 16-bit values. ROM pointers are 24-bit values if they point to 8-bit objects. ROM pointers are 16-bit values if they point to objects 16-bits or greater.

For MPLAB-C18:

RAM pointers are either 8-bit (near) or 16-bit (far) values. ROM pointers are either 16-bit (near) or 24-bit (far) values.

7.11.1.2 Syntax

declarator:

```
* type-qualifier-list declarator
```

The two special operators that are associated with pointers are the asterisk (*) and the ampersand (&). The address of a variable can be accessed by preceding the variable with the & operator. The * operator returns the value stored at the address pointed to by the variable.

7.11.1.3 Example

```
void main(void)
{
    unsigned char *Var1, Var2, Var3;
    Var2    = 6;
    Var1    = &Var2;
    Var3    = Var2;          /* These two do */
    Var3    = *Var1;        /* the same thing. */
}
```

The first statement declares three variables: Var1, which is an integer pointer, and Var2 and Var3, which are integers. The next statement assigns the value of 6 to Var2. Then the address of Var2 (&Var2) is assigned to the pointer variable Var1. Finally, the value of Var2 is assigned to Var3 in two ways: first by accessing Var2 directly, then by accessing Var2 through the pointer Var1.

7.11.2 Pointer Arithmetic

7.11.2.1 Description

In general, pointers may be treated like other variables. However, there are a few rules and exceptions. In addition to the * and & operators, there are only four other operators that can be applied to pointer variables: +, ++, -, --.

An important point to remember when performing pointer arithmetic is that the value of the pointer is adjusted according to the size of the data type it is pointing to. If a pointer's data type requires five bytes, incrementing the pointer actually increases the value of the pointer by five. Similarly, adding three to the pointer increases the value of the pointer by fifteen (three times five).

Note: ROM and RAM pointers in MPLAB-CXX:

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) */
```

7.11.2.2 Example

```
unsigned char *p, *q, r[30] ;
.
.
p = r + 20; /* p points to element 20 of r */
q = p - 5   /* q points to element 15 of r */
p++;       /* p points to element 21 of r */
```

It is possible to increment or decrement either the pointer itself or the object to which it points. Pointers may also be used in relational operations.

7.11.3 Passing Pointers to Functions

7.11.3.1 Description

A pointer may be passed to a function just like any other value.

7.11.3.2 Example

```
void inby10(unsigned char *n)
{
    *n += 10;
}
```

MPLAB[®]-CXX Compiler User's Guide

```
void main(void)
{
    unsigned char *p;
    unsigned char i = 0;
    p=&i;
    incby10(p);      /* i equals 10 */
    incby10(&i);    /* i equals 20 */
}
```

7.12 Structures and Unions

Structures are a group of related variables. Unions are a group of variables, often of differing types, that share the same memory space.

This section covers:

- Introduction to Structures
- Introduction to Unions
- Nesting Structures
- Bit-fields

7.12.1 Syntax

```
struct-or-union-type-name:
    struct-or-union identifier
    struct-or-union identifier
        { member-declaration-list }
    struct-or-union { member-declaration-list }
member-declaration-list:
    member-declaration
    member-declaration-list member-declaration
member-declaration:
    member-declaration-specifiers declarator-list ;
member-declaration-specifiers:
    member-declaration-specifier
    member-declaration-specifiers member-declaration-
    specifier
member-declaration-specifier:
    type-name
    const
    volatile
    near
    far
```


7.12.2 Introduction to Structures

Structures and Debugging in MPLAB IDE

User-defined data constructs are included in the symbolic information file from the linker, but MPLAB IDE doesn't use them in debugging.

Structures and unions allow the storage and manipulation of related data together rather than in separate variables.

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

MPLAB-CXX supports anonymous structures.

7.12.2.1 Description

A structure is a group of related items that can be accessed through a common name. Each item within a structure has its own data type, which can be different from the other data types.

7.12.2.2 Example

The following example is for a card catalog in a library.

```
struct catalog_tag
{
  char author[40];
  char title[40];
  char pub[40];
  unsigned int date;
  unsigned char rev;
} card;
```

In this example, the tag of the structure is catalog. It is not the name of a variable, only the name of the type of structure. The variable card is declared as a structure of type catalog. The following shows what the structure catalog looks like in memory.

author	40 bytes
title	40 bytes
pub	40 bytes
date	2 bytes
rev	1 byte

To access any member of a structure, specify the name of the variable and the name of the member separated by a period. For example, to change the revision member of the structure catalog, use the following:

```
card.rev='a';
```

To access the third character in the title, use the following:

```
ThirdChar = card.title[2];
```

MPLAB[®]-CXX Compiler User's Guide

7.12.3 Introduction to Unions

7.12.3.1 Description

A union is a memory block that is shared by two or more variables, which can be of any data type. A union resembles a structure, but its memory usage is fundamentally different. In a structure, the elements are arranged sequentially. In a union, all of the elements begin at the same address, making the size of the union equal to the size of the largest element.

7.12.3.2 Syntax

The <union-name> is the tag of the union, and the <variable-list> contains the names of the variables that have a data type of <union-name>.

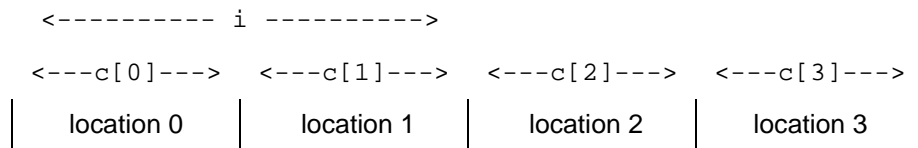
Accessing members of a union is the same as accessing members of a structure.

7.12.3.3 Example

Because an `int` is two bytes and a `char` is one byte, the union below is stored in memory as shown:

```
union u_tag
{
  int i;
  char c[4];
} temp;
```

where:



An example of saving space is shown below:

```
struct type_tag
{
  enum { VARIABLE, CONSTANT } type;
  union
  {
    char *variable_name;
    int constant_value;
  } value;
} variable_or_constant;
void function( struct type_tag var_or_const )
{
  int constant;
  char *variable;
```

```
switch( var_or_const.type )
{
    case VARIABLE:
        variable = var_or_const.value.variable_name;
        break;
    case CONSTANT:
        constant = var_or_const.value.constant_value;
        break;
}
```

Based on the type of data stored in `struct type_tag`, the access of the data is different. A union allows the data for the two types to share space.

7.12.4 Nesting Structures

7.12.4.1 Description

A structure member can have a data type that is another structure. This is referred to as a nested structure.

7.12.4.2 Example1

```
struct Memory
{
    int RAMSize;
    int ROMSize;
};
struct PIC
{
    char Name[12];
    struct Memory MemSizes;
};
```

Members of a structure or union define a separate name space, Meaning that two different structures can have the same names for their members.

7.12.4.3 Example2

```
struct struct_tag_1{
    int a;
    int b;
    char c;
} struct_1;
struct struct_tag_2
{
    char d;
    int a;
    int b;
} struct_2;
```

MPLAB[®]-CXX Compiler User's Guide

`struct_1.a` references the first two bytes of a structure of type `struct tag_1`.

`struct_2.a` references the second and third bytes of a structure of type `struct tag_2`.

`struct_2.c` and `struct_1.d` would produce an error because the referenced member is not part of the structure's definition.

7.12.5 Bit-fields

7.12.5.1 Description

Bit-fields allow the specification of integer-type members of a structure, 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.

7.12.5.2 Syntax

```
struct <struct_name>
{
<int type> <member1> : <bit-width>;
<int type> <member2> : <bit-width>;
:
<int type> <membern> : <bit-width>;
}
```

7.12.5.3 Example

See Section 8.4.3.

Chapter 8. MPLAB-CXX and PICmicro MCU Programming

8.1 Introduction

This section discusses specific details for programming PICmicro microcontrollers (MCUs) when using MPLAB-C17 and MPLAB-C18.

8.2 Highlights

Items discuss in this chapter are:

- PICmicro MCU Programming Specifics
- MPLAB-C17 and PICmicro MCU Programming
- MPLAB-C18 and PICmicro MCU Programming

8.3 PICmicro MCU Programming Specifics

When using C to program a PICmicro MCU device, the architecture and operation of the device must be considered. That is, the following items should be understood to write C code for a PICmicro MCU:

- Memory Models
- Processor Header File
- Register Definitions File
- Software Stack
- Startup and Initialization
- Interrupt Support
- Internal Assembler

8.3.1 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-CXX Reference Guide* for a list of libraries and precompiled object files available for different memory models.

8.3.2 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 (Section 8.3.3).

In addition to register declarations, the header file defines in-line assembly macros (Table 8.1) and interrupt install macros (MPLAB-C17 only).

MPLAB[®]-CXX Compiler User's Guide

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 8.1: Instruction Macro Actions

Instruction Macro	Action
<code>Nop()</code>	Executes a no operation (NOP)
<code>ClrWdt()</code>	Clears the watchdog timer (CLRWDT).
<code>Sleep()</code>	Executes a SLEEP instruction
<code>Reset()</code>	Executes a device reset (RESET)
<code>Rlcf(var)</code>	Rotates 'var' to the left through the carry bit
<code>Rlncf(var)</code>	Rotates 'var' to the left without going through the carry bit.
<code>Rrcf(var)</code>	Rotates 'var' to the right through the carry bit.
<code>Rrncf(var)</code>	Rotates 'var' to the right without going through the carry bit.
<code>Swapf(var)</code>	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.

8.3.3 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 (Section 8.3.2) 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.

MPLAB-CXX and PICmicro MCU Programming

8.3.4 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.

8.3.5 Startup and Initialization

A Startup file, written in either assembly or C, performs the following tasks:

1. Optionally calls the function `__STARTUP()` upon reset (MPLAB-C17 Only).
2. Optionally calls the code which copies initialized data from program memory to data memory.
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.

8.3.6 Interrupt Support

Support for interrupts is provided differently for the different compilers. In MPLAB-C17, interrupts are handled by interrupt support macros. In MPLAB-C18, interrupts are handled by the `#pragma interrupt` directive.

MPLAB[®]-CXX Compiler User's Guide

8.3.7 Internal Assembler

MPLAB-CXX has an internal assembler using a syntax similar to MPASM. 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 MPLINK.

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
- No default operators

8.4 MPLAB-C17 and PICmicro MCU Programming

This section discusses issues of using MPLAB-C17 for PICmicro MCU programming.

8.4.1 Memory Models

For PIC17CXXX devices with program memory (ROM) over 8K, the memory is broken up in to pages. For PIC17CXXX devices with data memory (RAM) over 256, the memory is broken up into banks. Table 8.2 spells out the memory models available for these devices via MPLAB-C17.

Table 8.2: 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.

MPLAB-CXX and PICmicro MCU Programming

8.4.2 Processor Header File

Processor header files for PIC17CXXX devices contain external declarations for the special function registers, in-line assembly macros and interrupt install macros.

PIC17CXXX 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 the Section 8.4.6. Interrupt support macros are listed in the following table:

Table 8.3: Macro Actions

Macro	Action
<code>Install_INT(<i>func</i>)</code>	Sets ' <i>func</i> ' as the handler for the INT interrupt.
<code>Install_TMR0(<i>func</i>)</code>	Sets ' <i>func</i> ' as the handler for the TMR0 interrupt.
<code>Install_T0CKI(<i>func</i>)</code>	Sets ' <i>func</i> ' as the handler for the T0CKI interrupt.
<code>Install_PIV(<i>func</i>)</code>	Sets ' <i>func</i> ' as the handler for the PIV interrupt.

8.4.3 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 (see previous section) that contains, among other things, external declarations for the special function registers.

Example 8.2: 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.

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;
```

MPLAB[®]-CXX Compiler User's Guide

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 TCKI:1;     /* Alternate name for bit 1 */
    unsigned :6;         /* pad next 6 locations */
  };
} 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.

8.4.4 Software Stack

Define a software stack in the linker script for the processor by using a command similar to the following:

```
STACK SIZE = 0x20
```

MPLAB-CXX and PICmicro MCU Programming

Stack Overflow

Avoidance: For MPSIM 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.

This reserves 32 bytes in the general purpose RAM area for the software stack. 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.

8.4.5 Startup and Initialization

The Startup file for PIC17CXXX 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 PIC17CXXX 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.

8.4.5.1 `__STARTUP()`

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.)

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;
}
```

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

```
#DEFINE USE_STARTUP
```

MPLAB[®]-CXX Compiler User's Guide

3. Compile the source file, assemble `c0117.asm` or `c0s17.asm` and link.

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

8.4.5.2 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 `c0117.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.

8.4.5.3 Stack Initialization

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

8.4.5.4 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
```

`c0s17.asm` is assembled with `USE_INITDATA` undefined by default.
`c0117.asm` is assembled with `USE_INITDATA` defined by default.

MPLAB-CXX and PICmicro MCU Programming

8.4.6 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`.

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.)

Startup code supplied with MPLAB-C17 does not support nested interrupts.

Table 8.4: Interrupt Example

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

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()
{
```

MPLAB[®]-CXX Compiler User's Guide

```
x++;
PORTB = x;
}
void main()
{
    x = 1;
    // Install interrupt handler for timer 0 interrupt
    Install_TMR0(_TMR0);
    // Set prescale value for TMR0
    TOSTA = 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
```

MPLAB-CXX and PICmicro MCU Programming

```
    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
    TOSTA = 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.

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.

8.4.7 Optimization Tips

Because of the limited memory on microcontrollers, optimization becomes an issue as code complexity increases. This section discusses some optimization tips for your C code.

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 Section 8.4.1 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.

Example 8.3: Minimizing Bank Switching – MPLAB-C17

In the linker script:

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

In the program:

```
#pragma varlocate coeff
```

3. Use of section pragma's to effectively manage RAM and ROM. Refer to Section 7.3 for information on the pragma directive. For examples of use, see the MPLINK examples found in the *MPASM User's Guide with MPLINK and MPLIB*.

MPLAB-CXX and PICmicro MCU Programming

8.5 MPLAB-C18 and PICmicro MCU Programming

This section discusses issues of using MPLAB-C18 for PICmicro MCU programming.

8.5.1 Memory Models

PIC18CXXX devices using more than 32K program memory (ROM), i.e., external ROM or future devices with more ROM, will need to use the large memory model. Table 8.5 spells out the memory models available for these devices via MPLAB-C18.

Table 8.5: Memory Model Usage - MPLAB-C18

Memory Model		Device Description
s	small	near rom – program memory \leq 64KB (16-Bit pointer used)
l	large	far rom – program memory $>$ 64KB (24-Bit pointer used)

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

8.5.2 Processor Header File

Processor header files for PIC18CXXX devices contain external declarations for the special function registers and in-line assembly macros, but no interrupt install macros. In MPLAB-C18, interrupts are handled by the `#pragma interrupt` directive. Please see Section 7.3.12 for more information.

8.5.3 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 (see previous section) that contains, among other things, external declarations for the special function registers.

8.5.4 Software Stack

Define a software stack in the linker script for the processor by using a command similar to the following:

```
STACK SIZE = 0x100
```

This reserves 256 bytes in the general purpose RAM area for the software stack. The size of the software stack required by a program varies with the complexity of the program. The following should be kept in mind:

- 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.

Stack Overflow

Avoidance: For MPSIM 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.

MPLAB[®]-CXX Compiler User's Guide

Example 8.4: Stack Operation – MPLAB-C18

The first instruction saves the current frame pointer onto the stack. The next instructions sets up the stack frame by copying the stack pointer (*FSR1*) to the frame pointer (*FSR2*). By default, the compiler assumes that the stack is contained within a single bank, and so only copies the low byte. Thus positive offsets from the frame pointer reference local variables and negative offsets reference parameter values.

```
000128   cfd9      MOVFF      0xfd9,0xfe6 void main( void )
00012a   ffe6
00012c   cfe1      MOVFF      0xfe1,0xfd9
00012e   ffd9
```

The next instructions add to the stack pointer the number of bytes of local variable storage required by the function which is, in this case, two bytes.

```
000130   0e02      MOVLW      0x2
000132   26e1      ADDWF      0xe1,0x1,0x0
                                     {
                                     unsigned char a, b;
```

Here's some code in the middle just so there is a function body.

```
000134   cfd9      MOVFF      0xfd9,0xfe9      do{
000136   ffe9                                     a=0;
000138   cfda      MOVFF      0xda,0xfea
00013a   ffea
00013c   6aef      CLRF      0xef,0x0
                                     }while(0);
```

The epilogue code subtracts from the stack pointer the number of bytes which were allocated by the prologue, plus one for the frame pointer. Then it uses *INDF1* to retrieve the prior value of the frame pointer back into *FSR2*.

```
00013e   0e03      MOVLW      0x3
000140   5ee1      SUBWF      0xe1,0x1,0x0
000142   cfe7      MOVFF      0xfe7,0xfd9
000144   ffd9
000146   0012      RETURN    0x0
```

8.5.5 Startup and Initialization Code

The Startup file for PIC18CXXX devices is a C file that is included in the default standard C library, `clib.lib`. This file performs these main tasks:

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

MPLAB-CXX and PICmicro MCU Programming

There are two startup code files for the PIC18CXXX family. The first is `c018i.c` which is used with initialized data (idata). The other is `c018.c`, which is used without idata. There is no extra idata file for PIC18CXXX devices.

8.5.5.1 Startup Customization

To execute some code immediately after a device reset but before any other code generated by the compiler is executed, edit the desired startup file and add the code before the `_entry()` function.

To customize the startup files:

1. Go to the `c:\mcc\src\startup` directory, where `c:\mcc` is the compiler install directory.
2. Edit either `c018.asm` or `c018i.asm` to add any customized startup code desired.
3. Compile the updated startup file to generate either `c018.o` or `c018i.o`.

8.5.5.2 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.

To use initialized data in a MPLAB-C18 program:

1. Use the standard C library `clib.lib` instead of `c_noinit.lib`.
2. Link the above file with the C object code.

8.5.5.3 Stack Initialization

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

8.5.5.4 Branching to main()

After the startup code optionally 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-C18 transfers control to `main()` via a looped call, i.e.;

```
loop:

// Call the user's main routine
main();

goto loop;
```

8.5.6 Interrupt Support

In MPLAB-C18, interrupts are handled by the `#pragma interrupt` directive. Please see Section 7.3.12 for more information.

8.5.7 Optimization Tips

Because of the limited memory on microcontrollers, optimization becomes an issue as code complexity increases. This section discusses some optimization tips for your C code.

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 Section 8.5.1 for more information.

Chapter 9. Mixing Assembly Language and C Modules

9.1 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.

9.2 Highlights

This chapter covers the following topics:

- Calling Conventions
- Mixing Assembly Language and C Variables and Functions
- Calling an Assembly Function in C – MPLAB-C17
- Using the File Selection Registers (FSR's)

9.3 Calling Conventions

This section describes the calling conventions of C and assembly routines.

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.

9.4 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
```

Mixing Assembly Language and C Modules

9.5 Calling an Assembly Function in C – MPLAB-C17

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( 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
```

9.6 Using the File Selection Registers (FSR's)

The following is an example of using the file selection registers (FSR's). The code is for the library function `strcpy`, which copies the source string (pointed to by `src`) into the destination string (pointed to by `dst`). See the *MPLAB-CXX Reference Guide* for more on the `strcpy` operation.

Example 9.1: FSR Usage

```
extern FSR2L, FSR2H, POSTINC1, POSTDEC1, PLUSW1, FSR0L, FSR0H, POSTINC0
extern POSTINC2, INDF0, INDF1, FSR1L, FSR1H

global strcpy

STRCPY CODE
; char *strcpy (char *dst, const char *src);
strcpy
; We'll play loose with the stack and frame pointers in this function.
; As long as the values are restored upon exit and the stack pointer
; always points into unallocated stack space, things will be fine,
; even if an interrupt occurs during our processing here.
;
; Save the current FSR2 value on the stack. We'll be using it as
; a source index.
movff FSR2L, POSTINC1
movff FSR2H, POSTINC1
; Store the 'src' pointer into FSR2
movlw -6
movff PLUSW1, FSR2L
movlw -5
movff PLUSW1, FSR2H
; Store the 'dst' pointer into FSR0
movlw -4
movff PLUSW1, FSR0L
movlw -3
movff PLUSW1, FSR0H
; Perform the copy...
copyloop:
movff POSTINC2, INDF0
; Was that the '\0'?
tstfsz POSTINC0,0
bra copyloop
; restore FSR2
movf POSTDEC1,1,0
movff POSTDEC1,FSR2H
movff INDF1,FSR2L
; The return value is a pointer to the destination
movlw -4
addwf FSR1L,0,0
```


Mixing Assembly Language and C Modules

```
movwf FSR0L,0
movlw 0xff
addwfc FSR1H,0,0
movwf FSR0H
return

end
```

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Chapter 10. ANSI Implementation Issues

10.1 Introduction

This section describes the behavior of MPLAB-CXX 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.

10.2 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

10.3 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-CXX identifiers have 31 significant characters. Case distinctions are significant in an identifier with external linkage.

10.4 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`.

For MPLAB-C18, this may be changed to `unsigned char` via a command line switch (`-k`).

10.5 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)

MPLAB-C18 does not enforce this by default. The `-oi` option can be used to require the compiler to enforce the ANSI defined behavior.

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.

10.6 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)

No floating point types are supported in MPLAB-C17 at this time.

32-bit floating point types are native to MPLAB-C18. The difference between the PICmicro format and the IEEE 754 format consists of a rotation of the top nine bits of the representation, with a left rotate for IEEE to PICmicro, and a right rotate for PICmicro to IEEE. Conversion to a 24-bit format is obtained by the rounding to the nearest from IEEE 754 representation. The limiting absolute values of the floating point formats are: 6.80564693E+38 max and 1.17549435E-38 min.

The rounding to the nearest method is used.

10.7 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` (MPLAB-C17) or unsigned short `int` (MPLAB-C18).

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` (MPLAB-C17) or unsigned long short (MPLAB-C18).

10.8 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.

10.9 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.

10.10 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 (MPLAB-C17) or a signed int bit-field (MPLAB-C18).

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.

10.11 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.

10.12 Switch Statement

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.

10.13 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 Chapter 6.

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Chapter 11. Examples

11.1 Introduction

This chapter gives an overview of the example programs included with the compiler program and support files.

At this time, only MPLAB-C17 has example files.

11.2 Highlights

The contents of this chapter are as following:

- Overview of Example Files
- Example Details

11.3 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`
- Demo
 - `DEMO`

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.

11.4 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 (Chapter 5).

<p>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 (<code>c:\mcc\src</code>).</p>
--



Appendices

Appendix A. ASCII Character Set.....	155
Appendix B. PIC17CXXX Instruction Set.....	157
Appendix C. PIC18CXXX Instruction Set.....	163
Appendix D. MPLAB-C17 Errors	169
Appendix E. MPLAB-C18 Errors	175
Appendix F. References.....	185

MPLAB[®]-CXX Compiler User's Guide

Appendix A. ASCII Character Set

A.1 Introduction

This appendix contains the ASCII character set.

A.2 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

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Appendix B. PIC17CXXX Instruction Set

B.1 Introduction

This appendix gives the instruction set for the PIC17CXXX device family.

B.2 Highlights

This appendix presents the following reference information:

- Key to PICmicro MCU Family Instruction Sets
- PIC17CXXX Instruction Set

B.3 Key to PICmicro MCU Family Instruction Sets

Field	Description
b	Bit address within an 8 bit file register
d	Destination select; d = 0 Store result in W (f0A). d = 1 Store result in file register f. Default is d = 1.
f	Register file address (0x00 to 0xFF)
k	Literal field, constant data or label
W	Working register (accumulator)
x	Don't care location
i	Table pointer control; i = 0 Do not change. i = 1 Increment after instruction execution.
p	Peripheral register file address (0x00 to 0x1f)
t	Table byte select; t = 0 Perform operation on lower byte. t = 1 Perform operation on upper byte.
PH:PL	Multiplication results registers

MPLAB[®]-CXX Compiler User's Guide

B.4 PIC17CXXX Instruction Set

Microchip's high-performance 8-bit microcontroller family uses a 16-bit wide instruction set. This instruction set consists of 55 instructions, each a single 16-bit wide word. Most instructions operate on a file register, *f*, and the working register, *W* (accumulator). The result can be directed either to the file register or the *W* register or to both in the case of some instructions. Some devices in this family also include hardware multiply instructions. A few instructions operate solely on a file register (BSF for example).

Table B.1: 16-Bit Core Data Movement Instructions

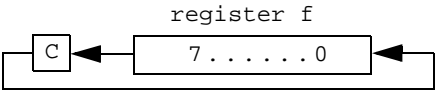
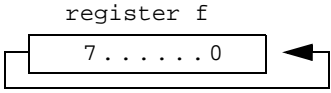
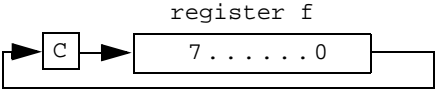
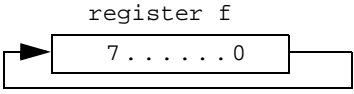
Hex	Mnemonic	Description	Function
6pff	MOVFP <i>f, p</i>	Move <i>f</i> to <i>p</i>	<i>f</i> → <i>p</i>
b8kk	MOVLB <i>k</i>	Move literal to BSR	<i>k</i> → BSR (3:0)
bakx	MOVLP <i>k</i>	Move literal to RAM page select	<i>k</i> → BSR (7:4)
4pff	MOVPF <i>p, f</i>	Move <i>p</i> to <i>f</i>	<i>p</i> → <i>W</i>
01ff	MOVWF <i>f</i>	Move <i>W</i> to <i>F</i>	<i>W</i> → <i>f</i>
a8ff	TABLRD <i>t, i, f</i>	Read data from table latch into file <i>f</i> , then update table latch with 16-bit contents of memory location addressed by table pointer	TBLATH → <i>f</i> if <i>t</i> =1, TBLATL → <i>f</i> if <i>t</i> =0; ProgMem(TBLPTR) → TBLAT; TBLPTR + 1 → TBLPTR if <i>i</i> =1
acff	TABLWT <i>t, i, f</i>	Write data from file <i>f</i> to table latch and then write 16-bit table latch to program memory location addressed by table pointer	<i>f</i> → TBLATH if <i>t</i> = 1, <i>f</i> → TBLATL if <i>t</i> = 0; TBLAT → ProgMem(TBLPTR); TBLPTR + 1 → TBLPTR if <i>i</i> =1
a0ff	TLRD <i>t, f</i>	Read data from table latch into file <i>f</i> (table latch unchanged)	TBLATH → <i>f</i> if <i>t</i> = 1 TBLATL → <i>f</i> if <i>t</i> = 0
a4ff	TLWT <i>t, f</i>	Write data from file <i>f</i> into table latch	<i>f</i> → TBLATH if <i>t</i> = 1 <i>f</i> → TBLATL if <i>t</i> = 0

Table B.2: 16-Bit Core Arithmetic and Logical Instruction

Hex	Mnemonic	Description	Function
b1kk	ADDLW <i>k</i>	Add literal to <i>W</i>	(<i>W</i> + <i>k</i>) → <i>W</i>
0eff	ADDWF <i>f, d</i>	Add <i>W</i> to <i>F</i>	(<i>W</i> + <i>f</i>) → <i>d</i>
10ff	ADDWFC <i>f, d</i>	Add <i>W</i> and Carry to <i>f</i>	(<i>W</i> + <i>f</i> + <i>C</i>) → <i>d</i>
b5kk	ANDLW <i>k</i>	AND Literal and <i>W</i>	(<i>W</i> .AND. <i>k</i>) → <i>W</i>
0aff	ANDWF <i>f, d</i>	AND <i>W</i> with <i>f</i>	(<i>W</i> .AND. <i>f</i>) → <i>d</i>
28ff	CLRF <i>f, d</i>	Clear <i>f</i> and Clear <i>d</i>	0x00 → <i>f</i> , 0x00 → <i>d</i>
12ff	COMF <i>f, d</i>	Complement <i>f</i>	.NOT. <i>f</i> → <i>d</i>
2eff	DAW <i>f, d</i>	Dec. adjust <i>W</i> , store in <i>f, d</i>	<i>W</i> adjusted → <i>f</i> and <i>d</i>

PIC17CXXX Instruction Set

Table B.2: 16-Bit Core Arithmetic and Logical Instruction (Continued)

Hex	Mnemonic		Description	Function
06ff	DECF	f, d	Decrement f	$(f - 1) \rightarrow f$ and d
14ff	INCF	f, d	Increment f	$(f + 1) \rightarrow f$ and d
b3kk	IORLW	k	Inclusive OR literal with W	$(W .OR. k) \rightarrow W$
08ff	IORWF	f, d	Inclusive or W with f	$(W .OR. f) \rightarrow d$
b0kk	MOVLW	k	Move literal to W	$k \rightarrow W$
bckk	MULLW	k	Multiply literal and W	$(k \times W) \rightarrow PH:PL$
34ff	MULWF	f	Multiply W and f	$(W \times f) \rightarrow PH:PL$
2cff	NEGW	f, d	Negate W, store in f and d	$(W + 1) \rightarrow f, (W + 1) \rightarrow d$
1aff	RLCF	f, d	Rotate left through carry	
22ff	RLNCF	f, d	Rotate left (no carry)	
18ff	RRCF	f, d	Rotate right through carry	
20ff	RRNCF	f, d	Rotate right (no carry)	
2aff	SETF	f, d	Set f and Set d	$0xff \rightarrow f, 0xff \rightarrow d$
b2kk	SUBLW	k	Subtract W from literal	$(k - W) \rightarrow W$
04ff	SUBWF	f, d	Subtract W from f	$(f - W) \rightarrow d$
02ff	SUBWFB	f, d	Subtract from f with borrow	$(f - W - c) \rightarrow d$
1cff	SWAPF	f, d	Swap f	$f(0:3) \rightarrow d(4:7),$ $f(4:7) \rightarrow d(0:3)$
b4kk	XORLW	k	Exclusive OR literal with W	$(W .XOR. k) \rightarrow W$
0cff	XORWF	f, d	Exclusive OR W with f	$(W .XOR. f) \rightarrow d$

MPLAB[®]-CXX Compiler User's Guide

Table B.3: 16-Bit Core Bit Handling Instructions

Hex	Mnemonic	Description	Function
8bff	BCF f , b	Bit clear f	0 → f(b)
8bff	BSF f , b	Bit set f	1 → f(b)
9bff	BTFSC f , b	Bit test, skip if clear	skip if f(b) = 0
9bff	BTFSS f , b	Bit test, skip if set	skip if f(b) = 1
3bff	BTG f , b	Bit toggle f	.NOT. f(b) → f(b)

Table B.4: 16-Bit Core Program Control Instructions

Hex	Mnemonic	Description	Function
ekkk	CALL k	Subroutine call (within 8k page)	PC+1 → TOS, k → PC(12:0), k(12:8) → PCLATH(4:0), PC(15:13) → PCLATH(7:5)
31ff	CPFSEQ f	Compare f/w, skip if f = w	f-W, skip if f = W
32ff	CPFSGT f	Compare f/w, skip if f > w	f-W, skip if f > W
30ff	CPFSLT f	Compare f/w, skip if f < w	f-W, skip if f < W
16ff	DECFSZ f , d	Decrement f, skip if 0	(f-1) → d, skip if 0
26ff	DCFSNZ f , d	Decrement f, skip if not 0	(f-1) → d, skip if not 0
ckkk	GOTO k	Unconditional branch (within 8k)	k → PC(12:0) k(12:8) → PCLATH(4:0), PC(15:13) → PCLATH(7:5)
1eff	INCFSZ f , d	Increment f, skip if zero	(f+1) → d, skip if 0
24ff	INFSNZ f , d	Increment f, skip if not zero	(f+1) → d, skip if not 0
b7kk	LCALL k	Long Call (within 64k)	(PC+1) → TOS; k → PCL, (PCLATH) → PCH
0005	RETFIE	Return from interrupt, enable interrupt	(PCLATH) → PCH; k → PCL 0 → GLINTD
b6kk	RETLW k	Return with literal in W	k → W, TOS → PC, (PCLATH unchanged)
0002	RETURN	Return from subroutine	TOS → PC (PCLATH unchanged)
33ff	TSTFSZ f	Test f, skip if zero	skip if f = 0

PIC17CXXX Instruction Set

Table B.5: 16-Bit Core Special Control Instructions

Hex	Mnemonic	Description	Function
0004	CLRWT	Clear watchdog timer	0 → WDT, 0 → WDT prescaler, 1 → PD, 1 → TO
0003	SLEEP	Enter Sleep Mode	Stop oscillator, power down, 0 → WDT, 0 → WDT Prescaler 1 → PD, 1 → TO

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Appendix C. PIC18CXXX Instruction Set

C.1 Introduction

This appendix gives the instruction set for the PIC18CXXX device family.

C.2 Highlights

This appendix presents the following reference information:

- Key to Enhanced 16-Bit Core Instruction Sets
- PIC18CXXX Instruction Set

C.3 Key to Enhanced 16-Bit Core Instruction Set

Field	Description
File Addresses	
f	8-bit file register address
fs	12-bit source file register address
fd	12-bit destination file register address
dest	W register if d = 0; file register if d = 1
Literals	
kk	8-bit literal value
kb	4-bit literal value
kc	bits 8-11 of 12-bit literal value
kd	bits 0-7 of 12-bit literal value
Offsets, Addresses	
nn	8-bit relative offset (signed, 2's complement)
nd	11-bit relative offset (signed, 2's complement)
ml	bits 0-7 of 20-bit program memory address
mm	bits 8-19 of 20-bit program memory address
xx	any 12-bit value
Bits	
b	bits 9-11; bit address
d	bit 9; 0=W destination; 1=f destination
a	bit 8; 0=common block; 1=BSR selects bank
s	bit 0 (bit 8 for CALL); 0=no update; 1(fast)=update/save W, STATUS, BSR

MPLAB[®]-CXX Compiler User's Guide

C.4 PIC18CXXX Instruction Set

Microchip's new high-performance 8-bit microcontroller family uses a 16-bit wide instruction set. This instruction set consists of 76 instructions, each a single 16-bit wide word (2 bytes). Most instructions operate on a file register, *f*, and the working register, *W* (accumulator). The result can be directed either to the file register or the *W* register or to both in the case of some instructions. A few instructions operate solely on a file register (BSF for example).

Table C.6: Enhanced 16-Bit Core Literal Operations

Hex	Mnemonic	Description	Function
0Fkk	ADDLW kk	ADD literal to W	W+kk → W
0Bkk	ANDLW kk	AND literal with W	W .AND. kk → W
0004	CLRWDT	Clear Watchdog Timer	0 → WDT, 0 → WDT postscaler, 1 → TO, 1 → PD
0007	DAW	Decimal Adjust W Register	if W<3:0> >9 or DC=1, W<3:0>+6→W<3:0>, else W<3:0> → W<3:0>; if W<7:4> >9 or C=1, W<7:4>+6→W<7:4>, else W<7:4> → W<7:4>;
09kk	IORLW kk	Inclusive OR literal with W	W .OR. kk → W
01kb	MOVLB kb	Move literal to low nibble in BSR	kb → BSR
EFkc F0kd	LFSR f,kd:kc	Load Literal to FSR (second word)	kd:kc → FSR
0Ekk	MOVLW kk	Move literal to W	kk → W
0Dkk	MULLW kk	Multiply literal with W	W * kk → PRODH:PRODL
08kk	SUBLW kk	Subtract W from literal	kk-W → W
0Akk	XORLW kk	Exclusive OR literal with W	W .XOR. kk → W

Table C.7: Enhanced 16-Bit Core Memory Operations

Hex	Mnemonic	Description	Function
0008	TBLRD *	Table Read (no change to TBLPTR)	Prog Mem (TBLPTR) → TABLAT
0009	TBLRD *+	Table Read (post-increment TBLPTR)	Prog Mem (TBLPTR) → TABLAT TBLPTR +1 → TBLPTR
000A	TBLRD *-	Table Read (post-decrement TBLPTR)	Prog Mem (TBLPTR) → TABLAT TBLPTR -1 → TBLPTR
000B	TBLRD +*	Table Read (pre-increment TBLPTR)	TBLPTR +1 → TBLPTR Prog Mem (TBLPTR) → TABLAT
000C	TBLWT *	Table Write (no change to TBLPTR)	TABLAT → Prog Mem(TBLPTR)
000D	TBLWT *+	Table Write (post-increment TBLPTR)	TABLAT → Prog Mem(TBLPTR) TBLPTR +1 → TBLPTR

PIC18CXXX Instruction Set

Table C.7: Enhanced 16-Bit Core Memory Operations (Continued)

Hex	Mnemonic	Description	Function
000E	TBLWT *-	Table Write (post-decrement TBLPTR)	TABLAT → Prog Mem(TBLPTR) TBLPTR -1 → TBLPTR
000F	TBLWT +*	Table Write (pre-increment TBLPTR)	TBLPTR +1 → TBLPTR TABLAT → Prog Mem(TBLPTR)

Table C.8: Enhanced 16-Bit Core Control Operations

Hex	Mnemonic	Description	Function	
E2nn	BC	nn	Relative Branch if Carry	if C=1, PC+2+2*nn→PC, else PC+2→PC
E6nn	BN	nn	Relative Branch if Negative	if N=1, PC+2+2*nn→PC, else PC+2→PC
E3nn	BNC	nn	Relative Branch if Not Carry	if C=0, PC+2+2*nn→PC, else PC+2→PC
E5nn	BNOV	nn	Relative Branch if Not Overflow	if OV=0, PC+2+2*nn→PC, else PC+2→PC
E1nn	BNZ	nn	Relative Branch if Not Zero	if Z=0, PC+2+2*nn→PC, else PC+2→PC
E7nn	BNN	nn	Relative Branch if Not Negative	if N=0, PC+2+2*nn→PC, else PC+2→PC
E0nd	BRA	nd	Unconditional relative branch	PC+2+2*nd→PC
E4nn	BOV	nn	Relative Branch if Overflow	if OV=1, PC+2+2*nn→PC, else PC+2→PC
E0nn	BZ	nn	Relative Branch if Zero	if Z=1, PC+2+2*nn→PC, else PC+2→PC
ECml Fmm	CALL	mm:ml,s	Absolute Subroutine Call (second word)	PC+4 → TOS, mm:ml → PC<20:1>, if s=1, W → WS, STATUS → STATUS, BSR → BSRS
EFml Fmm	GOTO	mm:ml	Absolute Branch (second word)	mm:ml → PC<20:1>
0001	HALT		Halt processor	PC=halt
0000	NOP		No Operation	No operation
0006	POP		Pop Top of return stack	TOS-1 → TOS
0005	PUSH		Push Top of return stack	PC +2 → TOS
D8nd	RCALL	nd	Relative Subroutine Call	PC+2 → TOS, PC+2+2*nd→PC
00FF	RESET		Generate a Reset (same as MCR reset)	same as MCLR reset
0010	RETFIE	s	Return from interrupt (and enable interrupts)	TOS → PC, 1 → GIE/GIEH or PEIE/GIEL, if s=1, WS → W, STATUS → STATUS, BSRS → BSR, PCLATU/PCLATH unchngd.
0Ckk	RETLW	kk	Return from subroutine, literal in W	kk → W,

MPLAB[®]-CXX Compiler User's Guide

Table C.8: Enhanced 16-Bit Core Control Operations (Continued)

Hex	Mnemonic	Description	Function
0012	RETURN s	Return from subroutine	TOS → PC, if s=1, WS → W, STATUSS → STATUS, BSRS → BSR, PCLATU/PCLATH are unchanged
0003	SLEEP	Enter SLEEP Mode	0 → WDT, 0 → WDT postscaler, 1 → TO, 0 → PD

Table C.9: Enhanced 16-Bit Core Bit Operations

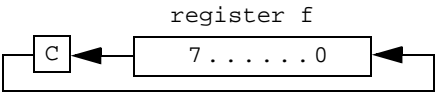
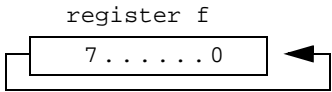
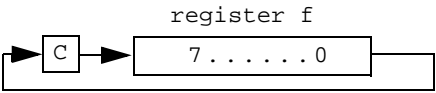
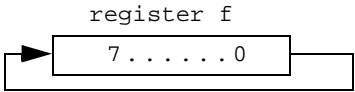
Hex	Mnemonic	Description	Function
9bf	BCF f,b,a	Bit Clear f	0 → f
8bf	BSF f,b,a	Bit Set f	1 → f
Bbf	BTFSC f,b,a	Bit test, skip if clear	if f=0, PC+4→PC, else PC+2→PC
Abf	BTFSS f,b,a	Bit test, skip if set	if f=1, PC+4→PC, else PC+2→PC
7bf	BTG f,b,a	Bit Toggle f	f → f

Table C.10: Enhanced 16-Bit Core File Register Operations

Hex	Mnemonic	Description	Function
24f	ADDWF f,d,a	ADD W to f	W+f → dest
20f	ADDWFC f,d,a	ADD W and Carry bit to f	W+f+C → dest
14f	ANDWF f,d,a	AND W with f	W .AND. f → dest
6Af	CLRF f,a	Clear f	0 → f
1Cf	COMF f,d,a	Complement f	f → dest
62f	CPFSEQ f,a	Compare f with W, skip if f=W	f-W, if f=W, PC+4 → PC else PC+2 → PC
64f	CPFSGT f,a	Compare f with W, skip if f > W	f-W, if f > W, PC+4 → PC else PC+2 → PC
60f	CPFSLT f,a	Compare f with W, skip if f < W	f-W, if f < W, PC+4 → PC else PC+2 → PC
04f	DECF f,d,a	Decrement f	f-1 → dest
2Cf	DECFSZ f,d,a	Decrement f, skip if 0	f-1 → dest, if dest=0, PC+4 → PC else PC+2 → PC
4Cf	DCFSNZ f,d,a	Decrement f, skip if not 0	f-1 → dest, if dest ≠ 0, PC+4 → PC else PC+2 → PC
28f	INCF f,d,a	Increment f	f+1 → dest
3Cf	INCFSZ f,d,a	Increment f, skip if 0	f+1 → dest, if dest=0, PC+4 → PC else PC+2 → PC
48f	INFSNZ f,d,a	Increment f, skip if not 0	f+1 → dest, if dest ≠ 0, PC+4 → PC else PC+2 → PC

PIC18CXXX Instruction Set

Table C.10: Enhanced 16-Bit Core File Register Operations (Continued)

Hex	Mnemonic		Description	Function
10f	IORWF	f,d,a	Inclusive OR W with f	W .OR. f → dest
50f	MOVF	f,d,a	Move f	f → dest
Cfs Ffd	MOVFF	fs,fd	Move fs to fd (second word)	fs → fd
6Ef	MOVWF	f,a	Move W to f	W → f
02f	MULWF	f,a	Multiply W with f	W * f → PRODH:PRODL
6Cf	NEGF	f,a	Negate f	f + 1 → f
34f	RLCF	f,d,a	Rotate left f through Carry	
44f	RLNCF	f,d,a	Rotate left f (no carry)	
30f	RRCF	f,d,a	Rotate right f through Carry	
40f	RRNCF	f,d,a	Rotate right f (no carry)	
48f	SETF	f,a	Set f	0xFF → f
54f	SUBFWB	f,d,a	Subtract f from W with Borrow	W - f - C → dest
5Cf	SUBWF	f,d,a	Subtract W from f	f - W → dest
58f	SUBWFB	f,d,a	Subtract W from f with Borrow	f - W - C → dest
38f	SWAPF	f,d,a	Swap f	f<3:0> → dest<7:4>, f<7:4> → dest<3:0>
66f	TSTFSZ	f,a	Test f, skip if 0	PC+4 → PC, if f=0, else PC+2 → PC
18f	XORWF	f,d,a	Exclusive OR W with f	W .XOR. f → dest

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Appendix D. MPLAB-C17 Errors

D.1 Introduction

This appendix lists errors generated by the MPLAB-C17 compiler.

D.2 Highlights

The following errors apply to MPLAB-C17:

- Errors
- Warnings

D.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

MPLAB[®]-CXX Compiler User's Guide

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: inline 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.

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

MPLAB[®]-CXX Compiler User's Guide

- 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'

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

D.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

MPLAB[®]-CXX Compiler User's Guide

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

Appendix E. MPLAB-C18 Errors

E.1 Introduction

This appendix lists errors generated by the MPLAB-C18 compiler.

E.2 Highlights

The following errors apply to MPLAB-C18:

- Errors
- Warnings

E.3 Errors

1000: unexpected input following '%s'
1001: identifier expected following '%s'
1002: syntax error, '%s' expected
1003: unknown pre-processor directive '%s'
1004: missing ')' in macro on line %d
1005: missing argument %d invoking macro %s
1006: extra argument(s) invoking macro '%s'
1007: '(' expected invoking macro '%s'
1008: #else without #if detected
1009: #elif without #if detected
1010: #endif without #if detected
1012: identifier expected in macro definition
1013: error in pragma directive
1014: attribute mismatch in resumption of section '%s'
1011: missing #endif
1015: missing include path for system header files
1050: section address permitted only at definition
1051: section pragma not allowed inside a function
1052: section overlay attribute does not match definition
1053: section share attribute does not match definition
1054: section type does not match definition
1100: syntax error

MPLAB[®]-CXX Compiler User's Guide

1101: lvalue required

An expression which designates an object is required. Common causes include missing parentheses and a missing '*' operator.

1102: cannot assign to 'const' modified object

An object qualified with 'const' is declared to be read-only data and modifications to it are therefore not allowed.

1103: unknown escape sequence '%s'

The specified escape sequence is not known to the compiler. Check the User's Guide for a list of valid character escape sequences

1104: division by zero in constant expression

The compiler cannot process a constant expression which contains a divide by (or modulus by) zero.

1105: symbol '%s' has not been defined

A symbol has been referenced before it has been defined. Common causes include a misspelled symbol name, a missing header file which declares the symbol, and a reference to a symbol valid only in an inner scope.

1500: unable to open file '%s'

The compiler was unable to open the named file. Common causes include misspelled filename and insufficient access rights

1501: unable to locate file '%s'

The compiler was unable to locate the named file. Common causes include misspelled filename and misconfigured include path.

1106: '%s' is not a function

A symbol must be a function name in order to be declared as an interrupt function

1107: interrupt functions must not take parameters

When the processor vectors to an interrupt routine, no parameters are passed, so a function declared as an interrupt function should not expect parameters.

1108: interrupt functions must not return a value

Since interrupts are invoked asynchronously by the processor, there will not be a calling routine to which a value can be returned.

1109: 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.

1110: 'auto' symbol '%s' not in function scope

Variables may only be allocated off the stack within the scope of a function.

1111: undefined label '%s' in '%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.

1112: integer type expected in switch control expression

The control expression for a switch statement must be an integer type. Common causes include a missing '*' operator and a missing '[' operator.

1113: integer constant expected for case label value

The value for a case label must be an integer constant.

1114: case label outside switch statement detected

A 'case' label is only valid inside the body of a switch statement. Common causes include a misplaced '}'.

1115: multiple default labels in switch statement

A switch statement can only have a single 'default' label. Common causes include a missing '}' to close an inner switch.

1116: type mismatch in return statement

The type of the return value is not compatible with the declared return type of the function. Common causes include a missing '*' or '[' operator.

1117: scalar type expected in 'if' statement

An 'if' statement control expression must be of scalar type, i.e., an integer or a pointer.

1118: scalar type expected in 'while' statement

A 'while' statement control expression must be of scalar type, i.e., an integer or a pointer.

1119: scalar type expected in 'do..while' statement

A 'do..while' statement control expression must be of scalar type, i.e., an integer or a pointer.

1120: scalar type expected in 'for' statement

A 'for' statement control expression must be of scalar type, i.e., an integer or a pointer.

1120: scalar type expected in '?:' expression

A '?:' operator control expression must be of scalar type, i.e., an integer or a pointer.

1122: scalar operand expected for '!' operator

The '!' operator requires that its operand be of scalar type.

1123: scalar operands expected for '||' operator

The logical OR operator, '||', requires scalar operands.

MPLAB[®]-CXX Compiler User's Guide

1124: scalar operands expected for '&&' operator

The logical AND operator, '&&', requires scalar operands.

1125: 'break' must appear in a loop or switch statement

A 'break' statement must be inside a 'while', 'do', 'for', or 'switch' statement. Common causes include a misplaced '}'.

1126: 'continue' must appear in a loop statement

A 'continue' statement must be inside a 'while', 'do', 'for', or 'switch' statement.

1127: operand type mismatch in '?' operator

The types of the result operands of the '?' operator must be either both scalar types, or compatible types.

1128: compatible scalar operands required for comparison

A comparison operator must have operands of compatible scalar types.

1129: [] operator requires a pointer and an integer as operands

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)'.

1130: pointer operand required for '*' operator

The '*' dereference operator requires a pointer to a non-void object as its operand

1131: type mismatch in assignment

The assignment operators require that the result of the right hand expression be of compatible type with the type of the result of the left hand expression. Common causes include a missing '*' or '[]' operator

1132: integer type expected for right hand operand of '-=' operator

The '-=' operator requires that the right hand side be of integer type when the left hand side is of pointer type. Common causes include a missing '*' or '[]' operator.

1133: type mismatch in '-=' operator

The types of the operands of the '-=' operator must be such that for 'x=y' the expression 'x=x-y' is valid.

1134: arithmetic operands required for multiplication operator

The '*' and '*=' multiplication operators require that their operands be of arithmetic type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1134: arithmetic operands required for division operator

The '/' and '/=' division operators require that their operands be of arithmetic type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1135: integer operands required for modulus operator

The '%' and '%=' division operators require that their operands be of integer type. Common causes include a missing '*' dereference operator or a missing '[' index operator.

1136: integer operands required for shift operator

The bitwise shift operators require that their operands be of integer type. Common causes include a missing '*' dereference operator or a missing '[' index operator.

1137: integer types required for bitwise AND operator

The '&' and '&=' operators require that both operands be of integer type. Common causes include a missing '*' or '[' operator

1138: integer types required for bitwise OR operator

The '|' and '|=' operators require that both operands be of integer type. Common causes include a missing '*' or '[' operator

1139: integer types required for bitwise XOR operator

The '^' and '^=' operators require that both operands be of integer type. Common causes include a missing '*' or '[' operator

1139: integer type required for bitwise NOT operator

The '~' operator requires that the operand be of integer type. Common causes include a missing '*' or '[' operator

1141: integer type expected for pointer addition

The addition operator requires that when one operand is of pointer type, the other must be of integer type. Common causes include a missing '*' or '[' operator.

1142: type mismatch in '+' operator

The types of the operands of the '+' operator must be such that one operand is of pointer type and the other is of integer type or both operands are of arithmetic type.

1143: pointer difference requires pointers to compatible types

When calculating the difference between two pointers, the pointers must point to objects of compatible type. Common causes include missing parentheses and a missing '[' operator.

1144: integer type required for pointer subtraction

When the left hand operand of the subtraction operator is of pointer type, the right hand operand must be of integer type. Common causes include a missing '*' or '[' operator.

1145: arithmetic type expected for subtraction operator

When the left hand operand is not of pointer type, the subtraction operator requires that both operands be of arithmetic type.

MPLAB[®]-CXX Compiler User's Guide

1146: 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

1147: scalar type expected for increment operator

The increment operators require that the operand be a modifiable lvalue of scalar type.

1148: scalar type expected for decrement operator

The decrement operators require that the operand be a modifiable lvalue of scalar type.

1149: arithmetic type expected for unary plus

The unary plus operator requires that its operand be of arithmetic type

1150: arithmetic type expected for unary minus

The unary minus operator requires that its operand be of arithmetic type

1151: struct or union object designator expected

The member access operators, '.' and '->' require operands of struct/union and pointer to struct/union, respectively

1152: scalar or void type expected for cast

An explicit cast requires that the type of the operand be of scalar type and the type being cast to be scalar type or void type.

1200: cannot reference the address of a bitfield

The address of a bitfield member of a structure cannot be referenced directly.

1201: cannot dereference a pointer to 'void' type

The '*' dereference operator requires a pointer to a non-void object as its operand

1202: 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.

1203: too few arguments in function call

To call a function, the number of arguments passed must match exactly the number of parameters declared for the function.

1204: too many arguments in function call

To call a function, the number of arguments passed must match exactly the number of parameters declared for the function.

1207: tag '%s' is incomplete

An incomplete struct or union tag cannot be referenced by the member access operators. Common causes include a misspelled structure tag name in the symbol definition.

1205: unknown member '%s' in '%s'

The structure or union tag does not have a member of the name requested. Common causes include a misspelled member name and a missing member access operator for a nested structure.

1206: unknown member '%s'

The structure or union type does not have a member of the name requested. Common causes include a misspelled member name and a missing member access operator for a nested structure.

1208: "#pragma interrupt" detected inside function body

The 'interrupt' pragma is only available at file level scope.

1209: unknown function '%s' in #pragma interrupt

The 'interrupt' pragma requires that the function being declared as an interrupt have an active prototype when the pragma is encountered

1210: unknown symbol '%s' in interrupt save list

The 'interrupt' pragma requires that symbols listed in the 'save' list must be declared and of in scope

1211: missing definition for interrupt function '%s'

The function was declared as an interrupt, but was never defined. The function definition of an interrupt function must be in the same module as the pragma declaring the function as an interrupt.

1212: static function '%s' referenced but not defined

The function has been declared as static and has been referenced elsewhere in the module, but there is no definition for the function present. Common causes include a misspelled function name in the function definition.

1300: stack frame too large

The size of the stack frame has exceeded the maximum addressable size. Commonly caused by too many local variables allocated as 'auto' storage class in a single function.

1301: parameter frame too large

The size of the parameter frame has exceeded the maximum addressable size. Commonly caused by too many parameters being passed to a single function.

1099: %s

source code '#error' directive message

E.4 Warnings

2000: redefinition of macro '%s'

2051: storage class mismatch in redeclaration of '%s'

2060: shift expression has no effect

2061: shift expression always zero

2052: unexpected return value

A return of a value statement has been detected in a function declared to return no value. The return value will be ignored.

2053: return value expected

A return with no value has been detected in a function declared to return a value. The return value will be undefined.

2054: suspicious pointer conversion

A pointer has been used as an integer or an integer has been used as a pointer without an explicit cast.

2055: expression is always false

The control expression of a conditional statement evaluates to a constant false value

2056: expression is always true

The control expression of a conditional statement evaluates to a constant true value

2057: possibly incorrect test of assignment

An implicit test of an assignment expression, e.g., 'if(x=y)' is often seen when a '=' operator has been used when a '==' operator was intended.

2058: call of function 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.

2059: unary minus of unsigned value

The unary minus operator is normally only applied to signed values.

2062: '->' operator expected, not '.'

A struct/union member access via a pointer to struct/union has been performed using the '.' operator.

2063: '.' operator expected, not '->'

A direct struct/union member access has been performed using the '->' operator.

2064: static function '%s' not defined

MPLAB-C18 Errors

The function has been declared as static, but there is no definition for the function present. Common causes include a misspelled function name in the function definition.

2065: static function '%s' never referenced

The static function has been defined, but has not been referenced.

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Appendix F. References

F.1 Introduction

This appendix gives references that may be helpful in programming with MPLAB-CXX.

F.2 Highlights

This appendix lists the following reference types:

- C Standards Information
- General C Information

F.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.

F.4 General C Information

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.

Harbison, Samuel P., and Steele, Guy L., *C A Reference Manual*, Fourth Edition, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

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.

MPLAB[®]-CXX Compiler User's Guide

NOTES:

Glossary

Introduction

To provide a common frame of reference, this glossary defines the terms for several Microchip tools.

Highlights

This glossary contains terms and definitions for the following tools:

- MPLAB IDE, MPLAB-SIM, MPLAB Editor
- MPASM, MPLINK, MPLIB
- MPLAB-CXX
- MPLAB-ICE, PICMASTER Emulators
- MPLAB-ICD
- PICSTART Plus, PRO MATE programmer

Terms

Absolute Section

A section with a fixed (absolute) address which can not be changed by the linker.

Access RAM (PIC18CXXX Devices Only)

Special general purpose registers on PIC18CXXX devices that allow access regardless of the setting of the bank select bit (BSR).

Alpha Character

Alpha characters are those characters, regardless of case, that are letters of the alphabet: (a, b, ..., z, A, B, ..., Z).

Alphanumeric

Alphanumeric characters include alpha characters and numbers: (0,1, ..., 9).

Application

A set of software and hardware developed by the user, usually designed to be a product controlled by a PICmicro microcontroller.

Assemble

What an assembler does. See assembler.

Assembler

A language tool that translates a user's assembly source code (.asm) into machine code. MPASM is Microchip's assembler.

MPLAB[®]-CXX Compiler User's Guide

Assembly

A programming language that is once removed from machine language. Machine languages consist entirely of numbers and are almost impossible for humans to read and write. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names (mnemonics) instead of numbers.

Assigned Section

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.

Break Point – Hardware

An event whose execution will cause a halt.

Break Point – Software

An address where execution of the firmware will halt. Usually achieved by a special break opcode.

Build

A function that recompiles all the source files for an application.

C

A high level programming language that may be used to generate code for PICmicro MCUs, 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.

COFF

Common Object File Format. An intermediate file format generated by MPLINK that contains machine code and debugging information.

Command Line Interface

Command line interface refers to executing a program on the DOS command line with options. Executing MPASM with any command line options or just the file name will invoke the assembler. In the absence of any command line options, a prompted input interface (shell) will be executed.

Compile

What a compiler does. See compiler.

Compiler

A language tool that translates a user's C source code into machine code. MPLAB-C17 and MPLAB-C18 are Microchip's C compilers for PIC17CXXX and PIC18CXXX devices, respectively.

Configuration Bits

Unique bits programmed to set PICmicro microcontroller modes of operation. A configuration bit may or may not be preprogrammed. These bits are set in the *Options > Development Mode* dialog for simulators or emulators and in the `_ _ CONFIG MPASM` directive for programmers.

Control Directives

Control directives in MPASM permit sections of conditionally assembled code.

Data Directives

Data directives are those that control MPASM's allocation of memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

General purpose file registers (GPRs) from RAM on the PICmicro device being emulated. The File Register window displays data memory.

Directives

Directives provide control of the assembler's operation by telling MPASM how to treat mnemonics, define data, and format the listing file. Directives make coding easier and provide custom output according to specific needs.

Download

Download is the process of sending data from the PC host to another device, such as an emulator, programmer or target board.

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.

Emulation

The process of executing software loaded into emulation memory as if the firmware resided on the microcontroller device under development.

Emulation Memory

Program memory contained within the emulator.

Emulator

Hardware that performs emulation.

Emulator System

The MPLAB-ICE emulator system includes the pod, processor module, device adapter, cables, and MPLAB Software. The PICMASTER emulator system includes the pod, device-specific probe, cables, and MPLAB Software.

Event

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 break points.

MPLAB[®]-CXX Compiler User's Guide

Executable Code

See Hex Code.

Export

Send data out of the MPLAB IDE in a standardized format.

Expressions

Expressions are used in the operand field of MPASM's source line and may contain constants, symbols, or any combination of constants and symbols separated by arithmetic operators. Each constant or symbol may be preceded by a plus or minus to indicate a positive or negative expression.

<p>Note: MPASM expressions are evaluated in 32 bit integer math. (Floating point is not currently supported.)</p>
--

Extended Microcontroller Mode (PIC17CXXX and PIC18CXXX 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 PIC17CXXX or PIC18CXXX device.

External Input Line (MPLAB-ICE only)

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

External Linkage

A function or variable has external linkage if it can be accessed from outside the module in which it is defined.

External RAM (PIC17CXXX and PIC18CXXX Devices Only)

Off-chip Read/Write memory.

External Symbol

A symbol for an identifier which has external linkage.

External Symbol Definition

A symbol for a function or variable defined in the current module.

External Symbol Reference

A symbol which references a function or variable defined outside the current module.

External Symbol Resolution

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to update all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

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 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.

GPR

See Data Memory.

Halt

A function that stops the emulator. Executing Halt is the same as stopping at a break point. The program counter stops, and the user can inspect and change register values, and single step through code.

Hex Code

Executable instructions assembled or compiled from source code into standard hexadecimal format code. Also called executable or machine 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. This format is readable by a device programmer.

High Level Language

A language for writing programs that is of a higher level of abstraction from the processor than assembler code. High level languages (such as C) employ a compiler to translate statements into machine instructions that the target processor can execute.

ICD

In-Circuit Debugger. MPLAB-ICD is Microchip's in-circuit debugger for PIC16F87X devices. MPLAB-ICD works with MPLAB IDE.

ICE

In-Circuit Emulator. MPLAB-ICE is Microchip's in-circuit emulator that works with MPLAB IDE.

IDE

Integrated Development Environment. An application that has multiple functions for firmware development. The MPLAB IDE integrates a compiler, an assembler, a project manager, an editor, a debugger, a simulator, and an

MPLAB[®]-CXX Compiler User's Guide

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

Bring data into the MPLAB Integrated Development Environment (IDE) from an outside source, such as from a hex file.

Initialized Data

Data which is defined with an initial value. In C, `int myVar=5;` defines a variable which will reside in an initialized data section.

Internal Linkage

A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

Librarian

A language tool that creates and manipulates libraries. MPLIB is Microchip's librarian.

Library

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.

Link

What a linker does. See Linker.

Linker

A language tool that combines object files and libraries to create executable code. Linking is performed by Microchip's linker, MPLINK.

Linker Script Files

Linker script files are the command files of MPLINK (.LKR). They define linker options and describe available memory on the target platform.

Listing Directives

Listing directives are those directives that control the MPASM listing file format. They allow the specification of titles, pagination and other listing control.

Listing File

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, MPASM directive, or macro encountered in a source file.

Local Label

A local label is one that is defined inside a macro with the `LOCAL` directive. These labels are particular to a given instance of a macro's instantiation. In other words, the symbols and labels that are declared as local are no longer accessible after the `ENDM` macro is encountered.

Logic Probes

Up to 14 logic probes connected to the emulator. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Machine Code

Either object or executable code.

Macro

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

Directives that control the execution and data allocation within macro body definitions.

Make Project

A command that rebuilds an application, re-compiling only those source files that have changed since the last complete compilation.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also μ C.

Memory Models

Versions of libraries and/or precompiled object files based on a device's memory (RAM/ROM) size and structure.

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 (PIC17CXXX and PIC18CXXX Devices Only)

One of the possible program memory configurations of the PIC17CXXX and PIC18CXXX families of microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

MPLAB[®]-CXX Compiler User's Guide

Microprocessor Mode (PIC17CXXX and PIC18CXXX Devices Only)

One of the possible program memory configurations of the PIC17CXXX and PIC18CXXX 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.

MPASM

Microchip Technology's relocatable macro assembler. MPASM is a DOS or Windows-based PC application that provides a platform for developing assembly language code for Microchip's PICmicro microcontroller families. Generically, MPASM will refer to the entire development platform including the macro assembler and utility functions.

MPASM will translate source code into either object or executable code. The object code created by MPASM may be turned into executable code through the use of the MPLINK linker.

MPLAB-CXX

Refers to MPLAB-C17 and MPLAB-C18 C compilers.

MPLAB-ICD

Microchip's in-circuit debugger for PIC16F87X devices. MPLAB-ICD works with MPLAB IDE. The MPLAB-ICD system consists of a module, header, demo board (optional), cables, and MPLAB Software.

MPLAB-ICE

Microchip's in-circuit emulator that works with MPLAB IDE.

MPLAB IDE

The name of the main executable program that supports the IDE with an Editor, Project Manager, and Emulator/Simulator Debugger. The MPLAB Software resides on the PC host. The executable file name is MPLAB.EXE. MPLAB.EXE calls many other files.

MPLAB-SIM

Microchip's simulator that works with MPLAB IDE.

MPLIB

MPLIB is a librarian for use with COFF object modules (*filename.o*) created using either MPASM v2.0, MPASMWIN v2.0, or MPLAB-C v2.0 or later.

MPLIB will combine multiple object files into one library file. Then MPLIB can be used to manipulate the object files within the created library.

MPLINK

MPLINK is a linker for the Microchip relocatable assembler, MPASM, and the Microchip C compilers, MPLAB-C17 or MPLAB-C18. MPLINK also may be used with the Microchip librarian, MPLIB. MPLINK is designed to be used with MPLAB IDE, though it does not have to be.

MPLINK will combine object files and libraries to create a single executable file.

MPSIM

The DOS version of Microchip's simulator. MPLAB-SIM is the newest simulator from Microchip.

MRU

Most Recently Used. Refers to files and windows available to be selected from MPLAB IDE main pull down menus.

Nesting Depth

The maximum level to which macros can include other macros. Macros can be nested to 16 levels deep.

Non Real-Time

Refers to the processor at a break point or executing single step instructions or MPLAB IDE being run in simulator mode.

Node

MPLAB IDE project component.

NOP

No Operation. An instruction that has no effect when executed except to advance the program counter.

Object Code

The intermediate code that is produced from the source code after it is processed by an assembler or compiler. Relocatable code is code produced by MPASM or MPLAB-C17/C18 that can be run through MPLINK to create executable code. Object code is contained in an object file.

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.

MPLAB[®]-CXX Compiler User's Guide

Off-Chip Memory (PIC17CXXX and PIC18CXXX Devices Only)

Off-chip memory refers to the memory selection option for the PIC17CXXX or PIC18CXXX 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.

Pass Counter

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. MPLAB IDE runs on 486 or higher machines.

PICmicro MCUs

PICmicro microcontrollers (MCUs) refers to all Microchip microcontroller families.

PICMASTER Emulator

The hardware unit that provides tools for emulating and debugging firmware applications. This unit contains emulation memory, break point logic, counters, timers, and a trace analyzer among some of its tools. MPLAB-ICE is the newest emulator from Microchip.

PICSTART Plus

A device programmer from Microchip. Programs 8, 14, 28, and 40 pin PICmicro microcontrollers. Must be used with MPLAB Software.

Pod

The external emulator box that contains emulation memory, trace memory, event and cycle timers, and trace/break point logic. Occasionally used as an abbreviated name for the MPLAB-ICE emulator.

Power-on-Reset Emulation

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

Precedence

The concept that some elements of an expression are evaluated before others; i.e., * and / before + and -. In MPASM, 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.

Program Memory

The memory area in a PICmicro microcontroller where instructions are stored. 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

A set of source files and instructions to build the object and executable code for an application.

PRO MATE

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 devices have a PWM peripheral.

Qualifier

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

Radix

The number base, hex, 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.

MPLAB[®]-CXX Compiler User's Guide

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 break point 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.

Relocatable Section

A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation

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.

Section

An portion of code or data which has a name, size, and address.

SFR

Special Function Registers of a PICmicro.

Shared Section

A section which resides in a shared (non-banked) region of data RAM.

Shell

The MPASM shell is a prompted input interface to the macro assembler. There are two MPASM shells: one for the DOS version and one for the Windows version.

Simulator

A software program that models the operation of the PICmicro microprocessor.

Single Step

This command steps through 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

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcode 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 opcode 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

When a hardware break point 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 break point is referred to as the skid.

Source Code - Assembly

Source code consists of PICmicro instructions and MPASM 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 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 instructions and MPASM 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.

MPLAB[®]-CXX Compiler User's Guide

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 PIC18CXXX 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

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

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a `CALL` instruction into a subroutine.

Step Over

Step Over allows you to debug code without stepping into subroutines. When stepping over a `CALL` instruction, the next break point 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 break point will never be reached.

The Step Over command is the same as Single Step except for its handling of `CALL` instructions.

Stimulus

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

A counter for measuring execution cycles.

Symbol

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.

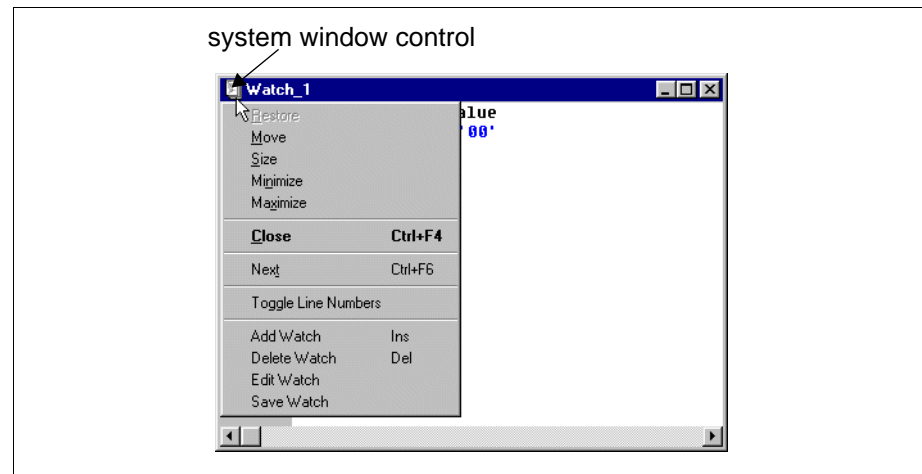


Figure G1: System Window Control Menu - Watch Window

Target

Refers to user hardware.

Target Application

Firmware residing on the target board.

Target Board

The circuitry and programmable device that makes up the target application.

Target Processor

The microcontroller device on the target application board that is being emulated.

Template

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

A row or column of icons that you can click on to execute MPLAB IDE functions.

MPLAB[®]-CXX Compiler User's Guide

Trace

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

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trigger Output

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 break point settings. Any number of trigger output points can be set.

Unassigned Section

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.

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

A variable that you may monitor during a debugging session in a watch window.

Watch Window

Watch windows contain a list of watch variables that are updated at each break point.

Index

Symbols

!	100
-	99
!=	100
#define	76
#elif	77
#else	77
#endif	77
#error	77
#if	78
#ifdef	78
#ifndef	79
#include	79
#line	80
#pragma interrupt	80, 125, 138
#pragma list	82
#pragma nolist	82
#pragma varlocate	84
#undef	84
%	99
&	101
&&	100
*	99
+	99
.asm	25, 39
.c	21, 25, 39
.cod	25, 39
.err	21, 28, 33
.h	28, 33
.hex	25, 29, 34, 39, 43
.lib	25, 39
.lkr	25, 39
.lst	25, 39
.map	25, 29, 34, 39
.o	21, 25, 39
.out	25, 29, 34, 39
/	99
/*	85
//	85
<	100
<<	101
<=	100
==	100
>	100

>=	100
>>	101
^	101
	101
	100
~	101

A

Absolute Section	187
Access RAM	187
Add Project Files	44, 58
Address Spaces, ROM and RAM	114
ALUSTA	131
AND (&&)	100
AND, Bitwise (&)	101
ANSI Compatibility	14, 145
Arithmetic Operators	99
Arrays	112
ANSI C	148
Initialization	113
ASCII	155
asm (_asm)	73, 126
Assembler	187
Assembler, Internal	126
Assembly Language, Mixing with C	140
Assigned Section	188
Assignment Operators	101
auto	88
AUTOEXEC.BAT	18

B

Banked/Paged Data	88, 126
Basic Data Types	88
bin directory	18, 27, 33
Binary	86
Bit-Fields	122
ANSI C	149
Bitwise Operators	101
Block Comment	85
break	110
Break Point, Hardware	188
Break Point, Software	188
BSR	131

MPLAB[®]-CXX Compiler User's Guide

C

C Keywords	73
C Programming	75
C++ Comment	85
c018.o	137
c018i.o	137
c0l17.asm	129
c0s17.asm	129
Calibration Memory	188
case	109
char	88, 89
Characters	
ANSI C	146
ClrWdt()	124
Code	
Start Up	25, 28, 39, 47, 125, 129, 136
Code File	25, 39
COFF File	25, 29, 34, 39
Command Line	
Multiple File Compile, MPLAB-C17	30
Multiple File Compile, MPLAB-C18	35
Options, MPLAB-C17	26
Options, MPLAB-C18	31
Single File Compile, MPLAB-C17	27
Single File Compile, MPLAB-C18	33
Command Line Interface	188
Comments	85, 126
Compiler	188
Compiler Overview	19
Conditional Operator	103
Configuration Bits	189
const	88
Constants	
Character	86, 87
Numeric	86, 87
String	84, 87
continue	111
Customer Notification Service	7
Customer Support	9

D

Data Memory	189
Data Types	89
Decrement Operators	102
default	109
Development Mode	41, 55
Directives	189
Control	189
Data	189

Listing	192
Macro	193
Object File	195
doc directory	18
Document Conventions	3
Document Layout	1
double	88, 90
do-while	108

E

Edit Project	43, 57
EEPROM	189
else	107
Emulator	189
endasm (_endasm)	73, 126
Enumerations	93
ANSI C	149
Environment Variable <i>See MCC_INCLUDE</i>	
equal to (==)	100
Error File	21, 28, 33
Escape Sequences	86
Example Code	151
examples directory	18
Executable	
Directory	18, 27, 33
Files	15, 25, 29, 34, 37, 39
Executable Code	190
Export	190
Expressions	190
Extended Microcontroller Mode	190
extern	88
External Declaration	123, 124, 127, 135
External RAM	190

F

far	73, 88, 90, 118, 126, 128, 135
File	
Listing	192
float	88, 90
Floating Point	
ANSI C	147
for	107
FSR0, PIC17CXXX Hardware	133
FSR1, PIC17CXXX Hardware	133
FSR1, PIC18CXXX Software	136
FSR2, PIC18CXXX Software	136
Functions	96
Declaration	96
Funky	96
Passing Arguments	97

Prototyping	96
Recursive	97
Returning Values	98
G	
global	90
Global variables	91
Glossary	187
Going Forward	36, 68
goto	111
H	
h directory	18, 27, 33, 43, 57
Header	
Directory	18, 27, 33
File	28, 33
Header Directory	43, 57
Hex Code	191
Hex File	25, 29, 34, 39, 43, 57
Hexadecimal	86
I	
ICD	191
ICE	191
IDE	191
Identifiers	
ANSI C	145
if	106, 107
Import	192
Include	
Current Search Path	43, 57, 79
Directory	27
Increment Operators	102
Initialization	
Arrays	113
Data	25, 28, 39, 47, 130, 137
Stack	130, 137
Initialized Data	192
Install_TMR0	132
Install Language Tool	51, 65
Install_INT	127
Install_PIV	127
Install_T0CKI	127
Install_TMR0	127
Installation	
Procedure	16
Requirements	15
Instruction Sets	157, 163
Integers	
ANSI C	146
int	88, 89
Internet Address	6
Interrupts	
Handler Code	25, 28, 39, 47
Nested	131
Saving FSR	133
Support - MPLAB-C18	125, 138
Support Macros - MPLAB-C17	131
L	
Left Shift (<<)	101
lib directory	18, 27, 33, 47, 61, 62
Librarian	192
Librarian <i>See MPLIB</i>	
Libraries	13
Library	192
Directory	18, 27, 33, 47, 61, 62
Files	25, 39
Linker	192
Directory	18, 27, 33
Script	25, 39, 49, 63
Linker Script Files	192
Linker <i>See MPLINK</i>	
list	82
Listing File	25, 39, 192
lkr directory	18, 27, 33
local	90
Local Label	193
Local Variables	91
Logic Probes	193
Logical Operators	100
long	88, 89
M	
main(), branching to	130, 137
Make Project	51, 65
Map File	25, 29, 34, 39
MCC_INCLUDE	18, 27, 33, 79
mcc17	15, 26, 37
mcc17d	15, 37
mcc18	15, 37
MCU	193
Memory	
Calibration	188
Data	189
Models	47, 61, 123
Program	197
Requirements	15
Trace	202
Memory Models	193

MPLAB[®]-CXX Compiler User's Guide

Microchip Internet Web Site	6
Microcontroller Mode	193
Microprocessor Mode	194
Mnemonics	194
modulus (%)	99
MPASM	187, 194
MPLAB IDE	13, 16, 194
MPLAB Projects	37
MPLAB-CXX	13, 194
Basic Program Components	72
vs. C	71
MPLAB-ICD	194
MPLAB-ICE	13, 14, 194
MPLAB-SIM	13, 194
MPLIB	19, 25, 39, 192, 194
MPLINK	19, 25, 39, 44, 52, 58, 66, 126, 192, 195

N

near	73, 89, 90, 118, 126, 135
Nesting	
Interrupts	131
Structures	121
New Project	42, 56
Node Properties	44, 58
Nop()	124
NOT (!)	100
not equal to (!=)	100

O

Object Code	195
Object Files	21
Object Files, Precompiled	25, 39, 47, 61
Octal	86
Off-Chip Memory	196
One's complement (~)	101
Opcodes	196
Operators	99, 196
Arithmetic	99
Assignment	101
Bitwise	101
Conditional	103
Decrement	102
Increment	102
Precedence	105
Relational	100
Smooth	99
Optimization Tips	134, 138
OR ()	100
OR, Bitwise ()	101

P

Paged/Banked Data	88, 126
Pass Counter	196
Passing Arguments to Functions	97
Passing Pointers to Functions	117
Passing Variables	97
PCLATH	131
PIC17CXXX Instruction Set	157
PIC18CXXX Instruction Set	163
PICMASTER	196
PICmicro	199
PICSTART	14
PICSTART Plus	196
Pointers	115
ANSI C	148
Arithmetic	117
Post-decrement (c - -)	102
Post-increment (c++)	102
Precedence	197
Precedence of Operators	105
Precompiled Object Files	47, 61
Pre-decrement (- - c)	102
Pre-increment (++c)	102
Preprocessor Directives	76
ANSI C	149
PRO MATE	14, 197
Processor Header File	72
PRODH	132, 133
PRODL	132, 133
Program Control Statements	106
Program Counter	197
Program Memory	197
Programmer	197
Project	197
Project Nodes	44, 58
Project Window	53, 67
Project, MPLAB	37
Prototyping, Functions	96

Q

Qualifier	197
-----------------	-----

R

Radix	197
RAM	
Address Spaces	114
Pointers	114, 116, 117
ram	73, 113, 126, 135
README File	4, 14

Real-Time	198	Size	128, 135
Recursive Functions	97	Start Up Code	25, 28, 39, 47, 125, 129, 136
References	4	STARTUP() (_STARTUP())	129
register	89	static	89
Register Definitions	25, 28, 39, 47	Static Strings	114
Register Definitions File	124, 127, 135	Stimulus	200
Registers		Stopwatch	200
ANSI C	148	Storage Class	
Relational Operators	100	extern	92
Relocatable Section	198	static	92
Reset ()	124	volatile	93
return	98	Strings	113
Returning Values from Functions	98	struct	119
Right Shift (>>)	101	Structures	119
Rlcf ()	124	ANSI C	148
Rlncf ()	124	Nested	121
ROM		Swapf ()	124
Address Spaces	114	switch	109
Pointers	114, 116, 117	switch, ANSI C	149
rom	73, 113, 126, 135	Symbol	200
String	115	System Button	201
Rrcf ()	124	System Requirements, Host Computer	15
S		System Window Control	201
Section	198	T	
Absolute	187	Target	201
Assigned	188	TBLPTR	134
Relocatable	198	Trace	202
Shared	198	Trace Memory	202
Unassigned	202	Troubleshooting	5, 51, 65
SFRs	124, 127, 135	Tutorial, Using MPLAB-C17 with MPLAB	40
Shared Section	198	Tutorial, Using MPLAB-C18 with MPLAB	54
short	89	typedef	95
signed	89	U	
Simulator	198	Unassigned Section	202
Simulator <i>See MPLAB-SIM</i>		Uninitialized Data	202
Single Step	199	Unions	120
Skew	199	ANSI C	148
Skid	199	unsigned	89
Sleep ()	124	Updates	3
Source Code	21, 25, 39	USE_INITDATA	130
Directory	18	USE_STARTUP	129
Source Code, Assembly	199	V	
Source Code, C	199	Variables	88
Special Function Registers	124, 127, 135	Variables, Declaration	90
src directory	18	void	88, 89
Stack, Hardware	200	volatile	127, 128
Stack, Software	125, 128, 135, 200		
Initialization	130, 137		

MPLAB[®]-CXX Compiler User's Guide

W

Watch Dog Timer	202
Watch Window	202
Watchdog Timer (WDT)	124
WDT	202
while	108, 109
WREG	131, 133
WWW Address	6

X

XOR, Bitwise (^)	101
------------------------	-----

NOTES:



WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office

Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-786-7200 Fax: 480-786-7277
Technical Support: 480-786-7627
Web Address: <http://www.microchip.com>

Atlanta

Microchip Technology Inc.
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston

Microchip Technology Inc.
5 Mount Royal Avenue
Marlborough, MA 01752
Tel: 508-480-9990 Fax: 508-480-8575

Chicago

Microchip Technology Inc.
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas

Microchip Technology Inc.
4570 Westgrove Drive, Suite 160
Addison, TX 75248
Tel: 972-818-7423 Fax: 972-818-2924

Dayton

Microchip Technology Inc.
Two Prestige Place, Suite 150
Miamisburg, OH 45342
Tel: 937-291-1654 Fax: 937-291-9175

Detroit

Microchip Technology Inc.
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Los Angeles

Microchip Technology Inc.
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York

Microchip Technology Inc.
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 631-273-5305 Fax: 631-273-5335

San Jose

Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

AMERICAS (continued)

Toronto

Microchip Technology Inc.
5925 Airport Road, Suite 200
Mississauga, Ontario L4V 1W1, Canada
Tel: 905-405-6279 Fax: 905-405-6253

ASIA/PACIFIC

Beijing

Microchip Technology, Beijing
Unit 915, 6 Chaoyangmen Bei Dajie
Dong Erhuan Road, Dongcheng District
New China Hong Kong Manhattan Building
Beijing 100027 PRC
Tel: 86-10-85282100 Fax: 86-10-85282104

Hong Kong

Microchip Asia Pacific
Unit 2101, Tower 2
Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2-401-1200 Fax: 852-2-401-3431

India

Microchip Technology Inc.
India Liaison Office
No. 6, Legacy, Convent Road
Bangalore 560 025, India
Tel: 91-80-229-0061 Fax: 91-80-229-0062

Japan

Microchip Technology Intl. Inc.
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
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Shanghai

Microchip Technology
Unit B701, Far East International Plaza,
No. 317, Xianxia Road
Shanghai, 200051 P.R.C.
Tel: 86-21-6275-5700 Fax: 86-21-6275-5060

ASIA/PACIFIC (continued)

Singapore

Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore 188980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan, R.O.C

Microchip Technology Taiwan
10F-1C 207
Tung Hua North Road
Taipei, Taiwan, ROC
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Denmark

Microchip Technology Denmark ApS
Regus Business Centre
Lautrup høj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France

Arizona Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - 1er Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany

Arizona Microchip Technology GmbH
Gustav-Heinemann-Ring 125
D-81739 München, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

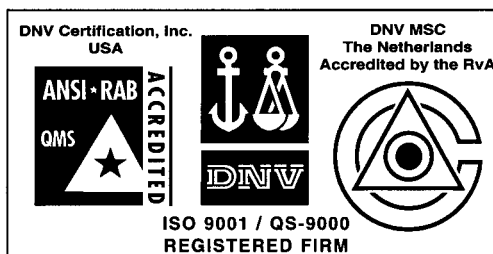
Italy

Arizona 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

Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5858 Fax: 44-118 921-5835

01/21/00



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

All rights reserved. © 2000 Microchip Technology Incorporated. Printed in the USA. 2/00 Printed on recycled paper.

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, except as maybe explicitly expressed herein, under any intellectual property rights. The Microchip logo and name are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. All rights reserved. All other trademarks mentioned herein are the property of their respective companies.