



MPLAB[®] XC16 C Compiler

User's Guide

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

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

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

Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer's risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Trademarks

The Microchip name and logo, the Microchip logo, dsPIC, FlashFlex, KEELQ, KEELQ logo, MPLAB, PIC, PICmicro, PICSTART, PIC³² logo, rfPIC, SST, SST Logo, SuperFlash and UNI/O are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

FilterLab, Hampshire, HI-TECH C, Linear Active Thermistor, MTP, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Silicon Storage Technology is a registered trademark of Microchip Technology Inc. in other countries.


Analog-for-the-Digital Age, Application Maestro, BodyCom, chipKIT, chipKIT logo, CodeGuard, dsPICDEM, dsPICDEM.net, dsPICworks, dsSPEAK, ECAN, ECONOMONITOR, FanSense, HI-TIDE, In-Circuit Serial Programming, ICSP, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, mTouch, Omniscent Code Generation, PICC, PICC-18, PICDEM, PICDEM.net, PICkit, PICtail, REAL ICE, rLAB, Select Mode, SQI, Serial Quad I/O, Total Endurance, TSHARC, UniWinDriver, WiperLock, ZENA and Z-Scale are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

SQTP is a service mark of Microchip Technology Incorporated in the U.S.A.

GestIC and ULPP are registered trademarks of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

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

© 2012-2014, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.

ISBN: 978-1-63276-493-5

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

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

Table of Contents

Preface	9
Chapter 1. Compiler Overview	
1.1 Introduction	13
1.2 Device Description	13
1.3 Compiler Description and Documentation	13
1.4 Compiler and Other Development Tools	15
Chapter 2. Common C Interface	
2.1 Introduction	17
2.2 Background: The Desire for Portable Code	17
2.3 Using the CCI	20
2.4 ANSI Standard Refinement	21
2.5 ANSI Standard Extensions	29
2.6 Compiler Features	44
Chapter 3. How To's	
3.1 Introduction	45
3.2 Installing and Activating the Compiler	45
3.3 Invoking the Compiler	47
3.4 Writing Source Code	49
3.5 Getting My Application to Do What I Want	60
3.6 Understanding the Compilation Process	62
3.7 Fixing Code That Does Not Work	67
Chapter 4. XC16 Toolchain and MPLAB X IDE	
4.1 Introduction	69
4.2 MPLAB X IDE and Tools Installation	69
4.3 MPLAB X IDE Setup	70
4.4 MPLAB X IDE Projects	71
4.5 Project Setup	73
4.6 Project Example	81
Chapter 5. Compiler Command-Line Driver	
5.1 Introduction	85
5.2 Invoking the Compiler	86
5.3 The Compilation Sequence	88
5.4 Runtime Files	92
5.5 Compiler Output	93
5.6 Compiler Messages	94
5.7 Driver Option Descriptions	95
5.8 MPLAB X IDE Toolchain or MPLAB IDE Toolsuite Equivalents	118

Chapter 6. Device-Related Features

6.1 Introduction	119
6.2 Device Support	119
6.3 Device Header Files	119
6.4 Stack	120
6.5 Configuration Bit Access	121
6.6 Using SFRs	121
6.7 Bit-Reversed and Modulo Addressing	124

Chapter 7. Differences Between MPLAB XC16 and ANSI C

7.1 Divergence from the ANSI C Standard	125
7.2 Extensions to the ANSI C Standard	125
7.3 Implementation-Defined Behavior	125

Chapter 8. Supported Data Types and Variables

8.1 Introduction	127
8.2 Identifiers	127
8.3 Integer Data Types	128
8.4 Floating-Point Data Types	129
8.5 Fixed-Point Data Types	130
8.6 Structures and Unions	131
8.7 Pointer Types	133
8.8 Complex Data Types	135
8.9 Literal Constant Types and Formats	136
8.10 Standard Type Qualifiers	138
8.11 Compiler-Specific type Qualifiers	139
8.12 Variable Attributes	142

Chapter 9. Fixed-Point Arithmetic Support

9.1 Introduction	151
9.2 Enabling Fixed-Point Arithmetic Support	151
9.3 Data Types	151
9.4 Rounding	152
9.5 Division By Zero	152
9.6 External Definitions	153
9.7 Mixing C and Assembly Language Code	153

Chapter 10. Memory Allocation and Access

10.1 Introduction	155
10.2 Address Spaces	155
10.3 Variables in Data Space Memory	156
10.4 Variables in Program Space	163
10.5 Parallel Master Port Access	168
10.6 External Memory Access	170
10.7 Extended Data Space Access	174
10.8 Packing Data Stored in Flash	175
10.9 Allocation of Variables to Registers	177

10.10 Variables in EEPROM	177
10.11 Dynamic Memory Allocation	179
10.12 Memory Models	180
Chapter 11. Operators and Statements	
11.1 Introduction	183
11.2 Built-In Functions	183
11.3 Integral Promotion	183
Chapter 12. Register Usage	
12.1 Introduction	185
12.2 Register Variables	185
12.3 Changing Register Contents	186
Chapter 13. Functions	
13.1 Introduction	187
13.2 Writing Functions	187
13.3 Function Size Limits	196
13.4 Allocation of Function Code	196
13.5 Changing the Default Function Allocation	196
13.6 Inline Functions	197
13.7 Memory Models	198
13.8 Function Call Conventions	200
Chapter 14. Interrupts	
14.1 Introduction	203
14.2 Interrupt Operation	204
14.3 Writing an Interrupt Service Routine	205
14.4 Specifying the Interrupt Vector	207
14.5 Interrupt Service Routine Context Saving	208
14.6 Nesting Interrupts	208
14.7 Enabling/Disabling Interrupts	209
14.8 ISR Considerations	210
Chapter 15. Main, Runtime Startup and Reset	
15.1 Introduction	215
15.2 The main Function	215
15.3 Runtime Startup and Initialization	215
Chapter 16. Mixing C and Assembly Code	
16.1 Introduction	217
16.2 Mixing Assembly Language and C Variables and Functions	217
16.3 Using Inline Assembly Language	220
16.4 Predefined Assembly Macros	224
Chapter 17. Library Routines	
Chapter 18. Optimizations	
18.1 Introduction	227
18.2 Setting Optimization Levels	227
18.3 Optimization Feature Summary	228

Chapter 19. Preprocessing

19.1 Introduction	229
19.2 C Language Comments	229
19.3 Preprocessing Directives	229
19.4 Predefined Macro Names	230
19.5 Pragmas vs. Attributes	233

Chapter 20. Linking Programs

20.1 Introduction	235
20.2 Default Memory Spaces	235
20.3 Replacing Library Symbols	237
20.4 Linker-Defined Symbols	237
20.5 Default Linker Script	238

Appendix A. Implementation-Defined Behavior

A.1 Introduction	239
A.2 Translation	240
A.3 Environment	240
A.4 Identifiers	241
A.5 Characters	241
A.6 Integers	242
A.7 Floating Point	243
A.8 Arrays and Pointers	243
A.9 Registers	244
A.10 Structures, Unions, Enumerations and Bit-Fields	244
A.11 Qualifiers	244
A.12 Declarators	244
A.13 Statements	245
A.14 Preprocessing Directives	245
A.15 Library Functions	246
A.16 Signals	247
A.17 Streams and Files	248
A.18 tmpfile	249
A.19 errno	249
A.20 Memory	249
A.21 abort	249
A.22 exit	249
A.23 getenv	249
A.24 system	249
A.25 strerror	250

Appendix B. Embedded Compiler Compatibility Mode

B.1 Introduction	251
B.2 Compiling in Compatibility Mode	252
B.3 Syntax Compatibility	252
B.4 Data Type	253
B.5 Operator	253

Table of Contents

B.6 Extended Keywords	254
B.7 Intrinsic Functions	255
B.8 Pragmas	255
Appendix C. Diagnostics	
C.1 Introduction	257
C.2 Errors	257
C.3 Warnings	276
Appendix D. GNU Free Documentation License	
D.1 Preamble	297
D.2 Applicability and Definitions	297
D.3 Verbatim Copying	299
D.4 Copying in Quantity	299
D.5 Modifications	300
D.6 Combining Documents	301
D.7 Collections of Documents	301
D.8 Aggregation with Independent Works	301
D.9 Translation	302
D.10 Termination	302
D.11 Future Revisions of this License	302
D.12 Relicensing	303
Appendix E. ASCII Character Set	305
Appendix F. Deprecated Features	
F.1 Introduction	307
F.2 Predefined Constants	307
F.3 Variables in Specified Registers	308
F.4 Changing Non-Auto Variable Allocation	309
Appendix G. Built-in Functions	
G.1 Introduction	311
G.2 Built-In Function Descriptions	313
Appendix H. Document Revision History	
Support	343
Glossary	347
Index	367
Worldwide Sales and Service	378

MPLAB[®] XC16 C Compiler User's Guide

NOTES:

Preface

NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document.

For the most up-to-date information on development tools, see the MPLAB[®] X IDE or MPLAB IDE v8 Help. Select the Help menu and then “Topics” or “Help Contents” to open a list of available Help files.

For the most current PDFs, please refer to our web site (<http://www.microchip.com>). Documents are identified by “DSXXXXXXXA”, where “XXXXXXX” is the document number and “A” is the revision level of the document. This number is located on the bottom of each page, in front of the page number.

INTRODUCTION

MPLAB XC16 C Compiler documentation and support information is discussed in the sections below:

- Document Layout
- Conventions Used
- Recommended Reading

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 16-bit applications. The document layout is as follows:


- **Chapter 1. “Compiler Overview”** – describes the compiler, development tools and feature set.
- **Chapter 2. “Common C Interface”** – describes the common C interface that may be used to enhance code portability between MPLAB XC compilers.
- **Chapter 4. “XC16 Toolchain and MPLAB X IDE”** – explains the basics of how to setup and use the compiler and related tools with MPLAB X IDE.
- **Chapter 4. “XC16 Toolchain and MPLAB IDE v8”** – explains the basics of how to setup and use the compiler and related tools with MPLAB IDE v8.
- **Chapter 5. “Compiler Command-Line Driver”** – describes how to use the compiler from the command line.
- **Chapter 6. “Device-Related Features”** – describes the compiler header and register definition files, as well as how to use with SFRs.
- **Chapter 7. “Differences Between MPLAB XC16 and ANSI C”** – describes the differences between the C language supported by the compiler syntax and the standard ANSI-89 C.
- **Chapter 8. “Supported Data Types and Variables”** – describes the compiler integer, floating point and pointer data types.

- **Chapter 9. “Fixed-Point Arithmetic Support”** – explains fixed-point arithmetic support in the compiler.
- **Chapter 10. “Memory Allocation and Access”** – describes the compiler run-time model, including information on sections, initialization, memory models, the software stack and much more.
- **Chapter 11. “Operators and Statements”** – discusses operators and statements.
- **Chapter 12. “Register Usage”** – explains how to access and use SFRs.
- **Chapter 13. “Functions”** – details available functions.
- **Chapter 14. “Interrupts”** – describes how to use interrupts.
- **Chapter 15. “Main, Runtime Startup and Reset”** – describes important elements of C code.
- **Chapter 16. “Mixing C and Assembly Code”** – provides guidelines to using the compiler with 16-bit assembly language modules.
- **Chapter 17. “Library Routines”** – explains how to use libraries.
- **Chapter 18. “Optimizations”** – describes optimization options.
- **Chapter 19. “Preprocessing”** – details preprocessing operation.
- **Chapter 20. “Linking Programs”** – explains how linking works.
- **Appendix A. “Implementation-Defined Behavior”** – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- **Appendix B. “Embedded Compiler Compatibility Mode”** – details the compiler’s compatibility mode.
- **Appendix C. “Diagnostics”** – lists error and warning messages generated by the compiler.
- **Appendix D. “GNU Free Documentation License”** – usage license for the Free Software Foundation.
- **Appendix E. “ASCII Character Set”** – a table of the ASCII character set.
- **Appendix F. “Deprecated Features”** – details features that are considered obsolete.
- **Appendix G. “Built-in Functions”** – lists the built-in functions of the C compiler.
- **Appendix H. “Document Revision History”** – information previous and current revisions of this document.

CONVENTIONS USED

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	<i>MPLAB® XC16 C Compiler User's Guide</i>
	Emphasized text	...is the <i>only</i> compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	<u><i>File>Save</i></u>
Bold characters	A dialog button	Click OK
	A tab	Click the Power tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	Sample source code	<code>#define START</code>
	File names	<code>autoexec.bat</code>
	File paths	<code>c:\mcc18\h</code>
	Keywords	<code>_asm, _endasm, static</code>
	Command-line options	<code>-Opa+, -Opa-</code>
	Bit values	<code>0, 1</code>
	Constants	<code>0xFF, 'A'</code>
Italic Courier	A variable argument	<i>file.c</i> , where <i>file</i> can be any valid file name
Square brackets []	Optional arguments	<code>mpasmwin [options]</code> <code>file [options]</code>
Curly brackets and pipe character: { }	Choice of mutually exclusive arguments; an OR selection	<code>errorlevel {0 1}</code>
Ellipses...	Replaces repeated text	<code>var_name [, var_name...]</code>
	Represents code supplied by user	<code>void main (void)</code> <code>{ ... }</code>
Sidebar Text		
	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	<code>xmemory attribute</code>

RECOMMENDED READING

This documentation describes how to use the MPLAB XC16 C Compiler. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Release Notes (Readme Files)

For the latest information on Microchip tools, read the associated Release Notes (HTML files) included with the software.

MPLAB® XC16 Assembler, Linker and Utilities User's Guide (DS52106)

A guide to using the 16-bit assembler, object linker, object archiver/librarian and various utilities.

MPLAB XC16 C Compiler User's Guide (DS51284)

A guide to using the 16-bit C compiler. The 16-bit linker is used with this tool.

16-Bit Language Tools Libraries (DS51456)

A descriptive listing of libraries available for Microchip 16-bit devices. This includes standard (including math) libraries and C compiler built-in functions. DSP and 16-bit peripheral libraries are described in Release Notes provided with each peripheral library type.

Device-Specific Documentation

The Microchip website contains many documents that describe 16-bit device functions and features. Among these are:

- Individual and family data sheets
- Family reference manuals
- Programmer's reference manuals

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.

C Reference Manuals

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

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*, Second Edition. Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*, Revised Edition. Hayden Books, Indianapolis, Indiana 46268.

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Van Sickle, Ted., *Programming Microcontrollers in C*, First Edition. LLH Technology Publishing, Eagle Rock, Virginia 24085.

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The MPLAB XC16 C compiler is defined and described in the following topics:

- Device Description
- Compiler Description and Documentation
- Compiler and Other Development Tools

1.2 DEVICE DESCRIPTION

The MPLAB XC16 C compiler fully supports all Microchip 16-bit devices:

- The dsPIC[®] family of digital signal controllers combines the high performance required in digital signal processor (DSP) applications with standard microcontroller (MCU) features needed for embedded applications.
- The PIC24 family of MCUs are identical to the dsPIC DSCs with the exception that they do not have the digital signal controller module or that subset of instructions. They are a subset, and are high-performance MCUs intended for applications that do not require the power of the DSC capabilities.

1.3 COMPILER DESCRIPTION AND DOCUMENTATION

The MPLAB XC16 C compiler is a full-featured, optimizing compiler that translates standard ANSI C programs into 16-bit device assembly language source. The compiler also supports many command-line options and language extensions that allow full access to the 16-bit device hardware capabilities, and affords fine control of the compiler code generator.

The compiler is a port of the GNU Compiler Collection (GCC) compiler from the Free Software Foundation.

The compiler is available for several popular operating systems, including 32 and 64-bit Windows[®], Linux and Apple OS X.

The compiler can run in one of three operating modes: Free, Standard or PRO. The Standard and PRO operating modes are licensed modes and require an activation key and Internet connectivity to enable them. Free mode is available for unlicensed customers. The basic compiler operation, supported devices and available memory are identical across all modes. The modes only differ in the level of optimization employed by the compiler (see **Chapter 18. "Optimizations"**).

This key features of the compiler are discussed in the following sections.

1.3.1 ANSI C Standard

The compiler is a fully validated compiler that conforms to the ANSI C standard as defined by the ANSI specification (ANSI x3.159-1989) and described in Kernighan and Ritchie's *The C Programming Language* (second edition). The ANSI standard includes extensions to the original C definition that are now standard features of the language. These extensions enhance portability and offer increased capability. In addition, language extensions for dsPIC DSC embedded-control applications are included.

1.3.2 Optimization

The compiler uses a set of sophisticated optimization passes that employ many advanced techniques for generating efficient, compact code from C source. The optimization passes include high-level optimizations that are applicable to any C code, as well as 16-bit device-specific optimizations that take advantage of the particular features of the device architecture.

For more on optimizations, see **Chapter 18. “Optimizations”**.

1.3.3 ANSI Standard Library Support

The compiler is distributed with a complete ANSI C standard library. All library functions have been validated, and conform to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, time-keeping and math functions (trigonometric, exponential and hyperbolic). The standard I/O functions for file handling are also included, and, as distributed, they support full access to the host file system using the command-line simulator. The fully functional source code for the low-level file I/O functions is provided in the compiler distribution, and may be used as a starting point for applications that require this capability.

1.3.4 Flexible Memory Models

The compiler supports both large and small code and data models. The small code model takes advantage of more efficient forms of call and branch instructions, while the small data model supports the use of compact instructions for accessing data in SFR space.

The compiler supports two models for accessing constant data. The “constants in data” model uses data memory, which is initialized by the run-time library. The “constants in code” model uses program memory, which is accessed through the Program Space Visibility (PSV) window.

1.3.5 Attributes and Qualifiers

The compiler keyword `__attribute__` allows you to specify special attributes of variables, structure fields or functions. This keyword is followed by an attribute specification inside double parentheses, as in:

```
int last_mode __attribute__((persistent));
```

In other compilers, qualifiers are used to create qualified types:

```
persistent int last_mode;
```

The MPLAB XC16 C Compiler does have some non-standard qualifiers described in **Section 8.11 “Compiler-Specific type Qualifiers”**.

Generally speaking, qualifiers indicate how an object should be accessed, whereas attributes indicate where objects are to be located. Attributes also have many other purposes.

1.3.6 Compiler Driver

The compiler includes a powerful command-line driver program. Using the driver program, application programs can be compiled, assembled and linked in a single step.

1.3.7 Documentation

The compiler is supported under both the MPLAB X IDE and MPLAB IDE v8.xx and above.

Features that are unique to specific devices, and therefore specific compilers, are noted with a “DD” icon next to the section and text that identifies the specific devices to which the information applies (see the **Preface**).

1.4 COMPILER AND OTHER DEVELOPMENT TOOLS

The compiler works with many other Microchip tools including:

- MPLAB XC16 Assembler and Linker - see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.
- MPLAB X IDE and MPLAB IDE v8.xx
- The MPLAB SIM Simulator and MPLAB X Simulator
- Command-line MDB Simulator - see the *Microchip Debugger (MDB) User's Guide (DS52102)* located in:
`<MPLAB X IDE Installation Directory>docs`
- All Microchip debug tools and programmers
- Demonstration boards and Starter kits that support 16-bit devices

NOTES:

Chapter 2. Common C Interface

2.1 INTRODUCTION

The Common C Interface (CCI) is available with all MPLAB XC C compilers and is designed to enhance code portability between these compilers. For example, CCI-conforming code would make it easier to port from a PIC18 MCU using the MPLAB XC8 C compiler to a PIC24 MCU using the MPLAB XC16 C compiler.

The CCI assumes that your source code already conforms to the ANSI Standard. If you intend to use the CCI, it is your responsibility to write code that conforms. Legacy projects will need to be migrated to achieve conformance. A compiler option must also be set to ensure that the operation of the compiler is consistent with the interface when the project is built.

The following topics are examined in this chapter of the MPLAB XC16 C Compiler User's Guide:

- Background: The Desire for Portable Code
- Using the CCI
- ANSI Standard Refinement
- ANSI Standard Extensions
- Compiler Features

2.2 BACKGROUND: THE DESIRE FOR PORTABLE CODE

All programmers want to write portable source code.

Portability means that the same source code can be compiled and run in a different execution environment than that for which it was written. Rarely can code be one hundred percent portable, but the more tolerant it is to change, the less time and effort it takes to have it running in a new environment.

Embedded engineers typically think of code portability as being across target devices, but this is only part of the situation. The same code could be compiled for the same target but with a different compiler. Differences between those compilers might lead to the code failing at compile time or runtime, so this must be considered as well.

You may only write code for one target device and only use one brand of compiler, but if there is no regulation of the compiler's operation, simply updating your compiler version may change your code's behavior.

Code must be portable across targets, tools, and time to be truly flexible.

Clearly, this portability cannot be achieved by the programmer alone, since the compiler vendors can base their products on different technologies, implement different features and code syntax, or improve the way their product works. Many a great compiler optimization has broken many an unsuspecting project.

Standards for the C language have been developed to ensure that change is managed and code is more portable. The American National Standards Institute (ANSI) publishes standards for many disciplines, including programming languages. The ANSI C Standard is a universally adopted standard for the C programming language.

2.2.1 The ANSI Standard

The ANSI C Standard has to reconcile two opposing goals: freedom for compilers vendors to target new devices and improve code generation, with the known functional operation of source code for programmers. If both goals can be met, source code can be made portable.

The standard is implemented as a set of rules which detail not only the syntax that a conforming C program must follow, but the semantic rules by which that program will be interpreted. Thus, for a compiler to conform to the standard, it must ensure that a conforming C program functions as described by the standard.

The standard describes *implementation*, the set of tools and the runtime environment on which the code will run. If any of these change, e.g., you build for, and run on, a different target device, or if you update the version of the compiler you use to build, then you are using a different implementation.

The standard uses the term *behavior* to mean the external appearance or action of the program. It has nothing to do with how a program is encoded.

Since the standard is trying to achieve goals that could be construed as conflicting, some specifications appear somewhat vague. For example, the standard states that an `int` type must be able to hold at least a 16-bit value, but it does not go as far as saying what the size of an `int` actually is; and the action of right-shifting a signed integer can produce different results on different implementations; yet, these different results are still ANSI C compliant.

If the standard is too strict, device architectures may not allow the compiler to conform. But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would lose its effectiveness.

Case in point: The mid-range PIC[®] microcontrollers do not have a data stack. Because a compiler targeting this device cannot implement recursion, it (strictly speaking) cannot conform to the ANSI C Standard. This example illustrates a situation in which the standard is too strict for mid-range devices and tools.

The standard organizes source code whose behavior is not fully defined into groups that include the following behaviors:

Implementation-defined behavior

This is unspecified behavior where each implementation documents how the choice is made.

Unspecified behavior

The standard provides two or more possibilities and imposes no further requirements on which possibility is chosen in any particular instance.

Undefined behavior

This is behavior for which the standard imposes no requirements.

Code that strictly conforms to the standard does not produce output that is dependent on any unspecified, undefined, or implementation-defined behavior. The size of an `int`, which we used as an example earlier, falls into the category of behavior that is defined by implementation. That is to say, the size of an `int` is defined by which compiler is being used, how that compiler is being used, and the device that is being targeted.

All the MPLAB XC compilers conform to the ANSI X3.159-1989 Standard for programming languages (with the exception of the XC8 compiler's inability to allow recursion, as mentioned in the footnote). This is commonly called the C89 Standard. Some features from the later standard, C99, are also supported.

For freestanding implementations – or for what we typically call embedded applications – the standard allows non-standard extensions to the language, but obviously does not enforce how they are specified or how they work. When working so closely to the device hardware, a programmer needs a means of specifying device setup and interrupts, as well as utilizing the often complex world of small-device memory architectures. This cannot be offered by the standard in a consistent way.

While the ANSI C Standard provides a mutual understanding for programmers and compiler vendors, programmers need to consider the implementation-defined behavior of their tools and the probability that they may need to use extensions to the C language that are non-standard. Both of these circumstances can have an impact on code portability.

2.2.2 The Common C Interface

The Common C Interface (CCI) supplements the ANSI C Standard and makes it easier for programmers to achieve consistent outcomes on all Microchip devices when using any of the MPLAB XC C compilers.

It delivers the following improvements, all designed with portability in mind.

Refinement of the ANSI C Standard

The CCI documents specific behavior for some code in which actions are implementation-defined behavior under the ANSI C Standard. For example, the result of right-shifting a signed integer is fully defined by the CCI. Note that many implementation-defined items that closely couple with device characteristics, such as the size of an `int`, are not defined by the CCI.

Consistent syntax for non-standard extensions

The CCI non-standard extensions are mostly implemented using keywords with a uniform syntax. They replace keywords, macros and attributes that are the native compiler implementation. The interpretation of the keyword may differ across each compiler, and any arguments to the keywords may be device specific.

Coding guidelines

The CCI may indicate advice on how code should be written so that it can be ported to other devices or compilers. While you may choose not to follow the advice, it will not conform to the CCI.

2.3 USING THE CCI

The CCI allows enhanced portability by refining implementation-defined behavior and standardizing the syntax for extensions to the language.

The CCI is something you choose to follow and put into effect, thus it is relevant for new projects, although you may choose to modify existing projects so they conform.

For your project to conform to the CCI, you must do the following things.

Enable the CCI

Select the MPLAB IDE widget Use CCI Syntax in your project, or use the command-line option that is equivalent.

Include <xc.h> in every module

Some CCI features are only enabled if this header is seen by the compiler.

Ensure ANSI compliance

Code that does not conform to the ANSI C Standard does not conform to the CCI.

Observe refinements to ANSI by the CCI

Some ANSI implementation-defined behavior is defined explicitly by the CCI.

Use the CCI extensions to the language

Use the CCI extensions rather than the native language extensions

The next sections detail specific items associated with the CCI. These items are segregated into those that refine the standard, those that deal with the ANSI C Standard extensions, and other miscellaneous compiler options and usage. Guidelines are indicated with these items.

If any implementation-defined behavior or any non-standard extension is not discussed in this document, then it is not part of the CCI. For example, GCC case ranges, label addresses and 24-bit `short long` types are not part of the CCI. Programs which use these features do not conform to the CCI. The compiler may issue a warning or error to indicate when you use a non-CCI feature and the CCI is enabled.

2.4 ANSI STANDARD REFINEMENT

The following topics describe how the CCI refines the implementation-defined behaviors outlined in the ANSI C Standard.

2.4.1 Source File Encoding

Under the CCI, a source file must be written using characters from the 7-bit ASCII set. Lines may be terminated using a *line feed* ('\n') or *carriage return* ('\r') that is immediately followed by a *line feed*. Escaped characters may be used in character constants or string literals to represent extended characters not in the basic character set.

2.4.1.1 EXAMPLE

The following shows a string constant being defined that uses escaped characters.

```
const char myName[] = "Bj\370rk\n";
```

2.4.1.2 DIFFERENCES

All compilers have used this character set.

2.4.1.3 MIGRATION TO THE CCI

No action required.

2.4.2 The Prototype for `main`

The prototype for the `main()` function is

```
int main(void);
```

2.4.2.1 EXAMPLE

The following shows an example of how `main()` might be defined

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

2.4.2.2 DIFFERENCES

The 8-bit compilers used a `void` return type for this function.

2.4.2.3 MIGRATION TO THE CCI

Each program has one definition for the `main()` function. Confirm the return type for `main()` in all projects previously compiled for 8-bit targets.

2.4.3 Header File Specification

Header file specifications that use directory separators do not conform to the CCI.

2.4.3.1 EXAMPLE

The following example shows two conforming include directives.

```
#include <usb_main.h>
#include "global.h"
```

2.4.3.2 DIFFERENCES

Header file specifications that use directory separators have been allowed in previous versions of all compilers. Compatibility problems arose when Windows-style separators “\” were used and the code compiled under other host operating systems. Under the CCI, no directory specifiers should be used.

2.4.3.3 MIGRATION TO THE CCI

Any `#include` directives that use directory separators in the header file specifications should be changed. Remove all but the header file name in the directive. Add the directory path to the compiler's include search path or MPLAB IDE equivalent. This will force the compiler to search the directories specified with this option.

For example, the following code:

```
#include <inc/lcd.h>
```

should be changed to:

```
#include <lcd.h>
```

and the path to the `inc` directory added to the compiler's header search path in your MPLAB IDE project properties, or on the command-line as follows:

```
-Ilcd
```

2.4.4 Include Search Paths

When you include a header file under the CCI, the file should be discoverable in the paths searched by the compiler detailed below.

For any header files specified in angle bracket delimiters `< >`, the search paths should be those specified by `-I` options (or the equivalent MPLAB IDE option), then the standard compiler include directories. The `-I` options are searched in the order in which they are specified.

For any file specified in quote characters `" "`, the search paths should first be the current working directory. In the case of an MPLAB X project, the current working directory is the directory in which the C source file is located. If unsuccessful, the search paths should be the same directories searched when the header files is specified in angle bracket delimiters.

Any other options to specify search paths for header files do not conform to the CCI.

2.4.4.1 EXAMPLE

If including a header file as in the following directive

```
#include "myGlobals.h"
```

The header file should be locatable in the current working directory, or the paths specified by any `-I` options, or the standard compiler directories. If it is located elsewhere, this does not conform to the CCI.

2.4.4.2 DIFFERENCES

The compiler operation under the CCI is not changed. This is purely a coding guide line.

2.4.4.3 MIGRATION TO THE CCI

Remove any option that specifies header file search paths other than the `-I` option (or the equivalent MPLAB IDE option), and use the `-I` option in place of this. Ensure the header file can be found in the directories specified in this section.

2.4.5 The number of Significant Initial Characters in an Identifier

At least the first 255 characters in an identifier (internal and external) are significant. This extends upon the requirement of the ANSI C Standard which states a lower number of significant characters are used to identify an object.

2.4.5.1 EXAMPLE

The following example shows two poorly named variables, but names which are considered unique under the CCI.

```
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningFast;
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningSlow;
```

2.4.5.2 DIFFERENCES

Former 8-bit compilers used 31 significant characters by default, but an option allowed this to be extended.

The 16- and 32-bit compilers did not impose a limit on the number of significant characters.

2.4.5.3 MIGRATION TO THE CCI

No action required. You may take advantage of the less restrictive naming scheme.

2.4.6 Sizes of Types

The sizes of the basic C types, for example `char`, `int` and `long`, are *not* fully defined by the CCI. These types, by design, reflect the size of registers and other architectural features in the target device. They allow the device to efficiently access objects of this type. The ANSI C Standard does, however, indicate minimum requirements for these types, as specified in `<limits.h>`.

If you need fixed-size types in your project, use the types defined in `<stdint.h>`, e.g., `uint8_t` or `int16_t`. These types are consistently defined across all XC compilers, even outside of the CCI.

Essentially, the C language offers a choice of two groups of types: those that offer sizes and formats that are tailored to the device you are using; or those that have a fixed size, regardless of the target.

2.4.6.1 EXAMPLE

The following example shows the definition of a variable, `native`, whose size will allow efficient access on the target device; and a variable, `fixed`, whose size is clearly indicated and remains fixed, even though it may not allow efficient access on every device.

```
int native;
int16_t fixed;
```

2.4.6.2 DIFFERENCES

This is consistent with previous types implemented by the compiler.

2.4.6.3 MIGRATION TO THE CCI

If you require a C type that has a fixed size, regardless of the target device, use one of the types defined by `<stdint.h>`.

2.4.7 Plain char Types

The type of a plain `char` is `unsigned char`. It is generally recommended that all definitions for the `char` type explicitly state the signedness of the object.

2.4.7.1 EXAMPLE

The following example

```
char foobar;
```

defines an unsigned char object called foobar.

2.4.7.2 DIFFERENCES

The 8-bit compilers have always treated plain char as an unsigned type.

The 16- and 32-bit compilers used signed char as the default plain char type. The -funsigned-char option on those compilers changed the default type to be unsigned char.

2.4.7.3 MIGRATION TO THE CCI

Any definition of an object defined as a plain char and using the 16- or 32-bit compilers needs review. Any plain char that was intended to be a signed quantity should be replaced with an explicit definition, for example.

```
signed char foobar;
```

You may use the -funsigned-char option on XC16/32 to change the type of plain char, but since this option is not supported on XC8, the code is not strictly conforming.

2.4.8 Signed Integer Representation

The value of a signed integer is determined by taking the two's complement of the integer.

2.4.8.1 EXAMPLE

The following shows a variable, test, that is assigned the value -28 decimal.

```
signed char test = 0xE4;
```

2.4.8.2 DIFFERENCES

All compilers have represented signed integers in the way described in this section.

2.4.8.3 MIGRATION TO THE CCI

No action required.

2.4.9 Integer conversion

When converting an integer type to a signed integer of insufficient size, the original value is truncated from the most-significant bit to accommodate the target size.

2.4.9.1 EXAMPLE

The following shows an assignment of a value that will be truncated.

```
signed char destination;  
unsigned int source = 0x12FE;  
destination = source;
```

Under the CCI, the value of destination after the alignment will be -2 (i.e., the bit pattern 0xFE).

2.4.9.2 DIFFERENCES

All compilers have performed integer conversion in an identical fashion to that described in this section.

2.4.9.3 MIGRATION TO THE CCI

No action required.

2.4.10 Bit-wise Operations on Signed Values

Bitwise operations on signed values act on the two's complement representation, including the sign bit. See also **Section 2.4.11 “Right-shifting Signed Values”**.

2.4.10.1 EXAMPLE

The following shows an example of a negative quantity involved in a bitwise AND operation.

```
signed char output, input = -13;
output = input & 0x7E;
```

Under the CCI, the value of `output` after the assignment will be 0x72.

2.4.10.2 DIFFERENCES

All compilers have performed bitwise operations in an identical fashion to that described in this section.

2.4.10.3 MIGRATION TO THE CCI

No action required.

2.4.11 Right-shifting Signed Values

Right-shifting a signed value will involve sign extension. This will preserve the sign of the original value.

2.4.11.1 EXAMPLE

The following shows an example of a negative quantity involved in a bitwise AND operation.

```
signed char input, output = -13;
output = input >> 3;
```

Under the CCI, the value of `output` after the assignment will be -2 (i.e., the bit pattern 0xFE).

2.4.11.2 DIFFERENCES

All compilers have performed right shifting as described in this section.

2.4.11.3 MIGRATION TO THE CCI

No action required.

2.4.12 Conversion of Union Member Accessed Using Member With Different Type

If a union defines several members of different types and you use one member identifier to try to access the contents of another (whether any conversion is applied to the result) is implementation-defined behavior in the standard. In the CCI, no conversion is applied and the bytes of the union object are interpreted as an object of the type of the member being accessed, without regard for alignment or other possible invalid conditions.

2.4.12.1 EXAMPLE

The following shows an example of a union defining several members.

```
union {
    signed char code;
    unsigned int data;
    float offset;
} foobar;
```

Code that attempts to extract `offset` by reading `data` is not guaranteed to read the correct value.

```
float result;
result = foobar.data;
```

2.4.12.2 DIFFERENCES

All compilers have not converted union members accessed via other members.

2.4.12.3 MIGRATION TO THE CCI

No action required.

2.4.13 Default Bit-field int Type

The type of a bit-field specified as a plain `int` will be identical to that of one defined using `unsigned int`. This is quite different to other objects where the types `int`, `signed` and `signed int` are synonymous. It is recommended that the signedness of the bit-field be explicitly stated in all bit-field definitions.

2.4.13.1 EXAMPLE

The following shows an example of a structure tag containing bit-fields which are unsigned integers and with the size specified.

```
struct OUTPUTS {
    int direction :1;
    int parity    :3;
    int value     :4;
};
```

2.4.13.2 DIFFERENCES

The 8-bit compilers have previously issued a warning if type `int` was used for bit-fields, but would implement the bit-field with an `unsigned int` type.

The 16- and 32-bit compilers have implemented bit-fields defined using `int` as having a `signed int` type, unless the option `-funsigned-bitfields` was specified.

2.4.13.3 MIGRATION TO THE CCI

Any code that defines a bit-field with the plain `int` type should be reviewed. If the intention was for these to be signed quantities, then the type of these should be changed to `signed int`, for example, in:

```
struct WAYPT {
    int log          :3;
    int direction    :4;
};
```

the bit-field type should be changed to `signed int`, as in:

```
struct WAYPT {
    signed int log      :3;
    signed int direction :4;
};
```

2.4.14 Bit-fields Straddling a Storage Unit Boundary

Whether a bit-field can straddle a storage unit boundary is implementation-defined behavior in the standard. In the CCI, bit-fields will not straddle a storage unit boundary; a new storage unit will be allocated to the structure, and padding bits will fill the gap.

Note that the size of a storage unit differs with each compiler as this is based on the size of the base data type (e.g., `int`) from which the bit-field type is derived. On 8-bit compilers this unit is 8-bits in size; for 16-bit compilers, it is 16 bits; and for 32-bit compilers, it is 32 bits in size.

2.4.14.1 EXAMPLE

The following shows a structure containing bit-fields being defined.

```
struct {
    unsigned first  : 6;
    unsigned second :6;
} order;
```

Under the CCI and using XC8, the storage allocation unit is byte sized. The bit-field `second`, will be allocated a new storage unit since there are only 2 bits remaining in the first storage unit in which `first` is allocated. The size of this structure, `order`, will be 2 bytes.

2.4.14.2 DIFFERENCES

This allocation is identical with that used by all previous compilers.

2.4.14.3 MIGRATION TO THE CCI

No action required.

2.4.15 The Allocation Order of Bits-field

The memory ordering of bit-fields into their storage unit is not specified by the ANSI C Standard. In the CCI, the first bit defined will be the least significant bit of the storage unit in which it will be allocated.

2.4.15.1 EXAMPLE

The following shows a structure containing bit-fields being defined.

```
struct {  
    unsigned lo   : 1;  
    unsigned mid  : 6;  
    unsigned hi   : 1;  
} foo;
```

The bit-field `lo` will be assigned the Least Significant bit of the storage unit assigned to the structure `foo`. The bit-field `mid` will be assigned the next 6 Least Significant bits, and `hi`, the Most Significant bit of that same storage unit byte.

2.4.15.2 DIFFERENCES

This is identical with the previous operation of all compilers.

2.4.15.3 MIGRATION TO THE CCI

No action required.

2.4.16 The NULL macro

The `NULL` macro is defined in `<stddef.h>`; however, its definition is implementation-defined behavior. Under the CCI, the definition of `NULL` is the expression `(0)`.

2.4.16.1 EXAMPLE

The following shows a pointer being assigned a null pointer constant via the `NULL` macro.

```
int * ip = NULL;
```

The value of `NULL`, `(0)`, is implicitly cast to the destination type.

2.4.16.2 DIFFERENCES

The 32-bit compilers previously assigned `NULL` the expression `((void *)0)`.

2.4.16.3 MIGRATION TO THE CCI

No action required.

2.4.17 Floating-point sizes

Under the CCI, floating-point types must not be smaller than 32 bits in size.

2.4.17.1 EXAMPLE

The following shows the definition for `outY`, which will be at least 32-bit in size.

```
float outY;
```

2.4.17.2 DIFFERENCES

The 8-bit compilers have allowed the use of 24-bit `float` and `double` types.

2.4.17.3 MIGRATION TO THE CCI

When using 8-bit compilers, the `float` and `double` type will automatically be made 32 bits in size once the CCI mode is enabled. Review any source code that may have assumed a `float` or `double` type and may have been 24 bits in size.

No migration is required for other compilers.

2.5 ANSI STANDARD EXTENSIONS

The following topics describe how the CCI provides device-specific extensions to the standard.

2.5.1 Generic Header File

A single header file `<xc.h>` must be used to declare all compiler- and device-specific types and SFRs. You *must* include this file into every module to conform with the CCI. Some CCI definitions depend on this header being seen.

2.5.1.1 EXAMPLE

The following shows this header file being included, thus allowing conformance with the CCI, as well as allowing access to SFRs.

```
#include <xc.h>
```

2.5.1.2 DIFFERENCES

Some 8-bit compilers used `<htc.h>` as the equivalent header. Previous versions of the 16- and 32-bit compilers used a variety of headers to do the same job.

2.5.1.3 MIGRATION TO THE CCI

Change:

```
#include <htc.h>
```

used previously in 8-bit compiler code, or family-specific header files as in the following examples:

```
#include <p32xxxx.h>
#include <p30fxxxx.h>
#include <p33Fxxxx.h>
#include <p24Fxxxx.h>
#include "p30f6014.h"
```

to:

```
#include <xc.h>
```

2.5.2 Absolute addressing

Variables and functions can be placed at an absolute address by using the `__at()` construct qualifier. Stack-based (`auto` and parameter) variables cannot use the `__at()` specifier.

2.5.2.1 EXAMPLE

The following shows two variables and a function being made absolute.

```
const char keys[] __at(123) = { 'r', 's', 'u', 'd' };
__at(0x1000) int modify(int x) {
    return x * 2 + 3;
}
```

2.5.2.2 DIFFERENCES

The 8-bit compilers have used an `@` symbol to specify an absolute address.

The 16- and 32-bit compilers have used the `address` attribute to specify an object's address.

2.5.2.3 MIGRATION TO THE CCI

Avoid making objects and functions absolute if possible.

In XC8, change absolute object definitions such as the following example:

```
int scanMode @ 0x200;
```

to:

```
int scanMode __at(0x200);
```

In XC16/32, change code such as:

```
int scanMode __attribute__((address(0x200)));
```

to:

```
int scanMode __at(0x200);
```

2.5.2.4 CAVEATS

If the `__at()` and `__section()` specifiers are both applied to an object when using XC8, the `__section()` specifier is currently ignored.

2.5.3 Far Objects and Functions

The `__far` qualifier may be used to indicate that variables or functions may be located in 'far memory'. Exactly what constitutes far memory is dependent on the target device, but it is typically memory that requires more complex code to access. Expressions involving far-qualified objects may generate slower and larger code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not have such memory implemented, in which case, use of this qualifier will be ignored. Stack-based (`auto` and `parameter`) variables cannot use the `__far` specifier.

2.5.3.1 EXAMPLE

The following shows a variable and function qualified using `__far`.

```
__far int serialNo;  
__far int ext_getCond(int selector);
```

2.5.3.2 DIFFERENCES

The 8-bit compilers have used the qualifier `far` to indicate this meaning. Functions could not be qualified as `far`.

The 16-bit compilers have used the `far` attribute with both variables and functions.

The 32-bit compilers have used the `far` attribute with functions, only.

2.5.3.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the `far` qualifier, as in the following example:

```
far char template[20];
```

to `__far`, i.e., `__far char template[20];`

In the 16- and 32-bit compilers, change any occurrence of the `far` attribute, as in the following

```
void bar(void) __attribute__((far));  
int tblIdx __attribute__((far));
```

to

```
void __far bar(void);  
int __far tblIdx;
```

2.5.3.4 CAVEATS

None.

2.5.4 Near Objects

The `__near` qualifier may be used to indicate that variables or functions may be located in 'near memory'. Exactly what constitutes near memory is dependent on the target device, but it is typically memory that can be accessed with less complex code. Expressions involving near-qualified objects may be faster and result in smaller code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not have such memory implemented, in which case, use of this qualifier will be ignored. Stack-based (`auto` and parameter) variables cannot use the `__near` specifier.

2.5.4.1 EXAMPLE

The following shows a variable and function qualified using `__near`.

```
__near int serialNo;  
__near int ext_getCond(int selector);
```

2.5.4.2 DIFFERENCES

The 8-bit compilers have used the qualifier `near` to indicate this meaning. Functions could not be qualified as `near`.

The 16-bit compilers have used the `near` attribute with both variables and functions.

The 32-bit compilers have used the `near` attribute for functions, only.

2.5.4.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the `near` qualifier, as in the following example:

```
near char template[20];
```

to `__near`, i.e., `__near char template[20];`

In 16- and 32-bit compilers, change any occurrence of the `near` attribute, as in the following

```
void bar(void) __attribute__((near));
```

```
int tblIdx __attribute__((near));
```

to

```
void __near bar(void);
```

```
int __near tblIdx;
```

2.5.4.4 CAVEATS

None.

2.5.5 Persistent Objects

The `__persistent` qualifier may be used to indicate that variables should not be cleared by the runtime startup code.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

2.5.5.1 EXAMPLE

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

2.5.5.2 DIFFERENCES

The 8-bit compilers have used the qualifier, `persistent`, to indicate this meaning.

The 16- and 32-bit compilers have used the `persistent` attribute with variables to indicate they were not to be cleared.

2.5.5.3 MIGRATION TO THE CCI

With 8-bit compilers, change any occurrence of the `persistent` qualifier, as in the following example:

```
persistent char template[20];
```

to `__persistent`, i.e., `__persistent char template[20];`

For the 16- and 32-bit compilers, change any occurrence of the `persistent` attribute, as in the following

```
int tblIdx __attribute__((persistent));
```

to

```
int __persistent tblIdx;
```

2.5.5.4 CAVEATS

None.

2.5.6 X and Y Data Objects

The `__xdata` and `__ydata` qualifiers may be used to indicate that variables may be located in special memory regions. Exactly what constitutes X and Y memory is dependent on the target device, but it is typically memory that can be accessed independently on separate buses. Such memory is often required for some DSP instructions.

Use the native keywords discussed in the Differences section to look up information on the semantics of these qualifiers.

Some devices may not have such memory implemented; in which case, use of these qualifiers will be ignored.

2.5.6.1 EXAMPLE

The following shows a variable qualified using `__xdata`, as well as another variable qualified with `__ydata`.

```
__xdata char data[16];  
__ydata char coeffs[4];
```

2.5.6.2 DIFFERENCES

The 16-bit compilers have used the `xmemory` and `ymemory` space attribute with variables.

Equivalent specifiers have never been defined for any other compiler.

2.5.6.3 MIGRATION TO THE CCI

For 16-bit compilers, change any occurrence of the `space` attributes `xmemory` or `ymemory`, as in the following example:

```
char __attribute__((space(xmemory))) template[20];  
to __xdata, or __ydata, i.e., __xdata char template[20];
```

2.5.6.4 CAVEATS

None.

2.5.7 Banked Data Objects

The `__bank(num)` qualifier may be used to indicate that variables may be located in a particular data memory bank. The number, `num`, represents the bank number. Exactly what constitutes banked memory is dependent on the target device, but it is typically a subdivision of data memory to allow for assembly instructions with a limited address width field.

Use the native keywords discussed in the Differences section to look up information on the semantics of these qualifiers.

Some devices may not have banked data memory implemented, in which case, use of this qualifier will be ignored. The number of data banks implemented will vary from one device to another.

2.5.7.1 EXAMPLE

The following shows a variable qualified using `__bank()`.

```
__bank(0) char start;  
__bank(5) char stop;
```

2.5.7.2 DIFFERENCES

The 8-bit compilers have used the four qualifiers `bank0`, `bank1`, `bank2` and `bank3` to indicate the same, albeit more limited, memory placement.

Equivalent specifiers have never been defined for any other compiler.

2.5.7.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the `bankx` qualifiers, as in the following example:

```
bank2 int logEntry;  
to __bank(, i.e., __bank(2) int logEntry;
```

2.5.7.4 CAVEATS

This feature is not yet implemented in the MPLAB XC8 compiler.

2.5.8 Alignment of Objects

The `__align(alignment)` specifier may be used to indicate that variables must be aligned on a memory address that is a multiple of the alignment specified. The alignment term must be a power of two. Positive values request that the object's start address be aligned; negative values imply the object's end address be aligned.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

2.5.8.1 EXAMPLE

The following shows variables qualified using `__align()` to ensure they end on an address that is a multiple of 8, and start on an address that is a multiple of 2, respectively.

```
__align(-8) int spacer;  
__align(2) char coeffs[6];
```

2.5.8.2 DIFFERENCES

An alignment feature has never been implemented on 8-bit compilers.

The 16- and 32-bit compilers used the `aligned` attribute with variables.

2.5.8.3 MIGRATION TO THE CCI

For 16- and 32-bit compilers, change any occurrence of the `aligned` attribute, as in the following example:

```
char __attribute__((aligned(4))) mode;  
to __align, i.e., __align(4) char mode;
```

2.5.8.4 CAVEATS

This feature is not yet implemented on XC8.

2.5.9 EEPROM Objects

The `__eeprom` qualifier may be used to indicate that variables should be positioned in EEPROM.

Use the native keywords discussed in the Differences section to look up information on the semantics of this qualifier.

Some devices may not implement EEPROM. Use of this qualifier for such devices will generate a warning. Stack-based (`auto` and `parameter`) variables cannot use the `__eeprom` specifier.

2.5.9.1 EXAMPLE

The following shows a variable qualified using `__eeprom`.

```
__eeprom int serialNos[4];
```

2.5.9.2 DIFFERENCES

The 8-bit compilers have used the qualifier, `eeprom`, to indicate this meaning for some devices.

The 16-bit compilers have used the `space` attribute to allocate variables to the memory space used for EEPROM.

2.5.9.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the `eeprom` qualifier, as in the following example:

```
eeprom char title[20];
```

to `__eeprom`, i.e., `__eeprom char title[20];`

For 16-bit compilers, change any occurrence of the `eedata space` attribute, as in the following

```
int mainSw __attribute__ ((space(eedata)));
```

to

```
int __eeprom mainSw;
```

2.5.9.4 CAVEATS

XC8 does not implement the `__eeprom` qualifiers for any PIC18 devices; this qualifier will work as expected for other 8-bit devices.

2.5.10 Interrupt Functions

The `__interrupt(type)` specifier may be used to indicate that a function is to act as an interrupt service routine. The *type* is a comma-separated list of keywords that indicate information about the interrupt function.

The current interrupt types are:

<empty>

Implement the default interrupt function

low_priority

The interrupt function corresponds to the low priority interrupt source (XC8 – PIC18 only)

high_priority

The interrupt function corresponds to the high priority interrupt source (XC8)

save(symbol-list)

Save on entry and restore on exit the listed symbols (XC16)

irq(irqid)

Specify the interrupt vector associated with this interrupt (XC16)

altirq(altirqid)

Specify the alternate interrupt vector associated with this interrupt (XC16)

preprologue(asm)

Specify assembly code to be executed before any compiler-generated interrupt code (XC16)

shadow

Allow the ISR to utilize the shadow registers for context switching (XC16)

auto_psv

The ISR will set the PSVPAG register and restore it on exit (XC16)

no_auto_psv

The ISR will not set the PSVPAG register (XC16)

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Some devices may not implement interrupts. Use of this qualifier for such devices will generate a warning. If the argument to the `__interrupt` specifier does not make sense for the target device, a warning or error will be issued by the compiler.

2.5.10.1 EXAMPLE

The following shows a function qualified using `__interrupt`.

```
__interrupt(low_priority) void getData(void) {
    if (TMR0IE && TMR0IF) {
        TMR0IF=0;
        ++tick_count;
    }
}
```

2.5.10.2 DIFFERENCES

The 8-bit compilers have used the `interrupt` and `low_priority` qualifiers to indicate this meaning for some devices. Interrupt routines were by default high priority.

The 16- and 32-bit compilers have used the `interrupt` attribute to define interrupt functions.

2.5.10.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the `interrupt` qualifier, as in the following examples:

```
void interrupt myIsr(void)
void interrupt low_priority myLoIsr(void)
```

to the following, respectively

```
void __interrupt(high_priority) myIsr(void)
void __interrupt(low_priority) myLoIsr(void)
```

For 16-bit compilers, change any occurrence of the `interrupt` attribute, as in the following example:

```
void __attribute__((interrupt,auto_psv,(irq(52)))) myIsr(void);
```

to

```
void __interrupt(auto_psv,(irq(52))) myIsr(void);
```

For 32-bit compilers, the `__interrupt()` keyword takes two parameters, the vector number and the (optional) IPL value. Change code which uses the `interrupt` attribute, similar to these examples:

```
void __attribute__((vector(0), interrupt(IPL7AUTO), nomips16))
myisr0_7A(void) {}
```

```
void __attribute__((vector(1), interrupt(IPL6SRS), nomips16))
myisr1_6SRS(void) {}
```

```
/* Determine IPL and context-saving mode at runtime */
void __attribute__((vector(2), interrupt(), nomips16))
myisr2_RUNTIME(void) {}
```

to

```
void __interrupt(0,IPL7AUTO) myisr0_7A(void) {}
```

```
void __interrupt(1,IPL6SRS) myisr1_6SRS(void) {}
```

```
/* Determine IPL and context-saving mode at runtime */
void __interrupt(2) myisr2_RUNTIME(void) {}
```

2.5.10.4 CAVEATS

None.

2.5.11 Packing Objects

The `__pack` specifier may be used to indicate that structures should not use memory gaps to align structure members, or that individual structure members should not be aligned.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

Some compilers may not pad structures with alignment gaps for some devices and use of this specifier for such devices will be ignored.

2.5.11.1 EXAMPLE

The following shows a structure qualified using `__pack` as well as a structure where one member has been explicitly packed.

```
__pack struct DATAPOINT {
    unsigned char type;
    int value;
} x-point;
struct LINETYPE {
    unsigned char type;
    __pack int start;
    long total;
} line;
```

2.5.11.2 DIFFERENCES

The `__pack` specifier is a new CCI specifier available with XC8. This specifier has no apparent effect since the device memory is byte addressable for all data objects.

The 16- and 32-bit compilers have used the `packed` attribute to indicate that a structure member was not aligned with a memory gap.

2.5.11.3 MIGRATION TO THE CCI

No migration is required for XC8.

For 16- and 32-bit compilers, change any occurrence of the `packed` attribute, as in the following example:

```
struct DOT
{
    char a;
    int x[2] __attribute__ ((packed));
};
```

to:

```
struct DOT
{
    char a;
    __pack int x[2];
};
```

Alternatively, you may pack the entire structure, if required.

2.5.11.4 CAVEATS

None.

2.5.12 Indicating Antiquated Objects

The `__deprecated` specifier may be used to indicate that an object has limited longevity and should not be used in new designs. It is commonly used by the compiler vendor to indicate that compiler extensions or features may become obsolete, or that better features have been developed and which should be used in preference.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

2.5.12.1 EXAMPLE

The following shows a function which uses the `__deprecated` keyword.

```
void __deprecated getValue(int mode)
{
    //...
}
```

2.5.12.2 DIFFERENCES

No `deprecated` feature was implemented on 8-bit compilers.

The 16- and 32-bit compilers have used the `deprecated` attribute (note different spelling) to indicate that objects should be avoided if possible.

2.5.12.3 MIGRATION TO THE CCI

For 16- and 32-bit compilers, change any occurrence of the `deprecated` attribute, as in the following example:

```
int __attribute__((deprecated)) intMask;
to:
int __deprecated intMask;
```

2.5.12.4 CAVEATS

None.

2.5.13 Assigning Objects to Sections

The `__section()` specifier may be used to indicate that an object should be located in the named section (or `psect`, using the XC8 terminology). This is typically used when the object has special and unique linking requirements which cannot be addressed by existing compiler features.

Use the native keywords discussed in the Differences section to look up information on the semantics of this specifier.

2.5.13.1 EXAMPLE

The following shows a variable which uses the `__section` keyword.

```
int __section("comSec") commonFlag;
```

2.5.13.2 DIFFERENCES

The 8-bit compilers have used the `#pragma psect` directive to redirect objects to a new section, or psect. The operation of the `__section()` specifier is different to this pragma in several ways, described below.

Unlike with the pragma, the new psect created with `__section()` does not inherit the flags of the psect in which the object would normally have been allocated. This means that the new psect can be linked in any memory area, including any data bank. The compiler will also make no assumptions about the location of the object in the new section. Objects redirected to new psects using the pragma must always be linked in the same memory area, albeit at any address in that area.

The `__section()` specifier allows objects that are initialized to be placed in a different psect. Initialization of the object will still be performed even in the new psect. This will require the automatic allocation of an additional psect (whose name will be the same as the new psect prefixed with the letter `i`), which will contain the initial values. The pragma cannot be used with objects that are initialized.

Objects allocated a different psect with `__section()` will be cleared by the runtime startup code, unlike objects which use the pragma.

You must reserve memory, and locate via a linker option, for any new psect created with a `__section()` specifier in the current XC8 compiler implementation.

The 16- and 32-bit compilers have used the `section` attribute to indicate a different destination section name. The `__section()` specifier works in a similar way to the attribute.

2.5.13.3 MIGRATION TO THE CCI

For XC8, change any occurrence of the `#pragma psect` directive, such as

```
#pragma psect text%%u=myText
int getMode(int target) {
//...
}
```

to the `__section()` specifier, as in

```
int __section ("myText") getMode(int target) {
//...
}
```

For 16- and 32-bit compilers, change any occurrence of the `section` attribute, as in the following example:

```
int __attribute__((section("myVars"))) intMask;
```

to:

```
int __section("myVars") intMask;
```

2.5.13.4 CAVEATS

With XC8, the `__section()` specifier cannot be used with any interrupt function.

2.5.14 Specifying Configuration Bits

The `#pragma config` directive may be used to program the configuration bits for a device. The pragma has the form:

```
#pragma config setting = state|value  
#pragma config register = value
```

where *setting* is a configuration setting descriptor (e.g., `WDT`), *state* is a descriptive value (e.g., `ON`) and *value* is a numerical value. The register token may represent a whole configuration word register, e.g., `CONFIG1L`.

Use the native keywords discussed in the Differences section to look up information on the semantics of this directive.

2.5.14.1 EXAMPLE

The following shows configuration bits being specified using this pragma.

```
#pragma config WDT=ON, WDTPS = 0x1A
```

2.5.14.2 DIFFERENCES

The 8-bit compilers have used the `__CONFIG()` macro for some targets that did not already have support for the `#pragma config`.

The 16-bit compilers have used a number of macros to specify the configuration settings.

The 32-bit compilers supported the use of `#pragma config`.

2.5.14.3 MIGRATION TO THE CCI

For the 8-bit compilers, change any occurrence of the `__CONFIG()` macro, such as

```
__CONFIG(WDTEN & XT & DPROT)
```

to the `#pragma config` directive, as in

```
#pragma config WDTE=ON, FOSC=XT, CPD=ON
```

No migration is required if the `#pragma config` was already used.

For the 16-bit compilers, change any occurrence of the `_FOSC()` or `_FBORPOR()` macros attribute, as in the following example:

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

to:

```
#pragma config FCKSMEM = CSW_ON_FSCM_ON, FPR = ECIO_PLL16
```

No migration is required for 32-bit code.

2.5.14.4 CAVEATS

None.

2.5.15 Manifest Macros

The CCI defines the general form for macros that manifest the compiler and target device characteristics. These macros can be used to conditionally compile alternate source code based on the compiler or the target device.

The macros and macro families are details in Table 2-1.

TABLE 2-1: MANIFEST MACROS DEFINED BY THE CCI

Name	Meaning if defined	Example
<code>__XC__</code>	Compiled with an MPLAB XC compiler	<code>__XC__</code>
<code>__CCI__</code>	Compiler is CCI compliant and CCI enforcement is enabled	<code>__CCI__</code>
<code>__XC##__</code>	The specific XC compiler used (## can be 8, 16 or 32)	<code>__XC8__</code>
<code>__DEVICEFAMILY__</code>	The family of the selected target device	<code>__dsPIC30F__</code>
<code>__DEVICENAME__</code>	The selected target device name	<code>__18F452__</code>

2.5.15.1 EXAMPLE

The following shows code which is conditionally compiled dependent on the device having EEPROM memory.

```
#ifdef __XC16__
void __interrupt(__auto_psv__) myIsr(void)
#else
void __interrupt(low_priority) myIsr(void)
#endif
```

2.5.15.2 DIFFERENCES

Some of these CCI macros are new (for example `__CCI__`), and others have different names to previous symbols with identical meaning (for example `__18F452` is now `__18F452__`).

2.5.15.3 MIGRATION TO THE CCI

Any code which uses compiler-defined macros will need review. Old macros will continue to work as expected, but they are not compliant with the CCI.

2.5.15.4 CAVEATS

None.

2.5.16 In-line Assembly

The `asm()` statement may be used to insert assembly code in-line with C code. The argument is a C string literal which represents a single assembly instruction. Obviously, the instructions contained in the argument are device specific.

Use the native keywords discussed in the Differences section to look up information on the semantics of this statement.

2.5.16.1 EXAMPLE

The following shows a `MOVLW` instruction being inserted in-line.

```
asm("MOVLW _foobar");
```

2.5.16.2 DIFFERENCES

The 8-bit compilers have used either the `asm()` or `#asm ... #endasm` constructs to insert in-line assembly code.

This is the same syntax used by the 16- and 32-bit compilers.

2.5.16.3 MIGRATION TO THE CCI

For 8-bit compilers change any instance of `#asm ... #endasm` so that each instruction in this `#asm` block is placed in its own `asm()` statement, for example:

```
#asm
    MOVLW 20
    MOVWF _i
    CLRF  Ii+1
#endasm
```

to

```
asm("MOVLW20");
asm("MOVWF _i");
asm("CLRFIi+1");
```

No migration is required for the 16- or 32-bit compilers.

2.5.16.4 CAVEATS

None.

2.6 COMPILER FEATURES

The following items detail compiler options and features that are not directly associated with source code that

2.6.1 Enabling the CCI

It is assumed you are using an IDE to build projects that use the CCI. The widget in the MPLAB X IDE Project Properties to enable CCI conformance is [Use CCI Syntax](#) in the Compiler category. A widget with the same name is available in MPLAB IDE v8 under the Compiler tab.

If you are not using this IDE, then the command-line options are `--EXT=cci` for XC8 or `-mcci` for XC16/32.

2.6.1.1 DIFFERENCES

This option has never been implemented previously.

2.6.1.2 MIGRATION TO THE CCI

Enable the option.

2.6.1.3 CAVEATS

None.

Chapter 3. How To's

3.1 INTRODUCTION

This section contains help and references for situations that are frequently encountered when building projects for Microchip 16-bit devices. Click the links at the beginning of each section to assist in finding the topic relevant to your question. Some topics are indexed in multiple sections.

Start Here

- Installing and Activating the Compiler
- Invoking the Compiler
- Writing Source Code
- Getting My Application to Do What I Want
- Understanding the Compilation Process
- Fixing Code That Does Not Work

3.2 INSTALLING AND ACTIVATING THE COMPILER

This section details questions that might arise when installing or activating the compiler.

- How Do I Install and Activate My Compiler?
- How Can I Tell If the Compiler Has Installed and Activated Successfully?
- Can I Install More Than One Version of the Same Compiler?

3.2.1 How Do I Install and Activate My Compiler?

Installation and activation of the license are performed simultaneously by the XC compiler installer. For instructions, refer to the following document. It is available for download from the Microchip Technology website, www.microchip.com.

"Installing and Licensing MPLAB XC C Compilers" (DS52059)

3.2.2 How Can I Tell If the Compiler Has Installed and Activated Successfully?

To verify installation and activation of the compiler, you must compile code. You can use the example code that comes with the compiler. It is located, by default, in the following folder: `c:\Program Files\Microchip\xc16\examples`

Then, depending on your operating system, find the file `run_hello.bat` (Windows OS) or `run_hello.sh` (Mac/Linux OS) in the `xc16_getting_started` folder. Edit the optimization level for your expected license level. For more information, see the following chapter and section in this user's guide:

Chapter 18. "Optimizations"

Section 5.7.6 "Options for Controlling Optimization"

Run the edited batch or shell file. If the tools are installed correctly, the output should show the various steps in the compilation and execution process, ending with the text: `Hello, world!`

3.2.3 Can I Install More Than One Version of the Same Compiler?

Yes, the compilers and installation process have been designed to allow you to have more than one version of the same compiler installed. In MPLAB X IDE, you can easily switch between compiler versions by changing options in the IDE. For details, see the following section in this user's guide:

Section 3.3.3 “How Can I Select Which Compiler Version to Build With?”

Compilers should be installed into a directory that is named according to the compiler version. This is reflected in the default directory specified by the installer. For example, the MPLAB XC16 compilers v1.00 and v1.10 would typically be placed in separate directories, as shown below:

```
C:\Program Files\Microchip\xc16\v1.00\  
C:\Program Files\Microchip\xc16\v1.10\
```

3.3 INVOKING THE COMPILER

This section discusses how the compiler is run from the command line, and from the IDE. Information about how to use compiler options and the build process to achieve maximum results from the compiler are also included.

- How Do I Compile from Within MPLAB X IDE?
- How Do I Compile on the Command Line?
- How Can I Select Which Compiler Version to Build With?
- How Can I Change the Compiler Optimizations?
- How Do I Know Which Optimization Features I Get?
- How Do I Know Which Compiler Options Are Available and What They Do?
- How Do I Build Libraries?
- How Do I Know What the Build Options in MPLAB X IDE Do?
- What is Different About an MPLAB X IDE Debug Build?
- See *also*, Why No Disassembly in the MPLAB X IDE Disassembly Window?
- See *also*, “MPLAB XC16 Assembler, Linker and Utilities User’s Guide” (DS52106)
- See *also*, How Do I Stop the Compiler from Using Certain Memory Locations?

3.3.1 How Do I Compile from Within MPLAB X IDE?

In MPLAB X IDE, you compile your code by building a project.

For more on using the compiler with MPLAB X IDE, see the following chapter and section in this user’s guide:

Chapter 4. “XC16 Toolchain and MPLAB X IDE”

Section 3.3.3 “How Can I Select Which Compiler Version to Build With?”

3.3.2 How Do I Compile on the Command Line?

To compile code on the command line, refer to the following chapter and section in this user’s guide:

Chapter 5. “Compiler Command-Line Driver”

Section 3.3.3 “How Can I Select Which Compiler Version to Build With?”

3.3.3 How Can I Select Which Compiler Version to Build With?

The compilation and installation process was designed to allow you to have more than one compiler version installed at the same time

In MPLAB X IDE, select the compiler to use for building a project by opening the Project Properties window (*File>Project Properties*) and selecting the Configuration category (Conf: [default]). A list of MPLAB XC16 compiler versions is shown in the Compiler Toolchain, on the far right of the window. Select the MPLAB XC16 compiler you require.

Once selected, the controls for that compiler are shown by selecting the XC16 global options, XC16 Compiler, and XC16 Linker categories. These reveal a pane of options on the right; each category has several panes which can be selected from a pull-down menu that is near the top of the pane.

3.3.4 How Can I Change the Compiler Optimizations?

You can only select optimizations that your license entitles you to use. For more on compiler licenses and related optimizations, and on setting optimizations, see the following chapter and section in this user's guide:

Chapter 18. "Optimizations"

Section 5.7.6 "Options for Controlling Optimization"

3.3.5 How Do I Know Which Optimization Features I Get?

When you select an optimization level, you get several optimization features. These features are tabulated in the following section of this user's guide:

Section 18.3 "Optimization Feature Summary"

3.3.6 How Do I Know Which Compiler Options Are Available and What They Do?

A list of all compiler options can be found in the following section of this user's guide:

Section 5.7 "Driver Option Descriptions"

3.3.7 How Do I Build Libraries?

For information on how to create and build your own libraries, see the following section of this user's guide:

Section 5.4.1.2 "User-Defined libraries"

3.3.8 How Do I Know What the Build Options in MPLAB X IDE Do?

Most of the widgets and controls in the MPLAB X IDE Project Properties window, XC16 options, map directly to a corresponding command-line driver option or suboption. For a list of options and any corresponding command-line options, refer to the following section of this user's guide:

Section 4.5 "Project Setup"

3.3.9 What is Different About an MPLAB X IDE Debug Build?

MPLAB X IDE needs to use extra memory for a debug build, as debugging requires additional resources. See MPLAB X IDE documentation for details.

3.4 WRITING SOURCE CODE

This section presents issues that pertain to the source code you write. It has been subdivided into sections listed below.

- C Language Specifics
- Device-Specific Features
- Memory Allocation
- Variables
- Functions
- Interrupts
- Assembly Code

3.4.1 C Language Specifics

The MPLAB XC16 C compiler is an ANSI C compliant compiler and therefore follows standard C language conventions. For more information, see the following section in this user's guide:

Section 1.3.1 "ANSI C Standard"

3.4.2 Device-Specific Features

This section discusses the code that needs to be written to set up or control a feature that is specific to Microchip devices.

- How Do I Port My Code to Different Device Architectures?
- How Do I Set the Configuration Bits?
- How Do I Access the User ID Locations?
- How Do I Access Special Function Registers (SFRs)?
- Which Device-Specific Symbols Does the Compiler Define, and Can I Use Them?
- See *also*, How Do I Stop the Compiler from Using Certain Memory Locations?

3.4.2.1 HOW DO I PORT MY CODE TO DIFFERENT DEVICE ARCHITECTURES?

To reduce the work required to port code between architectures, a Common C Interface (CCI) has been developed. If you use these coding styles, your code will more easily migrate upward. For more on CCI, see the following chapter in this user's guide:

Chapter 2. "Common C Interface"

3.4.2.2 HOW DO I SET THE CONFIGURATION BITS?

These should be set in your code by using either a macro or pragma. Earlier versions of MPLAB IDE allowed you to set these bits in a dialog, but MPLAB X IDE requires that they be specified in your source code. See the following section in this user's guide:

Section 6.5 "Configuration Bit Access"

3.4.2.3 HOW DO I ACCESS THE USER ID LOCATIONS?

Currently, the only way access a device (or family) ID location is to specify the fixed address of the device-ID register(s). There is not a supplied macro or pragma at this time. Consult your device data sheet for the address of the device-ID register(s).

3.4.2.4 HOW DO I ACCESS SPECIAL FUNCTION REGISTERS (SFRS)?

The compiler is distributed with header files that define variables. The variables are mapped over the top of memory-mapped SFRs. Since these are C variables, they can be used like any other C variables. No new syntax is required to access these registers.

The names of these variables should be the same as those indicated in the data sheet for the device you are using.

Bits within SFRs can also be accessed. Bit-fields are available in structures which map over the SFR as a whole. For example, PORTCbits.RC1 means the RC1 bit of PORTC. For more on header files, see the following section in this user's guide:

Section 6.3 “Device Header Files”

3.4.2.5 WHICH DEVICE-SPECIFIC SYMBOLS DOES THE COMPILER DEFINE, AND CAN I USE THEM?

The compiler defines some device-specific, and other, symbols or macros. They are discussed in the following section in this user's guide:

Section 19.4 “Predefined Macro Names”

3.4.3 Memory Allocation

These questions relate to the way in which your source code affects memory allocation.

- How Do I Position Variables or Functions at an Address I Nominate?
- How Do I Place Variables in Program Memory?
- How Do I Allocate Space for a Variable But Not Initialize/Load Any Value?
- How Do I Stop the Compiler from Using Certain Memory Locations?

3.4.3.1 HOW DO I POSITION VARIABLES OR FUNCTIONS AT AN ADDRESS I NOMINATE?

Nudging the toolchain to allocate variables or functions in specific areas of memory can make it harder for the linker to do its job. Tools are provided to solve problems that may exist, but they should always be used carefully. For example, instead of fixing an object at a specific address (using the `address` attribute or the `__at` construct), it may be sufficient to group variables together using the `section` attribute.

3.4.3.2 HOW DO I PLACE VARIABLES IN PROGRAM MEMORY?

For information on how to place variables in program memory space, refer to the following section in this user's guide:

Section 10.4 "Variables in Program Space"

3.4.3.3 HOW DO I ALLOCATE SPACE FOR A VARIABLE BUT NOT INITIALIZE/LOAD ANY VALUE?

To allocate memory space for a variable without initializing or loading the variable in memory, you can use the `noload` attribute. For more on variable attributes, see the following section in this user's guide:

Section 8.12 "Variable Attributes"

3.4.3.4 HOW DO I STOP THE COMPILER FROM USING CERTAIN MEMORY LOCATIONS?

Concatenating an `address` attribute with the `noload` attribute can be used to block out sections of memory. Also, you can use the option `-mreserved`. For more on variable attributes and options, see the following sections in this user's guide:

Section 8.12 "Variable Attributes"

Section 5.7.1 "Options Specific to 16-Bit Devices"

3.4.4 Variables

This examines questions that relate to the definition and usage of variables and types within a program.

- Why Are My Floating-point Results Not Quite What I Am Expecting?
- How Do I Retain the Value of a Variable Even After a Soft Reset?
- How Do I Save C Variables When an ISR Is Invoked?
- How Long Can I Make My Variable and Macro Names?
- How Do I Access Values Stored in a PSV or EDS Page?
- Why Would I Need to Place Data Into Its Own Section?
- How Can I Load a Value Into Flash Memory?
- How Can I Pack Data Into Flash Memory?
- How Can I Define a Large Array?
- See *also*, How Do I Share Data Between Interrupt and Main-line Code?
- See *also*, How Do I Position Variables or Functions at an Address I Nominate?
- See *also*, How Do I Place Variables in Program Memory?
- See *also*, How Do I Place Variables in Off-Chip Memory?
- See *also*, How Can I Rotate a Variable?
- See *also*, How Do I Learn Where Variables and Functions Have Been Positioned?

3.4.4.1 WHY ARE MY FLOATING-POINT RESULTS NOT QUITE WHAT I AM EXPECTING?

First, ensure that you are using the floating-point data types that you intend. We recommend using the types `long double`, explicitly, in your program, when IEEE double precision (64 bit) format floating-point values are desired, and `float` when IEEE single precision (32 bit) format values are desired. By default the compiler uses IEEE single precision format for the type `double`. Use the `-fno-short-double` switch to specify IEEE double precision (64 bit) format for the type `double`.

Next, be aware of the limitations of the floating-point formats, and the effects of rounding. Not all real numbers can be represented exactly in the floating-point formats. For example, the fraction 1/10 cannot be represented exactly in either the single or double precision formats.

If the result of a load or a computation is 1/10, the value stored in the floating-point format will be the closest approximation representable in that format. In such cases, it is said that the “true” value has been “rounded” to the nearest approximation, according to the rules of the IEEE arithmetic. This small discrepancy in a value that is introduced early in a computation can accumulate over many operations and produce noticeable error. In general, computations may start from numbers that are exactly representable (like 1, and 10), and yield results that are not (like 1/10). This is not due to the compiler's choice of code generated, nor any specifics of the microprocessor architecture, it's an essential characteristic the IEEE floating-point formats and rules of arithmetic. Any compiler/microprocessor platform faces the same issues. For more information, see the following section in this user's guide:

Section 8.4 “Floating-Point Data Types”

3.4.4.2 HOW DO I RETAIN THE VALUE OF A VARIABLE EVEN AFTER A SOFT RESET?

First, consult your device data sheet to see which Resets are available. Then, to save the values of your variables after a software Reset, you can use the `persistent` attribute, which specifies that the variable should not be initialized or cleared at startup. For more on this attribute, see the following section in this user's guide:

Section 8.12 “Variable Attributes”

3.4.4.3 HOW DO I SAVE C VARIABLES WHEN AN ISR IS INVOKED?

You can use the `save` parameter of the `interrupt` attribute to save variables and SFRs so that their values may be restored on a return from interrupt. For more information, see the following section in this user's guide:

Section 14.5 “Interrupt Service Routine Context Saving”

3.4.4.4 HOW LONG CAN I MAKE MY VARIABLE AND MACRO NAMES?

For MPLAB XC16, no limit is imposed, but for CCI there is a limit. For details, see the following section in this user's guide:

Section 2.4.5 “The number of Significant Initial Characters in an Identifier”

3.4.4.5 HOW DO I ACCESS VALUES STORED IN A PSV OR EDS PAGE?

16-bit devices have a method of accessing data memory from within Flash memory called Program Space Visibility (PSV). You can access values in PSV memory by using the `__psv__` qualifier. Another method to access data space from program memory is called Extended Data Space (EDS). You can access values in EDS by using the `__eds__` qualifier. For more on each of these qualifiers, see the following sections in this user's guide:

Section 8.11.1 “__psv__ Type Qualifier”

Section 8.11.3 “__eds__ Type Qualifier”

3.4.4.6 WHY WOULD I NEED TO PLACE DATA INTO ITS OWN SECTION?

The MPLAB XC16 Object Linker will place data into sections efficiently. However, you may want to manually place groups of variables into sections because it is easier than manually placing each individual variable at a specific address. If necessary, absolute starting addresses may be specified in user-defined sections within the device linker script. To place data into its own section, you can use the `section` attribute, discussed in the following section in this user's guide:

Section 8.12 “Variable Attributes”

To edit user-defined sections within the linker script, see the following document. It is available for download from the Microchip Technology website, www.microchip.com.

“MPLAB XC16 Assembler, Linker, and Utilities User's Guide” (DS52106),
Section 9.5 “Contents of a Linker Script”

3.4.4.7 HOW CAN I LOAD A VALUE INTO FLASH MEMORY?

The compiler provides different ways of defining Flash variables.

- A variable can be **explicitly** placed into Flash using an appropriate `space` attribute.
- Variables are **implicitly** placed into Flash in the default const-in-code memory model if they have the C `const` type qualifier.

These differences allow you to choose how much work you want to do to access variables, and how much you want the compiler to do. The compiler has the least to do when you simply specify the attribute `space(prog)`, and the initial value; which leaves the access (usually via `tblrd` instructions) up to you, the programmer. The compiler has the most to do when you combine the `space(prog)` attribute with an appropriate access qualifier, such as `__eds__` or `__prog__`.

Also, there is often a single page of Flash space dedicated to `const` qualified objects. See `-mconst-in-code` in the following sections of this user's guide for more details:

Section 8.12 “Variable Attributes”

Section 10.4 “Variables in Program Space”

3.4.4.8 HOW CAN I PACK DATA INTO FLASH MEMORY?

To specify the upper byte of variables stored into `space(prog)` sections, you can use either the `-mfillupper` option or the `fillupper` variable attribute. See the following sections of this user's guide for more information:

Section 5.7.1 “Options Specific to 16-Bit Devices” (`-mfillupper`)

Section 8.12 “Variable Attributes” (`fillupper`)

3.4.4.9 HOW CAN I DEFINE A LARGE ARRAY?

By default, arrays are allocated 32K of memory. If you need more, you can use the compiler option `-mlarge-arrays`, remembering that there will be a memory cost. For more on the option, see the following section of this user's guide:

Section 10.3.1.3 “Non-Auto Variable Size Limits”

3.4.5 Functions

This section examines questions that relate to functions.

- How Do I Stop A Function From Being Removed?
- Why Should I Inline My Function?
- Why is My Function Not Inline?
- Why Should I Place a Function Into its Own Section?
- How Do I Prevent the Compiler From Saving or Restoring Any Registers?
- How Can I Tell if a Function is Being Used
- How Can I Find Out Which Functions are Contained Inside of the Compiler?
- Where are Arguments That Are Passed to Functions Located in Memory?
- See *also*, How Can I Tell How Big a Function Is?
- See *also*, How Do I Learn Where Variables and Functions Have Been Positioned?
- See *also*, How Do I Use Interrupts in C?

3.4.5.1 HOW DO I STOP A FUNCTION FROM BEING REMOVED?

Apply the attribute `keep` to a function to prevent the linker from removing it with `--gc-sections`, even when the function is unused. See the following section in this user's guide:

Section 13.2.2 “Function Attributes”

3.4.5.2 WHY SHOULD I INLINE MY FUNCTION?

The reason why you might want to inline your function, and how you would do so are discussed in the following section in this user's guide:

Section 13.6 “Inline Functions”

3.4.5.3 WHY IS MY FUNCTION NOT INLINE?

Unless you use the `inline` keyword to specifically inline a function, the compiler will make the decision about which functions to inline. In general, small functions are inlined whereas larger ones are not. For details, see the following section in this user's guide:

Section 13.6 “Inline Functions”

3.4.5.4 WHY SHOULD I PLACE A FUNCTION INTO ITS OWN SECTION?

The MPLAB XC16 Object Linker will place functions into sections efficiently. Since manual placement of functions into program memory may reduce the linker's ability to do this with maximum efficiency, most applications should not include manual placement of functions into program memory. However, bootloader applications are special. They require application firmware to reside higher in memory than the bootloader, which requires manual placement of functions to avoid conflicts with the bootloader application.

Also, applications that require code placement in secure sections need custom placement of program functions using the `boot` or `secure` attributes. Address attributes can be applied to functions, as well.

To place a function into its own section with the `section` attribute, to place a function with the `boot` or `secure` attributes, or to place a function with an `address` attribute, see the following section in this user's guide:

Section 13.2.2 “Function Attributes”

3.4.5.5 HOW DO I PREVENT THE COMPILER FROM SAVING OR RESTORING ANY REGISTERS?

If you do not want register values saved or restored after an interrupt or Reset, you can use the `naked` attribute. This attribute should be used with care, though, because you generally do want to save these values. For details, see the following section in this user's guide:

Section 13.2.2 “Function Attributes”

3.4.5.6 HOW CAN I TELL IF A FUNCTION IS BEING USED

After the project has built, view the map file for a listing of used functions. Use the linker option `--gc-sections` to ensure unused functions are removed. For details, see the following section in this user's guide:

Section 13.2.2 “Function Attributes”

For more on the linker, see the following document. It is available for download from the Microchip Technology website, www.microchip.com.

“MPLAB XC16 Assembler, Linker, and Utilities User's Guide” (DS52106).

3.4.5.7 HOW CAN I FIND OUT WHICH FUNCTIONS ARE CONTAINED INSIDE OF THE COMPILER?

You can see compiler predefined symbols/macros and functions by running, and stopping the preprocessor, and then examining the output. Options to do this are discussed in the following section in this user's guide:

Section 5.7.2 “Options for Controlling the Kind of Output”

3.4.5.8 WHERE ARE ARGUMENTS THAT ARE PASSED TO FUNCTIONS LOCATED IN MEMORY?

You will need to run compiler code to determine this. See the application binary interface when running the compiler. Some built-ins may be helpful, also. See the following location in this user's guide:

Appendix G. “Built-in Functions”

3.4.6 Interrupts

Interrupt and interrupt service routine (ISR) questions are discussed in this section.

- How Do I Use Interrupts in C?
- How Do I Add a Trap Interrupt Vector to a Project?
- Can/Should My Application be able to Return from a Trap?
- How Do I Share Data between Two Interrupt Routines?
- What is the Default Interrupt, Where is it Defined, and How Do I Use It?
- See *also*, How Can I Make My Interrupt Routine Faster?
- See *also*, How Do I Share Data Between Interrupt and Main-line Code?
- See *also*, How Do I Save C Variables When an ISR Is Invoked?

3.4.6.1 HOW DO I USE INTERRUPTS IN C?

First, be aware of the interrupt hardware that is available on your target device. 16-bit devices implement several separate interrupt vector locations and use a priority scheme. See your device data sheet for details. Then, review the following chapter in this user's guide for more information:

Chapter 14. "Interrupts"

3.4.6.2 HOW DO I ADD A TRAP INTERRUPT VECTOR TO A PROJECT?

The compiler treats hard traps the same as normal interrupt vectors, they have names just like the handlers for peripherals. A useful place to find all the interrupt functions supported by a particular device is in the `docs` folder of your install:

`vector_index.html`.

The general format is:

```
void __attribute__((interrupt)) ISR_fn_name(void) {  
}
```

3.4.6.3 CAN/SHOULD MY APPLICATION BE ABLE TO RETURN FROM A TRAP?

This question is very specific to the application/trap. The general answer is that the application should probably safely restart when such an event occurs.

3.4.6.4 HOW DO I SHARE DATA BETWEEN TWO INTERRUPT ROUTINES?

By their very nature, ISRs do not send results or receive parameters. The only way to share data is by using common data sharing procedures. Examples of these would be by volatile global variables or via specialized accessor functions, which can carefully control access to data, and make your application more robust.

Whenever data is shared across different threads of control, which is really what an interrupt routine is, it is important that the shared data accesses are protected from further interruption as not all accesses are atomic.

3.4.6.5 WHAT IS THE DEFAULT INTERRUPT, WHERE IS IT DEFINED, AND HOW DO I USE IT?

The "default interrupt" fills in the vector table when no other named vector exists. If it is not defined, the compiler will create one that will halt in a debugging environment or Reset in a normal execution (non-debug) environment.

You can define your own handler by simply defining an ISR named:

`_DefaultInterrupt`.

3.4.7 Assembly Code

This section examines questions that arise when writing assembly code as part of a C project.

- How Should I Combine Assembly and C Code?
- What Do I Need Other Than Instructions In an Assembly Source File?
- How Do I Access C Objects from Assembly Code?
- How Can I Access SFRs From Within Assembly Code?
- When Should I Combine Assembly and C Code?
- What Is the Difference Between .s and .S Files?
- How Do I Make a Function Wrapper For an Assembly Module?
- When Should Inline Assembly Be Used Instead of Assembly Modules?

3.4.7.1 HOW SHOULD I COMBINE ASSEMBLY AND C CODE?

Assembly code can be written as separate functions that are called from C code or as inline from within the C code. For details, see the following chapter in this user's guide:

Chapter 16. "Mixing C and Assembly Code"

3.4.7.2 WHAT DO I NEED OTHER THAN INSTRUCTIONS IN AN ASSEMBLY SOURCE FILE?

Assembly code typically needs assembler directives, as well as the instructions themselves. The operation of these directives is described in the following document. It is available for download from the Microchip Technology website, www.microchip.com.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106).

There are two directives that are commonly used in assembly code. The first one is the `.section` directive. All assembly code must be placed in a section using this directive, so that it can be manipulated as a whole by the linker, and placed into memory. The second one is the `.global` directive. This directive is used to make symbols accessible across multiple source files.

3.4.7.3 HOW DO I ACCESS C OBJECTS FROM ASSEMBLY CODE?

Most C objects are accessible from assembly code. There is a mapping between the symbols used in the C source and those used in the assembly code generated from this source. Your assembly should access the assembly-equivalent symbols which are detailed in the following section in this user's guide:

Section 16.2 "Mixing Assembly Language and C Variables and Functions".

3.4.7.4 HOW CAN I ACCESS SFRs FROM WITHIN ASSEMBLY CODE?

The easiest way to gain access to SFRs in assembly code is to use the device-generic include file (`xc.h`) that equates symbols to the corresponding SFR address.

There is no guarantee that you will be able to access symbols generated by the compilation of C code, even if it is code that accesses the SFR that you require. See the following section in this user's guide:

Section 13.8 "Function Call Conventions"

3.4.7.5 WHEN SHOULD I COMBINE ASSEMBLY AND C CODE?

This is a very application-dependent question. There are some device-specific operations that cannot be done in normal C code, typically the language tool will provide a built-in function to provide this feature.

If you do decide to combine assembly and C code, ensure that the code complies with the run-time model, i.e., that arguments are transmitted in the correct registers; and that registers are properly used, and not overwritten.

3.4.7.6 WHAT IS THE DIFFERENCE BETWEEN .s AND .S FILES?

Both of these files should contain assembly language. .s files are preprocessed by the C compiler. This means that they might include C preprocessing statements (`#define`, `#ifdef`, and etc.), but they should not contain C statements. For information on these assembly files, see the following section in this user's guide:

Section 5.5.1 "Output Files"

3.4.7.7 HOW DO I MAKE A FUNCTION WRAPPER FOR AN ASSEMBLY MODULE?

The C compiler expects all C visible names to start with a leading underscore. In order to export a function to C code from assembly code, it must be globally visible. Of course, there should also be an external prototype in C so that the compiler can properly see it.

```
foo.s:

    .text
    .global _foo
    _foo: retlw #0,w0

main.c:

    extern int foo(void);
```

For details on using C code with an assembly modules, see the following section in this user's guide:

Section 16.2 "Mixing Assembly Language and C Variables and Functions"

3.4.7.8 WHEN SHOULD INLINE ASSEMBLY BE USED INSTEAD OF ASSEMBLY MODULES?

If the programmer does decide to combine assembly code and C code; ensure that the code complies with the run-time model; that registers are properly used, and not overwritten; and that the code uses the GNU extended inline assembly code. Long sequences are often hard to debug, so ensure that you correctly follow the guidelines.

3.5 GETTING MY APPLICATION TO DO WHAT I WANT

This section provides programming techniques, applications and examples. It also examines questions that relate to making an application perform a specific task.

- How Do I Generate Debug Information?
- Why No Disassembly in the MPLAB X IDE Disassembly Window?
- How Do I Share Data Between Interrupt and Main-line Code?
- How to Protect My Code After It Is Programmed Into a Device?
- How Do I Redirect Standard I/O When Using Printf?
- How Do I Place Variables in Off-Chip Memory?
- How Can I Implement a Delay in My Code?
- How Can I Rotate a Variable?

3.5.1 How Do I Generate Debug Information?

For the compiler and assembler, the command-line option `-g` is used to generate debugging information. For details, refer to the following document (available on the Microchip website), and also, see the following section in this user's guide:

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106)

Section 5.7.5 "Options for Debugging"

3.5.2 Why No Disassembly in the MPLAB X IDE Disassembly Window?

You must enable the generation of debug information before you can see anything in the disassembly window. See the following section in this user's guide:

Section 3.5.1 "How Do I Generate Debug Information?"

3.5.3 How Do I Share Data Between Interrupt and Main-line Code?

Variables accessed from both interrupt and main-line code can easily become corrupted or misread by the program. The `volatile` qualifier tells the compiler to avoid performing optimizations on such variables. This will fix some of the issues associated with this problem.

Other issues arise because the way variables are accessed can vary from statement to statement. Therefore it is usually best to avoid these issues entirely by disabling interrupts prior to the variable being accessed in main-line code, then to re-enable the interrupts afterwards. For more information on these solutions, see the following sections in this user's guide:

Section 8.10.2 "Volatile Type Qualifier"

Section 14.7 "Enabling/Disabling Interrupts"

3.5.4 How to Protect My Code After It Is Programmed Into a Device?

Many devices with flash program memory allow all or part of this memory to be write protected. The device Configuration bits need to be set correctly for this to take place. For more on using Configuration bits, see the following sections in this user's guide:

Section 6.5 "Configuration Bit Access"

Section 2.5.14 "Specifying Configuration Bits" (CCI)

Your device data sheet is also a good resource for this question.

3.5.5 How Do I Redirect Standard I/O When Using Printf?

The `printf` function does two things: it formats text, based on the format string and placeholders you specify; and it sends (prints) this formatted text to a destination (or stream). You can choose the `printf` output go to an LCD, SPI module or USART, for example. For more on the using ANSI C function `printf`, including how to customize the output so that it goes to another peripheral, refer to the following document. It is available for download from the Microchip Technology website, www.microchip.com.

“16-Bit Language Tools Libraries Reference Manual” (DS50001456)

3.5.6 How Do I Place Variables in Off-Chip Memory?

To locate variables in off-chip memory, use the `__external__` type qualifier. To locate variables in off-chip memory across a Parallel Master Port (PMP), use the `__pmp__` type qualifier.

The time required to access these variables is longer than for variables in the internal data memory. For details on accessing off-chip memory, see the following sections in this user's guide:

Section 8.11.6 “`__external__` Type Qualifier”

Section 8.11.5 “`__pmp__` Type Qualifier”

3.5.7 How Can I Implement a Delay in My Code?

Using a device timer to generate a delay is the best method. If no time is available, then you can use the library functions `_delay32`, `__delay_ms`, or `__delay_us`, as they are described in the following document. It is available for download from the Microchip Technology website, www.microchip.com.

“16-Bit Language Tools Libraries Reference Manual” (DS50001456)

3.5.8 How Can I Rotate a Variable?

The C language does not have a rotate operator, but rotations can be performed using the shift and bitwise OR operators. For more information, see the following sections in this user's guide:

Section 2.4.10 “Bit-wise Operations on Signed Values” (CCI)

Section 2.4.11 “Right-shifting Signed Values” (Signed variables)

3.6 UNDERSTANDING THE COMPILATION PROCESS

This section tells you how to find out what the compiler did during the build process, how it encoded output code, where it placed objects, etc. It also discusses the features that are supported by the compiler.

- How Does Licensing Affect Features and Optimization Levels?
- How Can I Make My Code Smaller?
- How Can I Reduce RAM Usage?
- How Can I Make My Code Faster?
- How Can I Control Where the Language Tool Places Objects in Memory?
- How Can I Make My Interrupt Routine Faster?
- How Big Can C Variables Be?
- Which Optimizations Will Be Applied to My Code?
- Which Devices are Supported by the Compiler?
- How Do I Know What Code the Compiler Is Producing?
- How Can I Tell How Big a Function Is?
- How Do I Learn Where Variables and Functions Have Been Positioned?
- How Do I Properly Reserve Memory?
- How Do I Know How Much Memory Is Still Available?
- Which Libraries Get Included by Default?
- How Do I Create My Own Libraries?
- Why Do I Get Out-of-memory Errors When I Select a Debugger?
- *See also*, How Do I Find Out What a Warning or Error Message Means?
- *See also*, How Do I Build Libraries?
- *See also*, What is Different About an MPLAB X IDE Debug Build?
- *See also*, How Do I Stop A Function From Being Removed?

3.6.1 How Does Licensing Affect Features and Optimization Levels?

Different licenses vary in the features and optimizations available. See the following chapter in this user's guide:

Chapter 18. "Optimizations"

3.6.2 How Can I Make My Code Smaller?

General advice for creating smaller code:

- Do not mix data types.
- Define index variables in the native word width.
- Don't use floating-point variables when integers will suffice. When using floating-point variables, consider using the smaller math libraries.
- Compiler option `-mpa` can be combined with any optimization level to reduce code size. See the following chapter in this user's guide:

Chapter 18. "Optimizations"

Note: Optimized code may be more difficult to debug.

- When initializing an SFR, use the full name of the SFR instead of bit names. You can initialize several fields at once with this technique.
- Instead of copying code literally into several places in your program, reorganize the shared code into functions.
- Use the small code, small scalar, and large data memory models. Refer to the following section in this user's guide:

Section 10.12 "Memory Models"

3.6.3 How Can I Reduce RAM Usage?

Try the following suggestions to reduce RAM usage:

1. Some of the same suggestions for making your code smaller can be useful to reduce RAM usage, see the following section in this user's guide:

Section 3.6.2 "How Can I Make My Code Smaller?"

2. Rather than pass large objects to (or from) functions, pass pointers that reference these objects (to save stack resources).
3. Objects that do not need to change throughout the program can be located in program memory using the `-mconst-in-code` option and the `const` qualifier. This frees up precious RAM, but slows execution. Refer to the following section in this user's guide:

Section 10.4 "Variables in Program Space"

3.6.4 How Can I Make My Code Faster?

Try the following suggestions for faster code execution:

1. Smaller code can be faster code. Reducing the number of machine instructions that are necessary to perform a task will result in faster execution of that task. For details on making smaller and faster code, see the following sections in this user's guide:

Section 3.6.2 "How Can I Make My Code Smaller?"

Section 3.6.6 "How Can I Make My Interrupt Routine Faster?"

2. Depending on your compiler license, you may be able to use increasing optimization levels to generate faster code. For details, see the following chapter and section in this user's guide:

Chapter 18. "Optimizations"

Section 5.7.6 "Options for Controlling Optimization"

3. Algorithm choice has more impact on size and speed of your solution than any other factor. Choose the right algorithm for the job.

3.6.5 How Can I Control Where the Language Tool Places Objects in Memory?

In most situations, you should allow the language tool to place objects in memory. If you still want to place objects, consult the following section in this user's guide for details:

Section 3.4.3 “Memory Allocation”

3.6.6 How Can I Make My Interrupt Routine Faster?

Try the following suggestions for faster ISR execution:

1. Smaller code is often faster code. For details, see the following section in this user's guide:

Section 3.6.2 “How Can I Make My Code Smaller?”

2. Suggestions to make code faster also work for ISR code. For details, refer to the following section in this user's guide:

Section 3.6.4 “How Can I Make My Code Faster?”

3. Consider having the ISR simply set a flag and return. The flag can then be checked in main-line code to handle the interrupt. This has the advantage of moving the complicated interrupt-processing code out of the ISR so that it no longer contributes to its register usage. Always use the `volatile` qualifier. For variables shared by the interrupt and main-line code, see the following sections in this user's guide:

Section 8.10.2 “Volatile Type Qualifier”

Section 3.5.3 “How Do I Share Data Between Interrupt and Main-line Code?”

3.6.7 How Big Can C Variables Be?

This question specifically relates to the size of individual C objects, such as arrays or structures. The total size of all variables is another matter.

To answer this question you need to know the memory space in which the variable is to be located. When using the `-mconst-in-code` option, objects qualified `const` will be located in program memory, other objects will be placed in data memory. Program memory object sizes are discussed in the following section of this user's guide:

Section 10.4.3 “Size Limitations of Program Memory Variables”

Objects in data memory are broadly grouped into “autos” and “non-autos”. These objects have size limitations. For more on auto and non-auto variables and the size limitations, see the following sections in this user's guide:

Section 10.3 “Variables in Data Space Memory”

Section 10.3.2.3 “Auto Variable Size Limits”

Section 10.3.1.3 “Non-Auto Variable Size Limits”

3.6.8 Which Optimizations Will Be Applied to My Code?

Code optimizations available depend on your compiler license. For more information, refer to the following chapter and section in this user's guide:

Chapter 18. “Optimizations”

Section 5.7.6 “Options for Controlling Optimization”

3.6.9 Which Devices are Supported by the Compiler?

Support for new devices usually occurs with each compiler release. To learn whether a device is supported by your compiler, see the following section in this user's guide:

Section 6.2 "Device Support"

3.6.10 How Do I Know What Code the Compiler Is Producing?

The assembly list file can be set up (using assembler listing file options) to contain a variety of information about the code. That information could include assembly output for almost the entire program, library routines linked in to your program, section information, symbol listings, and more.

The list file can be produced as follows:

- On the command line, create a basic list file using the option:
`-Wa, -a=projectname.lst`
- For MPLAB X IDE, right click on your project and select "Properties". In the Project Properties window, click on "xc16-as" under "Categories". From "Option categories", select "Listing file options" and ensure "List to file" is checked.

By default, the assembly list file will have a `.lst` extension.

For information on the list file, refer to the following document. It is available for download from the Microchip Technology website, www.microchip.com.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106).

3.6.11 How Can I Tell How Big a Function Is?

This size of a function (the amount of assembly code generated for that function) can be determined from the assembly list file. See the following section in this user's guide:

Section 3.6.10 "How Do I Know What Code the Compiler Is Producing?"

3.6.12 How Do I Learn Where Variables and Functions Have Been Positioned?

The `xc16-objdump` utility displays information about one or more object files. Use the `-t` option to print the symbol table entries of a file.

Also, you can determine where variables and functions have been positioned from the map file generated by the linker. Only global symbols are shown in the map file.

There is a mapping between C identifiers and the symbols used in assembly code. The symbol associated with a variable is assigned the address of the lowest byte of the variable; for functions it is the address of the first instruction generated for that function. For more on `xc16-objdump` and linker map files, refer to the following document. It is available for download from the Microchip Technology website, www.microchip.com.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106)

3.6.13 How Do I Properly Reserve Memory?

Memory may be reserved by creating a specific section in the linker script or by using attributes to block out sections of memory. If you have not used one of these methods to reserve memory, you may not be reserving the memory you thought you were, and the linker may be placing objects in this area. For more on reserving memory, consult the following document. It is available for download from the Microchip Technology website, www.microchip.com. Also, see the following section in this user's guide.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106)

Section 3.4.3.4 "How Do I Stop the Compiler from Using Certain Memory Locations?"

3.6.14 How Do I Know How Much Memory Is Still Available?

A memory usage summary is available from the compiler after compilation (`--report-mem` option) or from MPLAB X IDE in the Dashboard window. All of these summaries indicate the amount of memory used and the amount still available, but none indicate whether this memory is one contiguous block or broken into many small chunks. Since small blocks of free memory cannot be used for larger objects, out-of-memory errors may be produced even though the total amount of memory free is apparently sufficient for the objects to be positioned.

Consult the linker map file to determine exactly which memory is still available in each linker class. This file also indicates the largest contiguous block in that class, if there are memory page divisions. See the following document for information on the map file. It is available for download from the Microchip website, www.microchip.com.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106)

3.6.15 Which Libraries Get Included by Default?

The compiler automatically includes any applicable standard library into the build process when you compile. So, you never need to control these files. However, there are some libraries you must remember to include, such as any libraries that do not come with the compiler. One can tell which standard libraries have been used in the resulting compiled image by inspecting the MAP file. Archive members included from the standard library will be in a listing that is associated with the symbol that prompted the inclusion of a particular standard library archive within the MAP. For details on standard libraries, consult the following document. It is available for download from the Microchip Technology website, www.microchip.com.

"16-Bit Language Tools Libraries Reference Manual" (DS50001456)

3.6.16 How Do I Create My Own Libraries?

To use one or more library files that were built by yourself or a colleague, include them in the list of files being compiled on the command line. The library files can be specified in any position in the file list, relative to the source files. However, if there is more than one library file, they will be searched in the order specified in the command line.

An example of specifying the library `liblibrary.a` on the command line is:

```
xc16-gcc -mcpu=33FJ256GP710 main.c int.c liblibrary.a
```

If you want to use the `-l` option, then:

```
xc16-gcc -mcpu=33FJ256GP710 main.c int.c -llibrary
```

If you are using MPLAB X IDE to build a project, add the library file(s) to the Libraries folder that is in your project, and in the order in which they should be searched. The IDE will ensure that they are passed to the compiler at the appropriate point in the build sequence. For information on how you build your own library files, see the following section in this user's guide:

Section 5.4.1.2 "User-Defined libraries"

3.6.17 Why Do I Get Out-of-memory Errors When I Select a Debugger?

If you use a hardware-tool debugger - such as PICKit 3 in-circuit debugger, MPLAB ICD 3 in-circuit debugger, or MPLAB REAL ICE in-circuit emulator, RAM is required for debugging. See the following section in this user's guide:

Section 3.5.2 "Why No Disassembly in the MPLAB X IDE Disassembly Window?"

3.7 FIXING CODE THAT DOES NOT WORK

This section examines issues relating to projects that do not build due to compiler errors;, or which projects do build, but do not work as expected.

- How Do I Find Out What a Warning or Error Message Means?
- How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?
- How Can I Stop Warnings from Being Produced?
- How Do I Know If the Stack Has Overflowed?
- What Can Cause Corrupted Variables and Code Failure When Using Interrupts?
- See *also*, Invoking the Compiler
- See *also*, How Do I Properly Reserve Memory?

3.7.1 How Do I Find Out What a Warning or Error Message Means?

Most warning and error messages are self-explanatory; however, some require an additional discussion. All MPLAB XC16 warning and error messages are discussed in the appendix referenced below. Additionally, a discussion of how to control message output is included in the following section of this user's guide:

Appendix C. "Diagnostics"

Section 5.7.4 "Options for Controlling Warnings and Errors"

3.7.2 How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?

In most instances where the error is a syntax error relating to the source code, the message produced by the compiler indicates the offending line of code. If you are compiling in MPLAB X IDE, you can double click the message and have the editor take you to the offending line. But identifying the offending code is not always so easy.

In some instances, the error is reported on the line of code following the line that needs attention. This is because a C statement is allowed to extend over multiple lines of the source file. It is possible that the compiler may not be able to determine that there is an error until it has started to scan the next statement. Consider the following code:

```
input = PORTB    // oops - forgot the semicolon
if(input>6)
    // ...
```

The missing semicolon on the assignment statement has been flagged on the following line that contains the `if()` statement.

In other cases, the error might come from the assembler, not the compiler. If the source being compiled is an assembly module, the error directly indicates the line of assembly code that triggered the error.

Finally, there are errors that do not relate to any particular line of code at all. An error in a compiler option or a linker error are examples of these.

If you need to see the assembly code generated by the compiler, look in the assembly list file. For information on where the linker attempted to position objects, see the map file. Consult the following document for information on the list and map files. It is available for download from the Microchip Technology website, www.microchip.com.

"MPLAB XC16 Assembler, Linker and Utilities User's Guide" (DS52106)

3.7.3 How Can I Stop Warnings from Being Produced?

In general, you should not ignore warnings. Warnings indicate situations that could possibly lead to code failure. Always check your code to confirm that it is not a possible source of error.

However, if you feel that you want to inhibit warning messages, do the following:

- Inhibit specific warnings by using the `-Wno-` version of the option.
- Inhibit all warnings with the `-w` option.
- In MPLAB X IDE, inhibit warnings in the Project Properties window under each tool category. Also look in the Tool Options window, Embedded button, Suppressible Messages tab.

For details, see the following section in this user's guide:

Section 5.7.4 “Options for Controlling Warnings and Errors”.

3.7.4 How Do I Know If the Stack Has Overflowed?

The 16-bit devices use a stack that its upper address boundary can be set in the SPLIM register. Therefore, it is possible to set a stack level to prevent overflow.

Other stack errors, besides overflow, may be trapped and identified in code. For more information about using the software stack, see the following sections in this guide:

Section 10.3.2.1 “Software Stack”

Section 10.3.2.2 “The C Stack Usage”

Section on the software stack in your device data sheet

3.7.5 What Can Cause Corrupted Variables and Code Failure When Using Interrupts?

This is usually caused by having variables used in both interrupt and main-line code. If the compiler optimizes access to a variable, or access is interrupted by an interrupt routine, then corruption can occur. See the following section in this user's guide:

Section 3.5.3 “How Do I Share Data Between Interrupt and Main-line Code?”

Chapter 4. XC16 Toolchain and MPLAB X IDE

4.1 INTRODUCTION

The 16-bit language tools may be used together under MPLAB X IDE to provide GUI development of application code for the dsPIC[®] DSC and PIC24 MCU families of devices. The tools are:

- MPLAB XC16 C Compiler
- MPLAB XC16 Assembler
- MPLAB XC16 Object Linker
- MPLAB XC16 Object Archiver/Librarian and other 16-bit utilities

Topics covered in this chapter:

- MPLAB X IDE and Tools Installation
- MPLAB X IDE Setup
- MPLAB X IDE Projects
- Project Setup
- Project Example

4.2 MPLAB X IDE AND TOOLS INSTALLATION

In order to use the 16-bit language tools with MPLAB X IDE, you must install:

- MPLAB X IDE, which is available for free on the Microchip website.
- MPLAB XC16 C Compiler, which includes all of the 16-bit language tools. The compiler is available for free (Free and Evaluation editions) or for purchase (Standard or Pro editions) on the Microchip website.

The 16-bit language tools will be installed, by default, in the directory:

- Windows OS 32-bit - C:\Program Files\Microchip\xc16\x.xx
- Windows OS 64-bit - C:\Program Files (x86)\Microchip\xc16\x.xx
- Mac OS - Applications/microchip/xc16/x.xx
- Linux OS - /opt/microchip/xc16/x.xx

where *x.xx* is the version number.

The executables for each tool will be in the `bin` subdirectory:

- C Compiler - `xc16-gcc.exe`
- Assembler - `xc16-as.exe`
- Object Linker - `xc16-ld.exe`
- Object Archiver/Librarian - `xc16-ar.exe`
- Other Utilities - `xc16-utility.exe`

All device include (header) files are in `support/family/h` subdirectories, where *family* is the device family for your selected device (e.g., dsPIC30F is the device family for the dsPIC30F6015 device). For more on these files, see **Section 6.3 “Device Header Files”**.

MPLAB® XC16 C Compiler User's Guide

All device linker script files are in `support/family/gld` subdirectories, where *family* is the device family for your selected device (e.g., dsPIC30F is the device family for the dsPIC30F6015 device). For more on these files, see the linker documentation.

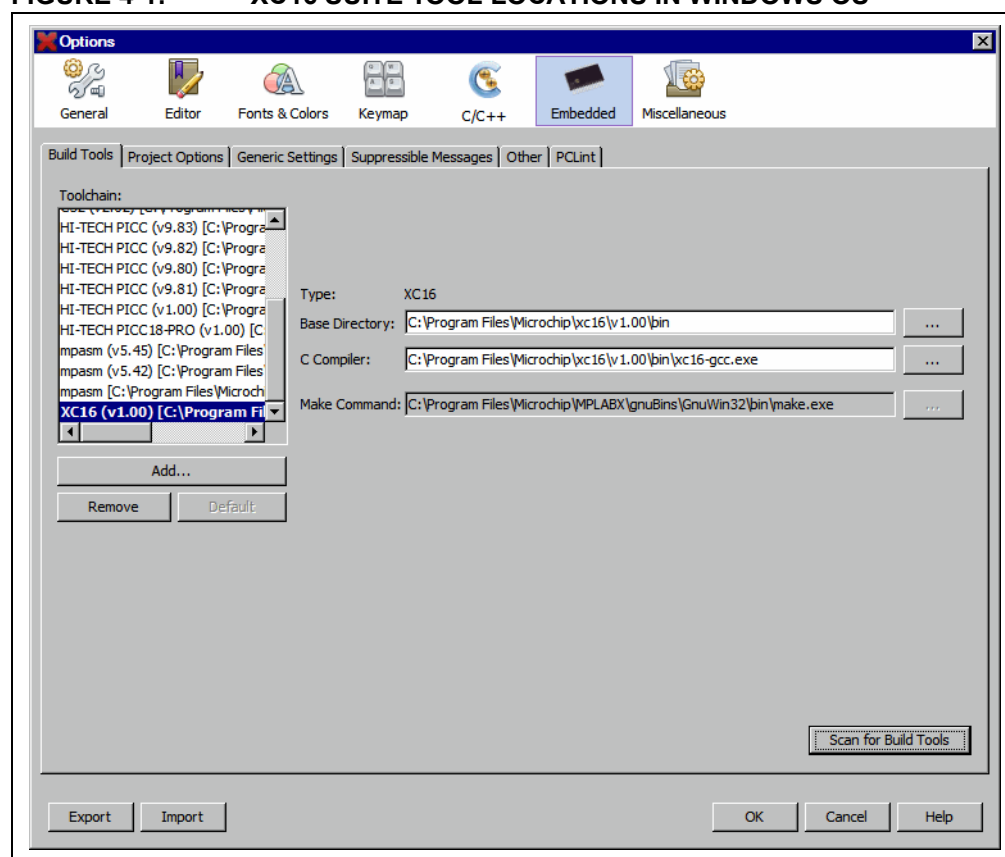
Code examples and template files are in the `support/templates/assembly` subdirectory.

4.3 MPLAB X IDE SETUP

Once MPLAB X IDE is installed on your PC, launch the application and check the settings below to ensure that the 16-bit language tools are properly recognized.

1. From the MPLAB X IDE menu bar, select **Tools>Options** to open the Options dialog. Click on the “Embedded” button and select the “Build Tools” tab.
2. Click on “XC16” under “Tool Collection”. Ensure that the paths are correct for your installation.
3. Click **OK**.

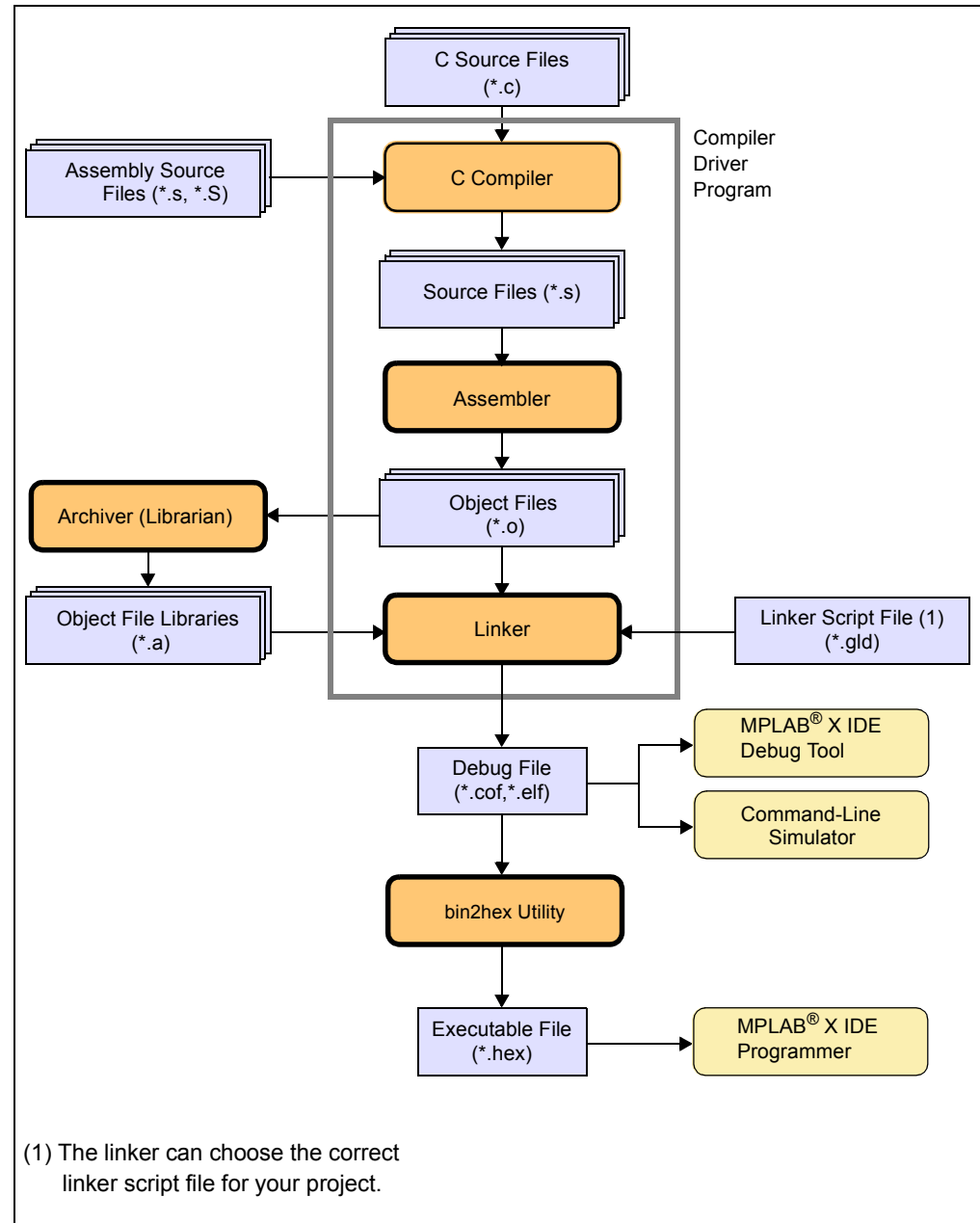
FIGURE 4-1: XC16 SUITE TOOL LOCATIONS IN WINDOWS OS



4.4 MPLAB X IDE PROJECTS

A project in MPLAB X IDE is a group of files needed to build an application, along with their associations to various build tools. Below is a generic MPLAB X IDE project.

FIGURE 4-2: COMPILER PROJECT RELATIONSHIPS



In this MPLAB X IDE project, C source files are shown as input to the compiler. The compiler will generate source files for input into the assembler. For more information on the compiler, see the compiler documentation.

Assembly source files are shown as input to the C preprocessor. The resulting source files are input to the assembler. The assembler will generate object files for input into the linker or archiver. For more information on the assembler, see the assembler documentation.

Object files can be archived into a library using the archiver/librarian. For more information on the archiver, see the archiver/librarian documentation.

The object files and any library files, as well as a linker script file (generic linker scripts are added automatically), are used to generate the project output files via the linker. The output file generated by the linker is either an ELF or COF file used by the simulator and debug tools. This file may be input into the bin2hex utility to produce an executable file (`.hex`). For more information on linker script files and using the object linker, see the linker documentation.

For more on projects, see MPLAB X IDE documentation.

4.5 PROJECT SETUP

To set up an MPLAB X IDE project for the first time, use the built-in Project Wizard (*File>New Project*). In this wizard, you will be able to select a language toolsuite that uses the 16-bit language tools. For more on the wizard and MPLAB X IDE projects, see MPLAB X IDE documentation.

Once you have a project set up, you may then set up properties of the tools in MPLAB X IDE.

1. From the MPLAB X IDE menu bar, select *File>Project Properties* to open a window to set/check project build options.
2. Under “Conf:[default]”, select a tool from the tool collection to set up.
 - XC16 (Global Options)
 - xc16-as (16-Bit Assembler)
 - xc16-gcc (16-Bit C Compiler)
 - xc16-ld (16-Bit Linker)

4.5.1 XC16 (Global Options)

Set up global options for all 16-bit language tools. See also “Options Page Features”.

TABLE 4-1: ALL OPTIONS CATEGORY

Option	Description	Command Line
Output file format	Select either ELF/DWARF or COFF.	-omf=elf -omf=cof
Define common macros	Add macros common to compiler, assembler and linker.	-Dmacro
Generic build	Build for a generic core device (no peripherals).	
Use legacy lib	Check to use libraries in the format before v3.25. Uncheck to use the new (HI-TECH) libraries format.	-legacy-libc
Fast floating point math	Check to use faster single and double floating point libraries, which consume more RAM. Uncheck to use original libraries which are slower but create smaller code.	-fast-math
Don't delete intermediate files	Check to not delete intermediate files. Place them in the object directory and name them based on the source file. Uncheck to remove intermediate files after a build.	-save-temps=obj

4.5.2 xc16-as (16-Bit Assembler)

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up assembler options. For additional options, see MPLAB XC16 Assembler documentation. See also “Options Page Features”.

TABLE 4-2: GENERAL OPTIONS CATEGORY

Option	Description	Command Line
Define ASM macros (.S only)	Add assembler macros.	-Dmacro
Assembler symbols	Define symbol 'sym' to a given 'value'.	--defsym sym=value
ASM include dirs	Add a directory to the list of directories the assembler searches for files specified in .include directives. For more information, see Section 4.5.6 “Additional Search Paths and Directories” .	-I"dir"
Preprocessor include dirs	Add a directory to the list of directories the compiler preprocessor searches for files specified in .include directives. For more information, see Section 4.5.6 “Additional Search Paths and Directories” .	-I"dir"

MPLAB® XC16 C Compiler User's Guide

TABLE 4-2: GENERAL OPTIONS CATEGORY

Option	Description	Command Line
Allow CALL optimization	Check to turn relaxation on. Uncheck to turn relaxation off.	--relax --no-relax
Keep local symbols	Check to keep local symbols, i.e., labels beginning with .L (upper case only). Uncheck to discard local symbols.	--keep-locals (-L)
Diagnostics level	Select warnings to display in the Output window. - Generate warnings - Suppress warnings - Fatal warnings	--warn --no-warn --fatal-warnings
Additional driver options	Enter any additional driver options not existing in the GUI. The string you introduce here will be emitted as-is in the driver invocation command.	

TABLE 4-3: LISTING FILE OPTIONS CATEGORY

Option	Description	Command Line
Include source code	Check for a high-level language listing. High-level listings require that the assembly source code is generated by a compiler, a debugging option like -g is given to the compiler, and assembly listings (-al) are requested. Uncheck for a regular listing.	-ah
Include macros expansions	Check to expand macros in a listing. Uncheck to collapse macros.	-am
Omit false conditionals	Check to omit false conditionals (.if, .ifdef) in a listing. Uncheck to include false conditionals.	-ac
Omit forms processing	Check to turn off all forms processing that would be performed by the listing directives .psize, .eject, .title and .sbttl. Uncheck to process by listing directives.	-an
Include assembly	Check for an assembly listing. This -a suboption may be used with other suboptions. Uncheck to exclude an assembly listing.	-al
Include symbols	Check for a symbol table listing. Uncheck to exclude the symbol table from the listing.	-as
Omit debugging directives	Check to omit debugging directives from a listing. This can make the listing cleaner. Uncheck to included debugging directives.	-ad
Include section information	Check to display information on each of the code and data sections. This information contains details on the size of each of the sections and then a total usage of program and data memory. Uncheck to not display this information.	-ai

4.5.3 xc16-gcc (16-Bit C Compiler)

Although the MPLAB XC16 C Compiler works with MPLAB X IDE, it must be acquired separately. The full version may be purchased, or a student (limited-feature) version may be downloaded for free. See the Microchip website (www.microchip.com) for details.

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up compiler options. For additional options, see the *MPLAB XC16 C Compiler User's Guide* (DS51284), also available on the Microchip website.

See also "Options Page Features".

TABLE 4-4: PREPROCESSING AND MESSAGES CATEGORY

Option	Description	Command Line
Include C dirs	Add the directory <i>dir</i> to the head of the list of directories to be searched for header files. For more information, see Section 4.5.6 "Additional Search Paths and Directories" .	<code>-I"dir"</code>
Define C macros	Define macro <i>macro</i> with the string 1 as its definition.	<code>-Dmacro</code>
ANSI-std C support	Check to support all (and only) ASCII C programs. Uncheck to support ASCII and non-ASCII programs.	<code>-ansi</code>
Errata	This option enables specific errata work-arounds identified by ID. Valid values for ID change from time to time and may not be required for a particular variant. The ID <code>all</code> will enable all currently supported errata work arounds. The ID <code>list</code> will display the currently supported errata identifiers along with a brief description of the errata.	<code>-merrata=id</code>
Smart IO forwarding level	This option attempts to statically analyze format strings passed to <code>printf</code> , <code>scanf</code> and the 'f' and 'v' variations of these functions. Uses of nonfloating point format arguments will be converted to use an integer-only variation of the library functions. Equivalent to <code>-msmart-io=n</code> option where <i>n</i> equals: <ul style="list-style-type: none"> • 0 - disables this option. • 1 - only convert the literal values it can prove. • 2 - causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. 	<code>-msmart-io=n</code>
Smart IOformat strings	Specifies what the format arguments are when the compiler is unable to determine them.	
Make warnings into errors	Check to halt compilation based on warnings as well as errors. Uncheck to halt compilation based on errors only.	<code>-Werror</code>
Additional warnings	Check to enable all warnings. Uncheck to disable warnings.	<code>-Wall</code>
Strict ANSI warnings	Check to issue all warnings demanded by strict ANSI C. Uncheck to issue all warnings.	<code>-pedantic</code>
Disable ISR warn	Disable warning for inappropriate use of ISR function names. By default the compiler will produce a warning if the <code>__interrupt__</code> is not attached to a recognized interrupt vector name. This option will disable that feature.	<code>-mno-isr-warn</code>
Enable SFR warnings	Enable warnings related to SRFs.	<code>-msfr-warn=on off</code>

TABLE 4-5: GENERAL CATEGORY

Option	Description	Command Line
Generate debugging information	Create a COFF or ELF file with information to allow debugging of code in MPLAB X IDE. Note: COFF supports debugging in the <code>.text</code> section only.	<code>-g</code>
Isolate each function in a section	Check to place each function into its own section in the output file. The name of the function determines the section's name in the output file. Note: When you specify this option, the assembler and linker may create larger object and executable files and will also be slower. Uncheck to place multiple functions in a section.	<code>-ffunction-sections</code>
Place data into its own section	Place each data item into its own section in the output file. The name of the data item determines the name of the section. When you specify this option, the assembler and linker may create larger object and executable files and will also be slower.	<code>-fdata-sections</code>
Use 64-bit double	Use <code>long double</code> instead of <code>double</code> type equivalent to float. Mixing this option across modules can have unexpected results if modules share double data either directly through argument passage or indirectly through shared buffer space.	<code>-fno-short-double</code>
Fillupper value for data in flash	Full upper flash memory with the value specified.	<code>-mfillupper=value</code>
Name the text section	Place text (program code) in a section named <i>name</i> rather than the default <code>.text</code> section.	<code>-mtext=name</code>

TABLE 4-6: MEMORY MODEL CATEGORY

Option	Description	Command Line
Code Model	Select a code (program memory/ROM) model. - default - large (>32Kwords) - small (≤32Kwords)	<code>-msmall-code</code> <code>-mlarge-code</code> <code>-msmall-code</code>
Data Model	Select a data (data memory/RAM) model. - default - large (>8KB) - small (≤8KB)	<code>-msmall-data</code> <code>-mlarge-data</code> <code>-msmall-data</code>
Scalar Model	Select a scalar model. - default - large (>8KB) - small (≤8KB)	<code>-msmall-scalar</code> <code>-mlarge-scalar</code> <code>-msmall-scalar</code>
Location of Constants	Select a memory location for constants. - default - Data - Code	<code>-mconst-in-code</code> <code>-mconst-in-data</code> <code>-mconst-in-code</code>
Place all code in auxiliary flash	Place all code from the current translation unit into auxiliary Flash. This option is only available on devices that have auxiliary Flash.	<code>-mauxflash</code>

TABLE 4-6: MEMORY MODEL CATEGORY (CONTINUED)

Option	Description	Command Line
Put constants into auxiliary flash	When combined with <code>-mconst-in-code</code> , put constants into auxiliary Flash.	<code>-mconst-in-auxflash</code>
Put char vars into near data space	Place <code>char</code> variables into near data space, regardless of memory model.	<code>-mnear-chars</code>
Allow arrays larger than 32K	Allow arrays that are larger than 32K, regardless of memory model.	<code>-menable-large-arrays</code>
Aggregate data model	Use aggregate data model.	<code>-mlarge-aggregate</code>

TABLE 4-7: OPTIMIZATION CATEGORY

Option	Description	Command Line
Optimization Level	Select an optimization level. Your compiler edition may support only some optimizations. Equivalent to <code>-On</code> option, where <i>n</i> is an option below: <ul style="list-style-type: none"> 0 - Do not optimize. The compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. 1 - Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. The compiler tries to reduce code size and execution time. 2 - Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. 3 - Optimize yet more favoring speed (superset of O2). s - Optimize yet more favoring size (superset of O2). 	<code>-On</code>
Unroll loops	Check to perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. Uncheck to not unroll loops.	<code>-funroll-loops</code>
Omit frame pointer	Check to not keep the Frame Pointer in a register for functions that don't need one. Uncheck to keep the Frame Pointer.	<code>-fomit-frame-pointer</code>
Unlimited procedural abstraction	Enable the procedure abstraction optimization. There is no limit on the nesting level.	<code>-mpa</code>
Procedural abstraction	Enable the procedure abstraction optimization up to level <i>n</i> . Equivalent to <code>-mpa=<i>n</i></code> option, where <i>n</i> equals: <ul style="list-style-type: none"> 0 - Optimization is disabled. 1 - The first level of abstraction is allowed; that is, instruction sequences in the source code may be abstracted into a subroutine. 2 or greater - A second level of abstraction is allowed; that is, instructions that were put into a subroutine in the first level may be abstracted into a subroutine one level deeper. This pattern continues for larger values of <i>n</i>. The net effect is to limit the subroutine call nesting depth to a maximum of <i>n</i>. 	<code>-mpa=<i>n</i></code>
Align arrays	Set the minimum alignment for array variables to be the largest power of two less than or equal to their total storage size, or the biggest alignment used on the machine, whichever is smaller.	<code>-falign-arrays</code>

4.5.4 xc16-ld (16-Bit Linker)

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up linker options. For additional options, see MPLAB Object Linker for 16-Bit Devices documentation. See also “Options Page Features”.

TABLE 4-8: GENERAL CATEGORY

Option	Description	Command Line
Heap Size (bytes)	Specify the size of the heap in bytes. Allocate a run-time heap of size bytes for use by C programs. The heap is allocated from unused data memory. If not enough memory is available, an error is reported.	<code>--heap size</code>
Stack Size (bytes)	Specify the minimum size of the stack in bytes. By default, the linker allocates all unused data memory for the run-time stack. Alternatively, the programmer may allocate the stack by declaring two global symbols: <code>__SP_init</code> and <code>__SPLIM_init</code> . Use this option to ensure that at least a minimum sized stack is available. The actual stack size is reported in the link map output file. If the minimum size is not available, an error is reported.	<code>--stack size</code>
Allow overlapped sections	Check to not check section addresses for overlaps. Uncheck to check for overlaps.	<code>--check-sections</code> <code>--no-check-sections</code>
Initialize data sections	Check to support initialized data. Uncheck to not support.	<code>--data-init</code> <code>--no-data-init</code>
Pack data template	Check to pack initial data values. Uncheck to not pack.	<code>--pack-data</code> <code>--no-pack-data</code>
Create handles	Check to support far code pointers. Uncheck to not support.	<code>--handles</code> <code>--no-handles</code>
Create default ISR	Check to create an interrupt function for unused vectors. Uncheck to not create a default ISR.	<code>--isr</code> <code>--no-isr</code>
Remove unused sections	Check to not enable garbage collection of unused input sections (on some targets). Uncheck to enable garbage collection.	<code>--no-gc-sections</code> <code>--gc-sections</code>
Fill value for upper byte of data	Enter a fill value for upper byte of data. Use this value as the upper byte (bits 16-23) when encoding data into program memory. This option affects the encoding of sections created with the <code>psv</code> or <code>eedata</code> attribute, as well as the data initialization template if the <code>--no-pack-data</code> option is enabled.	<code>--fill-upper=value</code>
Stack guardband size	Enter a stack guardband size to ensure that enough stack space is available to process a stack overflow exception.	<code>--stackguard=size</code>
Additional driver options	Type here any additional driver options not existing in this GUI otherwise. The string you introduce here will be emitted as is in the driver invocation command.	

TABLE 4-9: SYMBOLS AND MACROS CATEGORY

Option	Description	Command Line
Linker symbols	Create a global symbol in the output file containing the absolute address (<i>expr</i>). You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the <i>expr</i> in this context: you may give a hexadecimal constant or the name of an existing symbol, or use + and - to add or subtract hexadecimal constants or symbols.	<code>--defsym=sym</code>
Define Linker macros	Add linker macros.	<code>-Dmacro</code>
Symbols	Specify symbol information in the output. - Keep all - Strip debugging info - Strip all symbol info	<code>—</code> <code>--strip-debug (-S)</code> <code>--strip-all (-s)</code>

TABLE 4-10: LIBRARIES CATEGORY

Option	Description	Command Line
Libraries	Add libraries to be linked with the project files. You may add more than one.	<code>--library=name</code>
Library directory	Add a library directory to the library search path. You may add more than one.	<code>--library-path="name"</code>
Force linking of objects that might not be compatible	Check to force linking of objects that might not be compatible. The linker will compare the project device to information contained in the objects combined during the link. If a possible conflict is detected, an error (in the case of a possible instruction set incompatibility) or a warning (in the case of possible register incompatibility) will be reported. Specify this option to override such errors or warnings. Uncheck to not force linking.	<code>--force-link</code> <code>--no-force-link</code>
Don't merge I/O library functions	Check to not merge I/O library functions. Do not attempt to conserve memory by merging I/O library function calls. In some instances the use of this option will increase memory usage. Uncheck to merge I/O library functions to conserve memory.	<code>--no-smart-io</code> <code>--smart-io</code>
Exclude standard libraries	Check to not use the standard system startup files or libraries when linking. Only use library directories specified on the command line. Uncheck to use the standard system startup files and libraries.	<code>--nostdlib</code>

TABLE 4-11: DIAGNOSTICS CATEGORY

Option	Description	Command Line
Generate map file	Create a map file.	<code>-Map="file"</code>
Display memory usage	Check to print memory usage report. Uncheck to not print a report.	<code>--report-mem</code>
Generate cross-reference file	Check to create a cross-reference table. Uncheck to not create this table.	<code>--cref</code>

MPLAB® XC16 C Compiler User's Guide

TABLE 4-11: DIAGNOSTICS CATEGORY (CONTINUED)

Option	Description	Command Line
Warn on section realignment	Check to warn if start of section changes due to alignment. Uncheck to not warn.	--warn-section-align
Trace Symbols	Add/remove trace symbols.	--trace-symbol= <i>symbol</i>

TABLE 4-12: CODE GUARD CATEGORY

Option	Description	Command Line
Boot RAM	Specify the boot RAM segment: none, small, medium or large.	--boot= <i>option_ram</i>
Boot Flash	Specify the boot Flash segment: none, small, medium, or large standard or none, small, medium, or large high.	--boot= <i>option_flash_std</i> --boot= <i>option_flash_high</i>
Boot EEPROM	Specify the boot EEPROM segment.	--boot=eeprom
Boot write-protect	Specify the boot write protected segment.	--boot=write_protect
Secure RAM	Specify the secure RAM segment: none, small, medium or large.	--secure= <i>option_ram</i>
Secure Flash	Specify the secure Flash segment: none, small, medium, or large standard or none, small, medium, or large high.	--secure= <i>option_flash_std</i> --secure= <i>option_flash_high</i>
Secure EEPROM	Specify the secure EEPROM segment.	--secure=eeprom
Secure write-protect	Specify the secure write protected segment.	--secure=write_protect
General write-protect	Specify the general write protected segment.	--general=write_protect
General code-protect	Specify the secure code protected segment: standard or high.	--general=code_protect_std --general=code_protect_high

For more information on CodeGuard™ options, see “Options that Specify CodeGuard Security Features” in the linker documentation.

Note: Not all development tools support CodeGuard programming. See tool documentation for more information.

4.5.5 Options Page Features

The Options section of the Properties page has the following features for all tools:

TABLE 4-13: PAGE FEATURES OPTIONS

Reset	Reset the page to default values.
Additional options	Enter options in a command-line (non-GUI) format.
Option Description	Click on an option name to see information on the option in this window. Not all options have information in this window.
Generated Command Line	Click on an option name to see the command-line equivalent of the option in this window.

4.5.6 Additional Search Paths and Directories

For the compiler, assembler and linker, you may set additional paths to directories to be searched for include files and libraries.

You may add as many directories as necessary to include a variety of paths. The current working directory is always searched first and then the additional directories in the order in which they were specified.

All paths specified should be relative to the project directory, which is the directory containing the `nbproject` directory.

4.6 PROJECT EXAMPLE

In this example, you will create an MPLAB X IDE project with two C code files.

- Run the Project Wizard
- Add Files to the Project
- Set Build Options
- Build the Project
- Output Files
- Further Development

4.6.1 Run the Project Wizard

In MPLAB X IDE, select *File>New Project* to launch the wizard.

1. **Choose Project:** Select “Microchip Embedded” for the category and “Stand-alone Project” for the project. Click **Next>** to continue.
2. **Select Device:** Select the dsPIC30F6014. Click **Next>** to continue.
3. **Select Header:** There is no header for this device so this is skipped.
4. **Select Tool:** Choose a development tool from the list. Tool support for the selected device is shown as a colored circle next to the tool. Mouse over the circle to see the support as text. Click **Next>** to continue.
5. **Select Compiler:** Choose a version of the XC16 toolchain. Click **Next>** to continue.
6. **Select Project Name and Folder:** Enter a project name, such as `MyXC16Project`. Then select a location for the project folder. Click **Finish** to complete the project creation and setup.

Once the Project Wizard has completed, the Project window should contain the project tree. For more on projects, see the MPLAB X IDE documentation.

4.6.2 Add Files to the Project

To add the C code files `tmp6014.c` and `traps.c` to the project:

1. Right click on the “Source Files” folder in the project tree. Select “Add Existing Item” to open the Select Item dialog.
2. Go to the directory:
`C:\Program Files\Microchip\xc16\vx.xx\support\templates\c`
where `vx.xx` is the version number.
3. Click the file `tmp6014.c`. Then shift-click the file `traps.c`. Click **Select**.

When you are done, the project tree should now have the Source Files folder open, containing to the two added files.

4.6.3 Set Build Options

Select *File>Project Properties* or right click on the project name and select “Properties” to open the Project Properties dialog.

1. Under “Conf:[default]>XC16 (Global Options)”, select “xc16-gcc”.
2. Select “Preprocessing and messages” from the “Option Categories”. Enter the following path for “C Include Dirs”:
`C:\Program Files\Microchip\xc16\vx.xx\support\dsPIC30F\h`
where `vx.xx` is the version number.
3. Under “Conf:[default]>XC16 (Global Options)”, select “xc16-ld”.
4. Select “Diagnostics” from the “Option Categories”. Then enter a file name to “Generate map file”, i.e., `example.map`.
5. Click **OK** on the bottom of the dialog to accept the build options and close the dialog.

4.6.4 Build the Project

Right click on the project name, “MyXC16Project”, in the project tree and select “Build” from the pop-up menu. The Output window displays the build results.

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

1. Review the previous steps in this example. Make sure you have set up the language tools correctly and have all the correct project files and build options.
2. If you modified the sample source code, examine the Build tab of the Output window for syntax errors in the source code. If you find any, click on the error to go to the source code line that contains that error. Correct the error, and then try to build again.

4.6.5 Output Files

View the project output files by opening the files in MPLAB X IDE.

1. Select *File>Open File*. In the Open dialog, find the project directory.
2. Under “Files of type” select “All Files” to see all project files.
3. Select *File>Open File*. In the Open dialog, select “example.map”. Click **Open** to view the linker map file in an MPLAB X IDE editor window. For more on this file, see the linker documentation.
4. Select *File>Open File*. In the Open dialog, return to the project directory and then go to the *dist>default>production* directory. Notice that there is only one hex file, “MyXC16Project.X.production.hex”. This is the primary output file. Click **Open** to view the hex file in an MPLAB X IDE editor window. For more on this file, see the Utilities documentation.

There is also another file, “MyXC16Project.X.production.cof” or “MyXC16Project.X.production.elf”. This file contains debug information and is used by debug tools to debug your code. For information on selecting the type of debug file, see **Section 4.5.1 “XC16 (Global Options)”**.

4.6.6 Further Development

Usually, your application code will contain errors and not work the first time. Therefore, you will need a debug tool to help you develop your code. Using the output files previously discussed, several debug tools exist that work with MPLAB X IDE to help you do this. You may choose from simulators, in-circuit emulators or in-circuit debuggers, either manufactured by Microchip Technology or third-party developers. Please see the documentation for these tools to learn how they can help you. When debugging, you will use *Debug>Debug Project* to run and debug your code. Please see MPLAB X IDE documentation for more information.

Once you have developed your code, you will want to program it into a device. Again, there are several programmers that work with MPLAB X IDE to help you do this. Please see the documentation for these tools to see how they can help you. When programming, you will use “Make and Program Device Project” button on the debug toolbar. Please see MPLAB X IDE documentation concerning this control.

NOTES:

Chapter 5. Compiler Command-Line Driver

5.1 INTRODUCTION

The compiler command-line driver (`xc16-gcc`) is the application that invokes the operation of the MPLAB XC16 C Compiler. The driver compiles, assembles and links C and assembly language modules and library archives. Most of the compiler command-line options are common to all implementations of the GCC toolset. A few are specific to the compiler and will be discussed below.

The compiler driver also may be used with MPLAB X IDE or MPLAB IDE v8. Compiler options are selected in the GUI and passed to the compiler driver for execution.

Topics concerning the command-line use of the driver are discussed below.

- Invoking the Compiler
- The Compilation Sequence
- Runtime Files
- Compiler Output
- Compiler Messages
- Driver Option Descriptions
- MPLAB X IDE Toolchain or MPLAB IDE Toolsuite Equivalents

5.2 INVOKING THE COMPILER

The compiler is invoked and run on the command line as specified in the next section. Additionally, environmental variables and input files used by the compiler are discussed in the following sections.

5.2.1 Drive Command-Line Format

The basic form of the compiler command line is:

```
xc16-gcc [options] files
```

where:

options: See **Section 5.7 “Driver Option Descriptions”** for available options.

files: See **Section 5.2.3 “Input File Types”** for details.

Note: Command line options and file name extensions are case-sensitive.

It is assumed in this manual that the compiler applications are either in the console's search path, see **Section 5.2.2 “Environment Variables”**, or the full path is specified when executing any application.

It is conventional to supply *options* (identified by a leading *dash* “-”) before the file names, although this is not mandatory.

The *files* may be any mixture of C and assembler source files, and precompiled intermediate files, such as relocatable object (.o) files. The order of the files is not important, except that it may affect the order in which code or data appears in memory.

For example, to compile, assemble and link the C source file `hello.c`, creating a relocatable object output, `hello.elf`.

```
xc16-gcc -mcpu=30f2010 -o hello.elf hello.c
```

5.2.2 Environment Variables

The variables in this section are optional, but, if defined, they will be used by the compiler. The compiler driver, or other subprogram, may choose to determine an appropriate value for some of the following environment variables if they are unset. The driver, or other subprogram, takes advantage of internal knowledge about the installation of the compiler. As long as the installation structure remains intact, with all subdirectories and executables remaining in the same relative position, the driver or subprogram will be able to determine a usable value.

TABLE 5-1: COMPILER-RELATED ENVIRONMENTAL VARIABLES

Variable	Definition
XC16_C_INCLUDE_PATH PIC30_C_INCLUDE_PATH	This variable's value is a semicolon-separated list of directories, much like <code>PATH</code> . When the compiler searches for header files, it tries the directories listed in the variable, after the directories specified with <code>-I</code> but before the standard header file directories. If the environment variable is undefined, the preprocessor chooses an appropriate value based on the standard installation. By default, the following directories are searched for include files: <install-path>\include and <install-path>\support\h
XC16_COMPILER_PATH PIC30_COMPILER_PATH	The value of the variable is a semicolon-separated list of directories, much like <code>PATH</code> . The compiler tries the directories thus specified when searching for subprograms, if it can't find the subprograms using <code>PIC30_EXEC_PREFIX</code> .

TABLE 5-1: COMPILER-RELATED ENVIRONMENTAL VARIABLES

Variable	Definition
XC16_EXEC_PREFIX PIC30_EXEC_PREFIX	If the environment variable is set, it specifies a prefix to use in the names of subprograms executed by the compiler. No directory delimiter is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish. If the compiler cannot find the subprogram using the specified prefix, it tries looking in your <code>PATH</code> environment variable. If the environment variable is not set or set to an empty value, the compiler driver chooses an appropriate value based on the standard installation. If the installation has not been modified, this will result in the driver being able to locate the required subprograms. Other prefixes specified with the <code>-B</code> command line option take precedence over the user- or driver-defined value of the variable. Under normal circumstances it is best to leave this value undefined and let the driver locate subprograms itself.
XC16_LIBRARY_PATH PIC30_LIBRARY_PATH	This variable's value is a semicolon-separated list of directories, much like <code>PATH</code> . This variable specifies a list of directories to be passed to the linker. The driver's default evaluation of this variable is: <install-path>\lib; <install-path>\support\gld.
XC16_OMF PIC30_OMF	Specifies the OMF (Object Module Format) to be used by the compiler. By default, the tools create ELF object files. If the environment variable has the value <code>coff</code> , the tools will create COFF object files.
TMPDIR	If the variable is set, it specifies the directory to use for temporary files. The compiler uses temporary files to hold the output of one stage of compilation that is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

5.2.3 Input File Types

The compilation driver distinguishes source files, intermediate files and library files solely by the file type, or extension. It recognizes the following file extensions, which are case-sensitive.

TABLE 5-2: FILE NAMES

Extensions	Definition
<i>file.c</i>	A C source file that must be preprocessed.
<i>file.h</i>	A header file (not to be compiled or linked).
<i>file.i</i>	A C source file that should not be preprocessed.
<i>file.o</i>	An object file.
<i>file.p</i>	A pre procedural-abstraction assembly language file.
<i>file.s</i>	Assembler code.
<i>file.S</i>	Assembler code that must be preprocessed.
other	A file to be passed to the linker.

There are no compiler restrictions imposed on the names of source files, but be aware of case, name-length and other restrictions imposed by your operating system. If you are using an IDE, avoid assembly source files whose basename is the same as the basename of any project in which the file is used. This may result in the source file being overwritten by a temporary file during the build process.

The terms “source file” and “module” are often used when talking about computer programs. They are often used interchangeably, but they refer to the source code at different points in the compilation sequence.

A source file is a file that contains all or part of a program. Source files are initially passed to the preprocessor by the driver.

A module is the output of the preprocessor, for a given source file, after inclusion of any header files (or other source files) which are specified by `#include` preprocessor directives. These modules are then passed to the remainder of the compiler applications. Thus, a module may consist of several source and header files. A module is also often referred to as a translation unit. These terms can also be applied to assembly files, as they too can include other header and source files.

5.3 THE COMPILATION SEQUENCE

How the compiler operates with other applications and how to perform different types of compilations is discussed in the following sections.

5.3.1 The Compiler Applications

The MPLAB XC16 C Compiler compiles C source files, producing assembly language files. These compiler-generated files are assembled and linked with other object files and libraries to produce the final application program in executable ELF or COFF file format. The ELF or COFF file can be loaded into the MPLAB X IDE or MPLAB IDE v8, where it can be tested and debugged, or the conversion utility can be used to convert the ELF or COFF file to Intel® hex format, suitable for loading into the command-line simulator or a device programmer. A software development tools data flow diagram is shown in **Section 4.4 “MPLAB X IDE Projects”**.

The driver program will call the required internal compiler applications. These applications are shown as the smaller boxes inside the large driver box. The temporary file produced by each application can also be seen in this diagram.

Table 5-3 lists the compiler applications. The names shown are the names of the executables, which can be found in the `bin` directory under the compiler's installation directory. Your `PATH` environment variable should include this directory.

TABLE 5-3: COMPILER APPLICATION NAMES

Name	Description
<code>xc16-gcc</code>	Command line driver; the interface to the compiler
<code>xc16-as</code>	Assembler (based on the target device)
<code>xc16-ld</code>	Linker
<code>xc16-bin2hex</code>	Conversion utility to create HEX files
<code>xc16-strings</code>	String extractor utility
<code>xc16-strip</code>	Symbol stripper utility
<code>xc16-nm</code>	Symbol list utility
<code>xc16-ar</code>	Archiver/Librarian
<code>xc16-objdump</code>	Object file display utility
<code>xc16-ranlib</code>	Archive indexer utility

5.3.2 Single-Step Compilation

A single command-line instruction can be used to compile one file or multiple files.

5.3.2.1 COMPILING A SINGLE FILE

This section demonstrates how to compile and link a single file. For the purpose of this discussion, it is assumed the compiler is installed in the standard directory location and that your PATH or other environment variables (see **Section 5.2.2 “Environment Variables”**) are set up in such a way that the full compiler path need not be specified when you run the compiler.

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

Create the following program with any text editor and save it as `ex1.c`.

```
#include <xc.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int
main(void)
{
    x = 2;
    y = 5;
    z = Add(x,y);
    return 0;
}
unsigned int
Add(unsigned int a, unsigned int b)
{
    return(a+b);
}
```

The first line of the program includes the header file `xc.h`, which will include the appropriate header files that provides definitions for all special function registers on the target device. For more information on header files, see **Section 6.3 “Device Header Files”**.

Compile the program by typing the following at the prompt in your favorite terminal.

```
xc16-gcc -mcpu=30f2010 -o ex1.elf ex1.c
```

The command-line option `-o ex1.elf` names the output executable file (if the `-o` option is not specified, then the output file is named `a.out`). The executable file may be loaded into the MPLAB X IDE or MPLAB IDE v8.

If a hex file is required, for example, to load into a device programmer, then use the following command:

```
xc16-bin2hex ex1.elf
```

This creates an Intel hex file named `ex1.hex`.

5.3.2.2 COMPILING MULTIPLE FILES

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

File 1

```
/* ex1.c */
#include <xc.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
    x = 2;
    y = 5;
    z = Add(x,y);
    return 0;
}
```

File 2

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

Compile both files in the one command by typing the following in your terminal program.

```
xc16-gcc -mcpu=30f2010 -o ex1.elf ex1.c add.c
```

This command compiles the modules `ex1.c` and `add.c`. The compiled modules are linked with the compiler libraries and the executable file `ex1.elf` is created.

5.3.3 Multi-Step Compilation

Make utilities and integrated development environments, such as MPLAB IDE, allow for an incremental build of projects that contain multiple source files. When building a project, they take note of which source files have changed since the last build and use this information to speed up compilation.

For example, if compiling two source files, but only one has changed since the last build, the intermediate file corresponding to the unchanged source file need not be regenerated.

If the compiler is being invoked using a make utility, the make file will need to be configured to recognize the different intermediate file format and the options used to generate the intermediate files. Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file, and once to perform the second stage compilation.

You may also wish to generate intermediate files to construct your own library files, although MPLAB XC16 is capable of constructing libraries so this is typically not necessary. See *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for more information on library creation.

For example, the files `ex1.c` and `add.c` are to be compiled using a make utility. The command lines that the make utility should use to compile these files might be something like:

```
xc16-gcc -mcpu=30f6014 -c ex1.c
xc16-gcc -mcpu=30f6014 -c add.c
xc16-gcc -mcpu=30f6014 -o ex1 ex1.o add.o
```

The `-c` option will compile the named file into the intermediate (object) file format, but not link. Once all files are compiled as specified by the make, then the resultant object files are linked in the final step to create the final output `ex1`. The above example uses the command-line driver, `xc16-gcc`, to perform the final link step. You can explicitly call the linker application, `xc16-ld`, but this is not recommended. When driving the linker application, you must specify linker options, not driver options. For more on using the linker, see *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.

When compiling debug code, the object module format (OMF) must be consistent for compilation, assembly and linking. The ELF/DWARF format is used by default but the COFF format may also be selected using `-omf=coff` or the environmental variable `XC16_OMF`.

5.3.4 Assembly Compilation

A mix of C and assembly code can be compiled together using the compiler (Figure). For more details, see **Chapter 16. "Mixing C and Assembly Code"**.

Additionally, the compiler may be used to generate assembly code (`.s`) from C code (`.c`) using the `-S` option. The assembly output may then be used in subsequent compilation using the command-line driver.

5.4 RUNTIME FILES

The compiler uses the following files in addition to source, linker and header files.

5.4.1 Library Files

The compiler may include library files into the output per Figure 4-2.

By default, `xc16-gcc` will search known locations under the compiler installation directory for library files that are required during compilation.

5.4.1.1 STANDARD LIBRARIES

The C standard libraries contain a standardized collection of functions, such as string, math and input/output routines. The range of these functions are described in the “16-Bit Language Tool Libraries” (DS51456).

5.4.1.2 USER-DEFINED LIBRARIES

Users may create their own libraries. Libraries are useful for bundling and precompiling selected functions so that application file management is easier and application compilation times are shorter.

Libraries can be created manually using the compiler and the librarian. To create files that may then be used as input to the 16-bit librarian (`xc16-ar`), use the `-c` compiler option to stop compilation before the linker stage. For information on using the librarian, see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide* (DS52106).

Libraries should be called `liblibrary.a` and can be added to the compiler command line by specifying its pathname (`-Ldir`) and `-llibrary`. For details on these options, see **Section 5.7.9 “Options for Linking”**.

A simple example of adding the library `libmyfns.a` to the command-line is:

```
xc16-gcc -mcpu=30f2010 -lmyfns example.c
```

As with Standard C library functions, any functions contained in user-defined libraries should have a declaration added to a header file. It is common practice to create one or more header files that are packaged with the library file. These header files can then be included into source code when required.

Library files specified on the command line are scanned first for unresolved symbols, so these files may redefine anything that is defined in the C standard libraries.

5.4.2 Startup and Initialization

Two kinds of startup modules are available to initialize the C runtime environment:

- The primary startup module which is linked by default (or the `-Wl, --data-init` option.)
- The alternate startup module which is linked when the `-Wl, --no-data-init` option is specified (no data initialization.)

These modules are included in the `libpic30-omf.a` archive/library. Multiple versions of these modules exist in order to support architectural differences between device families. The compiler automatically uses the correct module.

For more information on the startup modules, see **Section 15.3 “Runtime Startup and Initialization”**.

5.5 COMPILER OUTPUT

There are many files created by the compiler during the compilation. A large number of these are intermediate files are deleted after compilation is complete, but many remain and are used for programming the device or for debugging purposes.

5.5.1 Output Files

The compilation driver can produce output files with the following extensions, which are case-sensitive.

TABLE 5-4: FILE NAMES

Extensions	Definition
<i>file.hex</i>	Executable file
<i>file.cof</i>	COF debug file (default)
<i>file.elf</i>	ELF debug file
<i>file.o</i>	Object file (intermediate file)
<i>file.S</i>	Assembly code file (required preprocessing)
<i>file.s</i>	Assembly code file (intermediate file)
<i>file.i</i>	Preprocessed file (intermediate file)
<i>file.p</i>	Preprocedure abstraction assembly language file (intermediate file)
<i>file.map</i>	Map file

The names of many output files use the same base name as the source file from which they were derived. For example the source file `input.c` will create an object file called `input.o` when the `-c` option is used.

The default output file is a ELF file called `a.out`, unless you override that name using the `-o` option.

If you are using MPLAB X IDE or MPLAB IDE v8 to specify options to the compiler, there is typically a project file that is created for each application. The name of this project is used as the base name for project-wide output files, unless otherwise specified by the user. However check the manual for the IDE you are using for more details.

Note: Throughout this manual, the term *project name* will refer to the name of the project created in the IDE.

The compiler is able to directly produce a number of the output file formats which are used by Microchip development tools.

The default behavior of `xc16-gcc` is to produce a ELF output. To make changes to the files output or the file names, see **Section 5.7 “Driver Option Descriptions”**.

5.5.2 Diagnostic Files

Two valuable files produced by the compiler are the assembly list file, produced by the assembler, and the map file, produced by the linker.

The assembly list file contains the mapping between the original source code and the generated assembly code. It is useful for information such as how C source was encoded, or how assembly source may have been optimized. It is essential when confirming if compiler-produced code that accesses objects is atomic, and shows the region in which all objects and code are placed.

The option to create a listing file in the assembler is `-a`. There are many variants to this option, which may be found in the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*. To pass the option from the compiler, see **Section 5.7.8 “Options for Assembling”**.

There is one list file produced for each build. Thus, if you require a list file for each source file, these files must be compiled separately, see **Section 5.3.3 “Multi-Step Compilation”**. This is the case if you build using MPLAB IDE. Each list file will be assigned the module name and extension `.lst`.

The map file shows information relating to where objects were positioned in memory. It is useful for confirming if user-defined linker options were correctly processed, and for determining the exact placement of objects and functions.

The linker option to create a map file in the linker application is `-Map file`, which may be found in the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*. To specify the option from the command-line driver, see **Section 5.7.9 “Options for Linking”**.

There is one map file produced when you build a project, assuming the linker was executed and ran to completion.

5.6 COMPILER MESSAGES

Compiler output messages for errors, warnings or comments as discussed in **Appendix C. “Diagnostics”**.

For information on options that control compiler output of errors, warnings or comments, see **Section 5.7.4 “Options for Controlling Warnings and Errors”**.

There are no pragmas that directly control messages issued by the compiler.

5.7 DRIVER OPTION DESCRIPTIONS

The compiler has many options for controlling compilation, all of which are case-sensitive. They have been grouped, as shown below, according to their function. Remember, these are options for the command-line driver; refer to **Section 5.7.8 “Options for Assembling”** or **Section 5.7.9 “Options for Linking”** for information on specifying options for these tools to the compiler.

- Options Specific to 16-Bit Devices
- Options for Controlling the Kind of Output
- Options for Controlling the C Dialect
- Options for Controlling Warnings and Errors
- Options for Debugging
- Options for Controlling Optimization
- Options for Controlling the Preprocessor
- Options for Assembling
- Options for Linking
- Options for Directory Search
- Options for Code Generation Conventions

5.7.1 Options Specific to 16-Bit Devices

For more information on the memory models, see **Section 10.12 “Memory Models”**.

TABLE 5-5: 16-BIT DEVICE-SPECIFIC OPTIONS

Option	Definition
<code>-mconst-in-code</code>	Put <code>const</code> qualified variables in the <code>auto_psv</code> space. The compiler will access these variables using the PSV window. (This is the default.)
<code>-mconst-in-data</code>	Put <code>const</code> qualified variables in the data memory space.
<code>-mconst-in-auxflash</code>	When combined with <code>-mconst-in-code</code> , put <code>const</code> qualified file scope variables into auxiliary FLASH. All modules with auxiliary FLASH should be compiled with this option; otherwise a link error may occur.
<code>-merrata=id[,id]*</code>	This option enables specific errata work arounds identified by <code>id</code> . Valid values for <code>id</code> change from time to time and may not be required for a particular variant. An <code>id</code> of <code>list</code> will display the currently supported errata identifiers along with a brief description of the errata. An <code>id</code> of <code>all</code> will enable all currently supported errata work arounds.
<code>-mfillupper</code>	Specify the upper byte of variables stored into <code>space(prog)</code> sections. The <code>fillupper</code> attribute will perform the same function on individual variables.

Note 1: The procedure abstractor behaves as the inverse of inlining functions. The pass is designed to extract common code sequences from multiple sites throughout a translation unit and place them into a common area of code. Although this option generally does not improve the run-time performance of the generated code, it can reduce the code size significantly. Programs compiled with `-mpa` can be harder to debug; it is not recommended that this option be used while debugging using the COFF object format.

The procedure abstractor is invoked as a separate phase of compilation, after the production of an assembly file. This phase does not optimize across translation units. When the procedure-optimizing phase is enabled, inline assembly code must be limited to valid machine instructions. Invalid machine instructions or instruction sequences, or assembler directives (sectioning directives, macros, include files, etc.) must not be used, or the procedure abstraction phase will fail, inhibiting the creation of an output file.

TABLE 5-5: 16-BIT DEVICE-SPECIFIC OPTIONS (CONTINUED)

Option	Definition
<code>-mlarge-arrays</code>	Specifies that arrays may be larger than the default maximum size of 32K. See Section 6.7 “Bit-Reversed and Modulo Addressing” for more information.
<code>-mlarge-code</code>	Compile using the large code model. No assumptions are made about the locality of called functions. When this option is chosen, single functions that are larger than 32k are not supported and may cause assembly-time errors since all branches inside of a function are of the short form.
<code>-mlarge-data</code>	Compile using the large data model. No assumptions are made about the location of static and external variables.
<code>-mcpu= target</code>	This option selects the target processor ID (and communicates it to the assembler and linker if those tools are invoked). This option affects how some predefined constants are set; see Section 19.4 “Predefined Macro Names” for more information. A full list of accepted targets can be seen in the <code>Readme.htm</code> file that came with the release.
<code>-mpa⁽¹⁾</code>	Enable the procedure abstraction optimization. There is no limit on the nesting level. Optimization levels depend on the compiler edition (see Chapter 18. “Optimizations” .)
<code>-mpa=<i>n</i>⁽¹⁾</code>	Enable the procedure abstraction optimization up to level <i>n</i> . If <i>n</i> is zero, the optimization is disabled. If <i>n</i> is 1, first level of abstraction is allowed; that is, instruction sequences in the source code may be abstracted into a subroutine. If <i>n</i> is 2, a second level of abstraction is allowed; that is, instructions that were put into a subroutine in the first level may be abstracted into a subroutine one level deeper. This pattern continues for larger values of <i>n</i> . The net effect is to limit the subroutine call nesting depth to a maximum of <i>n</i> . Optimization levels depend on the compiler edition (see Chapter 18. “Optimizations” .)
<code>-mno-pa⁽¹⁾</code>	Do not enable the procedure abstraction optimization. (This is the default.)
<code>-mno-isr-warn</code>	By default the compiler will produce a warning if the <code>__interrupt__</code> is not attached to a recognized interrupt vector name. This option will disable that feature.
<code>-omf</code>	Selects the OMF (Object Module Format) to be used by the compiler. The <i>omf</i> specifier can be one of the following: <code>elf</code> Produce ELF object files. (This is the default.) <code>coff</code> Produce COFF object files. The debugging format used for ELF object files is DWARF 2.0.
<code>-msmall-code</code>	Compile using the small code model. Called functions are assumed to be proximate (within 32 Kwords of the caller). (This is the default.)

Note 1: The procedure abstractor behaves as the inverse of inlining functions. The pass is designed to extract common code sequences from multiple sites throughout a translation unit and place them into a common area of code. Although this option generally does not improve the run-time performance of the generated code, it can reduce the code size significantly. Programs compiled with `-mpa` can be harder to debug; it is not recommended that this option be used while debugging using the COFF object format.

The procedure abstractor is invoked as a separate phase of compilation, after the production of an assembly file. This phase does not optimize across translation units. When the procedure-optimizing phase is enabled, inline assembly code must be limited to valid machine instructions. Invalid machine instructions or instruction sequences, or assembler directives (sectioning directives, macros, include files, etc.) must not be used, or the procedure abstraction phase will fail, inhibiting the creation of an output file.

TABLE 5-5: 16-BIT DEVICE-SPECIFIC OPTIONS (CONTINUED)

Option	Definition
<code>-msmall-data</code>	Compile using the small data model. All static and external variables are assumed to be located in the lower 8 KB of data memory space. (This is the default.)
<code>-msmall-scalar</code>	Like <code>-msmall-data</code> , except that only static and external scalars are assumed to be in the lower 8 KB of data memory space. (This is the default.)
<code>-mtext=name</code>	Specifying <code>-mtext=name</code> will cause text (program code) to be placed in a section named <i>name</i> rather than the default <code>.text</code> section. No white spaces should appear around the <code>=</code> .
<code>-mauxflash</code>	Place all code from the current translation unit into auxiliary FLASH. This option is only available on devices that have auxiliary FLASH.
<code>-msmart-io</code> <code>[=0 1 2]</code>	This option attempts to statically analyze format strings passed to <code>printf</code> , <code>scanf</code> and the 'f' and 'v' variations of these functions. Uses of nonfloating point format arguments will be converted to use an integer-only variation of the library functions. <code>-msmart-io=0</code> disables this option, while <code>-msmart-io=2</code> causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. <code>-msmart-io=1</code> is the default and will only convert the literal values it can prove.

Note 1: The procedure abstractor behaves as the inverse of inlining functions. The pass is designed to extract common code sequences from multiple sites throughout a translation unit and place them into a common area of code. Although this option generally does not improve the run-time performance of the generated code, it can reduce the code size significantly. Programs compiled with `-mpa` can be harder to debug; it is not recommended that this option be used while debugging using the COFF object format.

The procedure abstractor is invoked as a separate phase of compilation, after the production of an assembly file. This phase does not optimize across translation units. When the procedure-optimizing phase is enabled, inline assembly code must be limited to valid machine instructions. Invalid machine instructions or instruction sequences, or assembler directives (sectioning directives, macros, include files, etc.) must not be used, or the procedure abstraction phase will fail, inhibiting the creation of an output file.

5.7.2 Options for Controlling the Kind of Output

The following options control the kind of output produced by the compiler.

TABLE 5-6: KIND-OF-OUTPUT CONTROL OPTIONS

Option	Definition
<code>-c</code>	Compile or assemble the source files, but do not link. The default file extension is <code>.o</code> .
<code>-E</code>	Stop after the preprocessing stage, i.e., before running the compiler proper. The default output file is <code>stdout</code> .
<code>-o file</code>	Place the output in <i>file</i> .
<code>-S</code>	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is <code>.s</code> .
<code>-v</code>	Print the commands executed during each stage of compilation.

TABLE 5-6: KIND-OF-OUTPUT CONTROL OPTIONS (CONTINUED)

Option	Definition
-x	<p>You can specify the input language explicitly with the <code>-x</code> option:</p> <p>-x <i>language</i> Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next <code>-x</code> option. The following values are supported by the compiler:</p> <pre>c c-header cpp-output assembler assembler-with-cpp</pre> <p>-x none Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another <code>-x</code> option has been used.</p> <p>For example:</p> <pre>xc16-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s</pre> <p>Without the <code>-x none</code>, the compiler will assume all the input files are for the assembler.</p>
--help	Print a description of the command line options.

5.7.3 Options for Controlling the C Dialect

The following options define the kind of C dialect used by the compiler.

TABLE 5-7: C DIALECT CONTROL OPTIONS

Option	Definition
-ansi	Support all (and only) ANSI-standard C programs.
-aux-info filename	Output to the given file name prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C. Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (<code>I</code> , <code>N</code> for new or <code>O</code> for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (<code>C</code> or <code>F</code> , respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.
-menable-fixed [= <i>rounding mode</i>]	Enable fixed-point variable types and arithmetic operation support. Optionally, set the default <i>rounding mode</i> to one of truncation , conventional , or convergent . If the <i>rounding mode</i> is not specified, the default is truncation .
-ffreestanding	Assert that compilation takes place in a freestanding environment. This implies <code>-fno-builtin</code> . A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to <code>-fno-hosted</code> .
-fno-asm	Do not recognize <code>asm</code> , <code>inline</code> or <code>typeof</code> as a keyword, so that code can use these words as identifiers. You can use the keywords <code>__asm__</code> , <code>__inline__</code> and <code>__typeof__</code> instead. <code>-ansi</code> implies <code>-fno-asm</code> .

TABLE 5-7: C DIALECT CONTROL OPTIONS (CONTINUED)

Option	Definition
-fno-builtin -fno-builtin- <i>function</i>	Don't recognize built-in functions that do not begin with <code>__builtin_</code> as prefix.
-fsigned-char	Let the type <code>char</code> be signed, like <code>signed char</code> . (This is the default.)
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	These options control whether a bit-field is signed or unsigned, when the declaration does not use either <code>signed</code> or <code>unsigned</code> . By default, such a bit-field is signed, unless <code>-traditional</code> is used, in which case bit-fields are always unsigned.
-funsigned-char	Let the type <code>char</code> be unsigned, like <code>unsigned char</code> .

5.7.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W`, for example, `-Wimplicit`, to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-` to turn off warnings, for example, `-Wno-implicit`. This manual lists only one of the two forms, whichever is not the default.

The following options control the amount and kinds of warnings produced by the compiler.

TABLE 5-8: WARNING/ERROR OPTIONS IMPLIED BY `-Wall`

Option	Definition
-fsyntax-only	Check the code for syntax, but don't do anything beyond that.
-pedantic	Issue all the warnings demanded by strict ANSI C; reject all programs that use forbidden extensions.
-pedantic-errors	Like <code>-pedantic</code> , except that errors are produced rather than warnings.
-w	Inhibit all warning messages.
-Wall	All of the <code>-w</code> options listed in this table combined. This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
-Wchar-subscripts	Warn if an array subscript has type <code>char</code> .
-Wcomment -Wcomments	Warn whenever a comment-start sequence <code>/*</code> appears in a <code>/*</code> comment, or whenever a Backslash-Newline appears in a <code>//</code> comment.
-Wdiv-by-zero	Warn about compile-time integer division by zero. To inhibit the warning messages, use <code>-Wno-div-by-zero</code> . Floating point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs. (This is the default.)
-Werror-implicit- function-declaration	Give an error whenever a function is used before being declared.
-Wformat	Check calls to <code>printf</code> and <code>scanf</code> , etc., to make sure that the arguments supplied have types appropriate to the format string specified.
-Wimplicit	Equivalent to specifying both <code>-Wimplicit-int</code> and <code>-Wimplicit-function-declaration</code> .

TABLE 5-8: WARNING/ERROR OPTIONS IMPLIED BY -WALL (CONTINUED)

Option	Definition
-Wimplicit-function-declaration	Give a warning whenever a function is used before being declared.
-Wimplicit-int	Warn when a declaration does not specify a type.
-Wmain	Warn if the type of <code>main</code> is suspicious. <code>main</code> should be a function with external linkage, returning <code>int</code> , taking either zero, two or three arguments of appropriate types.
-Wmissing-braces	Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for <code>a</code> is not fully bracketed, but that for <code>b</code> is fully bracketed. <pre>int a[2][2] = { 0, 1, 2, 3 }; int b[2][2] = { { 0, 1 }, { 2, 3 } };</pre>
-Wmultichar -Wno-multichar	Warn if a multi-character <i>character</i> constant is used. Usually, such constants are typographical errors. Since they have implementation-defined values, they should not be used in portable code. The following example illustrates the use of a multi-character <i>character</i> constant: <pre>char xx(void) { return('xx'); }</pre>
-Wparentheses	Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often find confusing.
-Wreturn-type	Warn whenever a function is defined with a return-type that defaults to <code>int</code> . Also warn about any <code>return</code> statement with no return-value in a function whose return-type is not <code>void</code> .

TABLE 5-8: WARNING/ERROR OPTIONS IMPLIED BY `-Wall` (CONTINUED)

Option	Definition
<code>-Wsequence-point</code>	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a <code>&&</code> , <code> </code> , <code>?</code> or <code>,</code> (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap. It is not specified, when, between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that "Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored." If a program breaks these rules, the results on any particular implementation are entirely unpredictable. Examples of code with undefined behavior are <code>a = a++;</code> , <code>a[n] = b[n++]</code> and <code>a[i++] = i;</code> . Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.
<code>-Wswitch</code>	Warn whenever a <code>switch</code> statement has an index of enumerational type and lacks a <code>case</code> for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) <code>case</code> labels outside the enumeration range also provoke warnings when this option is used.
<code>-Wsystem-headers</code>	Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells the compiler to emit warnings from system headers as if they occurred in user code. However, note that using <code>-Wall</code> in conjunction with this option will not warn about unknown pragmas in system headers; for that, <code>-Wunknown-pragmas</code> must also be used.
<code>-Wtrigraphs</code>	Warn if any trigraphs are encountered (assuming they are enabled).

TABLE 5-8: WARNING/ERROR OPTIONS IMPLIED BY `-Wall` (CONTINUED)

Option	Definition
<code>-Wuninitialized</code>	Warn if an automatic variable is used without first being initialized. These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing. These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared <code>volatile</code> , or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.
<code>-Wunknown-pragmas</code>	Warn when a <code>#pragma</code> directive is encountered which is not understood by the compiler. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the <code>-Wall</code> command line option.
<code>-Wunused</code>	Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used. In order to get a warning about an unused function parameter, both <code>-W</code> and <code>-Wunused</code> must be specified. Casting an expression to void suppresses this warning for an expression. Similarly, the <code>unused</code> attribute suppresses this warning for unused variables, parameters and labels.
<code>-Wunused-function</code>	Warn whenever a static function is declared but not defined or a non-inline static function is unused.
<code>-Wunused-label</code>	Warn whenever a label is declared but not used. To suppress this warning, use the <code>unused</code> attribute (see Section 8.12 “Variable Attributes”).
<code>-Wunused-parameter</code>	Warn whenever a function parameter is unused aside from its declaration. To suppress this warning, use the <code>unused</code> attribute (see Section 8.12 “Variable Attributes”).
<code>-Wunused-variable</code>	Warn whenever a local variable or non-constant static variable is unused aside from its declaration. To suppress this warning, use the <code>unused</code> attribute (see Section 8.12 “Variable Attributes”).
<code>-Wunused-value</code>	Warn whenever a statement computes a result that is explicitly not used. To suppress this warning, cast the expression to <code>void</code> .

The following `-W` options are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for. Others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

TABLE 5-9: WARNING/ERROR OPTIONS NOT IMPLIED BY -WALL

Option	Definition
-W	<p>Print extra warning messages for these events:</p> <ul style="list-style-type: none"> A nonvolatile automatic variable might be changed by a call to <code>longjmp</code>. These warnings are possible only in optimizing compilation. The compiler sees only the calls to <code>setjmp</code>. It cannot know where <code>longjmp</code> will be called; in fact, a signal handler could call it at any point in the code. As a result, a warning may be generated even when there is in fact no problem, because <code>longjmp</code> cannot in fact be called at the place that would cause a problem. A function could exit both via <code>return value</code>; and <code>return;</code>. Completing the function body without passing any return statement is treated as <code>return;</code>. An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to <code>void</code>. For example, an expression such as <code>x[i, j]</code> will cause a warning, but <code>x[(void) i, j]</code> will not. An unsigned value is compared against zero with <code><</code> or <code><=</code>. A comparison like <code>x<=y<=z</code> appears; this is equivalent to <code>(x<=y ? 1 : 0) <= z</code>, which is a different interpretation from that of ordinary mathematical notation. Storage-class specifiers like <code>static</code> are not the first things in a declaration. According to the C Standard, this usage is obsolescent. If <code>-Wall</code> or <code>-Wunused</code> is also specified, warn about unused arguments. A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (But don't warn if <code>-Wno-sign-compare</code> is also specified.) An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for <code>x.h</code>: <pre>struct s { int f, g; }; struct t { struct s h; int i; }; struct t x = { 1, 2, 3 };</pre> An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because <code>x.h</code> would be implicitly initialized to zero: <pre>struct s { int f, g, h; }; struct s x = { 3, 4 };</pre>
-Waggregate-return	Warn if any functions that return structures or unions are defined or called.
-Wbad-function-cast	Warn whenever a function call is cast to a non-matching type. For example, warn if <code>int foof()</code> is cast to anything <code>*</code> .
-Wcast-align	Warn whenever a pointer is cast, such that the required alignment of the target is increased. For example, warn if a <code>char *</code> is cast to an <code>int *</code> .
-Wcast-qual	Warn whenever a pointer is cast, so as to remove a type qualifier from the target type. For example, warn if a <code>const char *</code> is cast to an ordinary <code>char *</code> .

TABLE 5-9: WARNING/ERROR OPTIONS NOT IMPLIED BY `-Wall`

Option	Definition
<code>-Wconversion</code>	Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument, except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment <code>x = -1</code> if <code>x</code> is unsigned. But do not warn about explicit casts like <code>(unsigned) -1</code> .
<code>-Werror</code>	Make all warnings into errors.
<code>-Winline</code>	Warn if a function can not be inlined, and either it was declared as inline, or else the <code>-finline-functions</code> option was given.
<code>-Wlarger-than-len</code>	Warn whenever an object of larger than <code>len</code> bytes is defined.
<code>-Wlong-long</code> <code>-Wno-long-long</code>	Warn if <code>long long</code> type is used. This is default. To inhibit the warning messages, use <code>-Wno-long-long</code> . Flags <code>-Wlong-long</code> and <code>-Wno-long-long</code> are taken into account only when <code>-pedantic</code> flag is used.
<code>-Wmissing-declarations</code>	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.
<code>-Wmissing-format-attribute</code>	If <code>-Wformat</code> is enabled, also warn about functions that might be candidates for format attributes. Note these are only possible candidates, not absolute ones. This option has no effect unless <code>-Wformat</code> is enabled.
<code>-Wmissing-noreturn</code>	Warn about functions that might be candidates for attribute <code>noreturn</code> . These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. Actually, do not ever return before adding the <code>noreturn</code> attribute; otherwise subtle code generation bugs could be introduced.
<code>-Wmissing-prototypes</code>	Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. (This option can be used to detect global functions that are not declared in header files.)
<code>-Wnested-externs</code>	Warn if an <code>extern</code> declaration is encountered within a function.
<code>-Wno-deprecated-declarations</code>	Do not warn about uses of functions, variables and types marked as deprecated by using the <code>deprecated</code> attribute.
<code>-Wpadded</code>	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.
<code>-Wpointer-arith</code>	Warn about anything that depends on the size of a function type or of <code>void</code> . The compiler assigns these types a size of 1, for convenience in calculations with <code>void *</code> pointers and pointers to functions.
<code>-Wredundant-decls</code>	Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.
<code>-Wshadow</code>	Warn whenever a local variable shadows another local variable.

TABLE 5-9: WARNING/ERROR OPTIONS NOT IMPLIED BY `-Wall`

Option	Definition
<code>-Wsign-compare</code> <code>-Wno-sign-compare</code>	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by <code>-W</code> ; to get the other warnings of <code>-W</code> without this warning, use <code>-W -Wno-sign-compare</code> .
<code>-Wstrict-prototypes</code>	Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)
<code>-Wtraditional</code>	Warn about certain constructs that behave differently in traditional and ANSI C. <ul style="list-style-type: none"> Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C. A function declared external in one block and then used after the end of the block. A switch statement has an operand of type <code>long</code>. A nonstatic function declaration follows a static one. This construct is not accepted by some traditional C compilers.
<code>-Wundef</code>	Warn if an undefined identifier is evaluated in an <code>#if</code> directive.
<code>-Wwrite-strings</code>	Give string constants the type <code>const char[length]</code> so that copying the address of one into a non- <code>const char *</code> pointer will get a warning. These warnings will help you find at compile time code that you can try to write into a string constant, but only if you have been very careful about using <code>const</code> in declarations and prototypes. Otherwise, it will just be a nuisance, which is why <code>-Wall</code> does not request these warnings.

5.7.5 Options for Debugging

The following options are used for debugging.

TABLE 5-10: DEBUGGING OPTIONS

Option	Definition
<code>-g</code>	Produce debugging information. The compiler supports the use of <code>-g</code> with <code>-O</code> making it possible to debug optimized code. The shortcuts taken by optimized code may occasionally produce surprising results: <ul style="list-style-type: none"> Some declared variables may not exist at all; Flow of control may briefly move unexpectedly; Some statements may not be executed because they compute constant results or their values were already at hand; Some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.
<code>-Q</code>	Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.

TABLE 5-10: DEBUGGING OPTIONS (CONTINUED)

Option	Definition
-save-temps	Don't delete intermediate files. Place them in the current directory and name them based on the source file. Thus, compiling <code>foo.c</code> with <code>-c -save-temps</code> would produce the following files: <code>foo.i</code> (preprocessed file) <code>foo.p</code> (pre procedure abstraction assembly language file) <code>foo.s</code> (assembly language file) <code>foo.o</code> (object file)

5.7.6 Options for Controlling Optimization

The following options control compiler optimizations. Optimization levels available depend on the compiler edition (see **Chapter 18. "Optimizations"**.)

TABLE 5-11: GENERAL OPTIMIZATION OPTIONS

Option	Edition	Definition
-O0	All	Do not optimize. (This is the default.) Without <code>-O</code> , the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. The compiler only allocates variables declared <code>register</code> in registers.
-O -O1	All	Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. With <code>-O</code> , the compiler tries to reduce code size and execution time. When <code>-O</code> is specified, the compiler turns on <code>-fthread-jumps</code> and <code>-fdefer-pop</code> . The compiler turns on <code>-fomit-frame-pointer</code> .
-O2	STD, PRO	Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. <code>-O2</code> turns on all optional optimizations except for loop unrolling (<code>-funroll-loops</code>), function inlining (<code>-finline-functions</code>), and strict aliasing optimizations (<code>-fstrict-aliasing</code>). It also turns on Frame Pointer elimination (<code>-fomit-frame-pointer</code>). As compared to <code>-O</code> , this option increases both compilation time and the performance of the generated code.
-O3	PRO	Optimize for speed. <code>-O3</code> turns on all optimizations specified by <code>-O2</code> and also turns on the <code>inline-functions</code> option.
-Os	PRO	Optimize for size. <code>-Os</code> enables all <code>-O2</code> optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

The following options control specific optimizations. The `-O2` option turns on all of these optimizations except `-funroll-loops`, `-funroll-all-loops` and `-fstrict-aliasing`.

You can use the following flags in the rare cases when “fine-tuning” of optimizations to be performed is desired.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS

Option	Definition
<code>-falign-functions</code> <code>-falign-functions=<i>n</i></code>	Align the start of functions to the next power-of-two greater than <i>n</i> , skipping up to <i>n</i> bytes. For instance, <code>-falign-functions=32</code> aligns functions to the next 32-byte boundary, but <code>-falign-functions=24</code> would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less. <code>-fno-align-functions</code> and <code>-falign-functions=1</code> are equivalent and mean that functions will not be aligned. The assembler only supports this flag when <i>n</i> is a power of two; so <i>n</i> is rounded up. If <i>n</i> is not specified, use a machine-dependent default.
<code>-falign-labels</code> <code>-falign-labels=<i>n</i></code>	Align all branch targets to a power-of-two boundary, skipping up to <i>n</i> bytes like <code>-falign-functions</code> . This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code. If <code>-falign-loops</code> or <code>-falign-jumps</code> are applicable and are greater than this value, then their values are used instead. If <i>n</i> is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.
<code>-falign-loops</code> <code>-falign-loops=<i>n</i></code>	Align loops to a power-of-two boundary, skipping up to <i>n</i> bytes like <code>-falign-functions</code> . The hope is that the loop will be executed many times, which will make up for any execution of the dummy operations. If <i>n</i> is not specified, use a machine-dependent default.
<code>-fcaller-saves</code>	Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.
<code>-fcse-follow-jumps</code>	In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an <code>if</code> statement with an <code>else</code> clause, CSE will follow the jump when the condition tested is false.
<code>-fcse-skip-blocks</code>	This is similar to <code>-fcse-follow-jumps</code> , but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple <code>if</code> statement with no <code>else</code> clause, <code>-fcse-skip-blocks</code> causes CSE to follow the jump around the body of the <code>if</code> .
<code>-fexpensive-optimizations</code>	Perform a number of minor optimizations that are relatively expensive.
<code>-ffunction-sections</code> <code>-fdata-sections</code>	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file. Only use these options when there are significant benefits for doing so. When you specify these options, the assembler and linker may create larger object and executable files and will also be slower.
<code>-fgcse</code>	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
-fgcse-lm	When -fgcse-lm is enabled, global common subexpression elimination will attempt to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.
-fgcse-sm	When -fgcse-sm is enabled, a store motion pass is run after global common subexpression elimination. This pass will attempt to move stores out of loops. When used in conjunction with -fgcse-lm, loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.
-fno-defer-pop	Always pop the arguments to each function call as soon as that function returns. The compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.
-fno-peephole -fno-peephole2	Disable machine specific peephole optimizations. Peephole optimizations occur at various points during the compilation. -fno-peephole disables peephole optimization on machine instructions, while -fno-peephole2 disables high level peephole optimizations. To disable peephole entirely, use both options.
-foptimize-register-move -fregmove	Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. -fregmove and -foptimize-register-moves are the same optimization.
-frename-registers	Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization will most benefit processors with lots of registers. It can, however, make debugging impossible, since variables will no longer stay in a "home register".
-frerun-cse-after-loop	Rerun common subexpression elimination after loop optimizations has been performed.
-frerun-loop-opt	Run the loop optimizer twice.
-fschedule-insns	Attempt to reorder instructions to eliminate dsPIC® DSC Read-After-Write stalls (see the "dsPIC30F Family Reference Manual" (DS70046) for more details). Typically improves performance with no impact on code size.
-fschedule-insns2	Similar to -fschedule-insns, but requests an additional pass of instruction scheduling after register allocation has been done.
-fstrength-reduce	Perform the optimizations of loop strength reduction and elimination of iteration variables.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
<code>-fstrict-aliasing</code>	<p>Allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C, this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an <code>unsigned int</code> can alias an <code>int</code>, but not a <code>void*</code> or a <code>double</code>. A character type may alias any other type.</p> <p>Pay special attention to code like this:</p> <pre>union a_union { int i; double d; }; int f() { union a_union t; t.d = 3.0; return t.i; }</pre> <p>The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with <code>-fstrict-aliasing</code>, type-punning is allowed, provided the memory is accessed through the union type. So, the code above will work as expected. However, this code might not:</p> <pre>int f() { a_union t; int* ip; t.d = 3.0; ip = &t.i; return *ip; }</pre>
<code>-fthread-jumps</code>	<p>Perform optimizations where a check is made to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.</p>
<code>-funroll-loops</code>	<p>Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. <code>-funroll-loops</code> implies both <code>-fstrength-reduce</code> and <code>-frerun-cse-after-loop</code>.</p>
<code>-funroll-all-loops</code>	<p>Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. <code>-funroll-all-loops</code> implies <code>-fstrength-reduce</code>, as well as <code>-frerun-cse-after-loop</code>.</p>

Options of the form `-fflag` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. In the table below, only one of the forms is listed (the one that is not the default.)

TABLE 5-13: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
<code>-finline-functions</code>	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared <code>static</code> , then the function is normally not output as assembler code in its own right.
<code>-finline-limit=n</code>	By default, the compiler limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (i.e., marked with the <code>inline</code> keyword). <i>n</i> is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of <i>n</i> is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code will be inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining. Note: Pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way does it represent a count of assembly instructions and as such, its exact meaning might change from one release of the compiler to another.
<code>-fkeep-inline-functions</code>	Even if all calls to a given function are integrated, and the function is declared <code>static</code> , output a separate run time callable version of the function. This switch does not affect <code>extern inline</code> functions.
<code>-fkeep-static-consts</code>	Emit variables declared <code>static const</code> when optimization isn't turned on, even if the variables aren't referenced. The compiler enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the <code>-fno-keep-static-consts</code> option.
<code>-fno-function-cse</code>	Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly. This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.
<code>-fno-inline</code>	Do not pay attention to the <code>inline</code> keyword. Normally this option is used to keep the compiler from expanding any functions inline. If optimization is not enabled, no functions can be expanded inline.
<code>-fomit-frame-pointer</code>	Do not keep the Frame Pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore Frame Pointers; it also makes an extra register available in many functions.
<code>-foptimize-sibling-calls</code>	Optimize sibling and tail recursive calls.

5.7.7 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

TABLE 5-14: PREPROCESSOR OPTIONS

Option	Definition
<code>-Aquestion (answer)</code>	Assert the answer <i>answer</i> for question <i>question</i> , in case it is tested with a preprocessing conditional such as <code>#if question(answer)</code> . <code>-A-</code> disables the standard assertions that normally describe the target machine. For example, the function prototype for main might be declared as follows: <pre>#if #environ(freestanding) int main(void); #else int main(int argc, char *argv[]); #endif</pre> A <code>-A</code> command-line option could then be used to select between the two prototypes. For example, to select the first of the two, the following command-line option could be used: <code>-Aenviron(freestanding)</code>
<code>-A -predicate =answer</code>	Cancel an assertion with the predicate <i>predicate</i> and answer <i>answer</i> .
<code>-A predicate =answer</code>	Make an assertion with the predicate <i>predicate</i> and answer <i>answer</i> . This form is preferred to the older form <code>-A predicate(answer)</code> , which is still supported, because it does not use shell special characters.
<code>-C</code>	Tell the preprocessor not to discard comments. Used with the <code>-E</code> option.
<code>-dD</code>	Tell the preprocessor to not remove macro definitions into the output, in their proper sequence.
<code>-Dmacro</code>	Define macro <i>macro</i> with the string 1 as its definition.
<code>-Dmacro=defn</code>	Define macro <i>macro</i> as <i>defn</i> . All instances of <code>-D</code> on the command line are processed before any <code>-U</code> options.
<code>-dM</code>	Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the <code>-E</code> option.
<code>-dN</code>	Like <code>-dD</code> except that the macro arguments and contents are omitted. Only <code>#define name</code> is included in the output.
<code>-fno-show-column</code>	Do not print column numbers in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as <code>dejagnum</code> .
<code>-H</code>	Print the name of each header file used, in addition to other normal activities.

TABLE 5-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
<code>-I-</code>	Any directories you specify with <code>-I</code> options before the <code>-I-</code> options are searched only for the case of <code>#include "file"</code> ; they are not searched for <code>#include <file></code> . If additional directories are specified with <code>-I</code> options after the <code>-I-</code> , these directories are searched for all <code>#include</code> directives. (Ordinarily all <code>-I</code> directories are used this way.) In addition, the <code>-I-</code> option inhibits the use of the current directory (where the current input file came from) as the first search directory for <code>#include "file"</code> . There is no way to override this effect of <code>-I-</code> . With <code>-I.</code> you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. <code>-I-</code> does not inhibit the use of the standard system directories for header files. Thus, <code>-I-</code> and <code>-nostdinc</code> are independent.
<code>-I dir</code>	Add the directory <i>dir</i> to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one <code>-I</code> option, the directories are scanned in left-to-right order; the standard system directories come after.
<code>-idirafter dir</code>	Add the directory <i>dir</i> to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that <code>-I</code> adds to).
<code>-imacros file</code>	Process <i>file</i> as input, discarding the resulting output, before processing the regular input file. Because the output generated from the file is discarded, the only effect of <code>-imacros file</code> is to make the macros defined in <i>file</i> available for use in the main input. Any <code>-D</code> and <code>-U</code> options on the command line are always processed before <code>-imacros file</code> , regardless of the order in which they are written. All the <code>-include</code> and <code>-imacros</code> options are processed in the order in which they are written.
<code>-include file</code>	Process <i>file</i> as input before processing the regular input file. In effect, the contents of <i>file</i> are compiled first. Any <code>-D</code> and <code>-U</code> options on the command line are always processed before <code>-include file</code> , regardless of the order in which they are written. All the <code>-include</code> and <code>-imacros</code> options are processed in the order in which they are written.
<code>-iprefix prefix</code>	Specify <i>prefix</i> as the prefix for subsequent <code>-iwithprefix</code> options.
<code>-isystem dir</code>	Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.
<code>-iwithprefix dir</code>	Add a directory to the second include path. The directory's name is made by concatenating <i>prefix</i> and <i>dir</i> , where <i>prefix</i> was specified previously with <code>-iprefix</code> . If a prefix has not yet been specified, the directory containing the installed passes of the compiler is used as the default.
<code>-iwithprefixbefore dir</code>	Add a directory to the main include path. The directory's name is made by concatenating <i>prefix</i> and <i>dir</i> , as in the case of <code>-iwithprefix</code> .

TABLE 5-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-M	Tell the preprocessor to output a rule suitable for <code>make</code> describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the <code>#include</code> header files it uses. This rule may be a single line or may be continued with <code>\-newline</code> if it is long. The list of rules is printed on standard output instead of the preprocessed C program. -M implies -E (see Section 5.7.2 “Options for Controlling the Kind of Output”).
-MD	Like -M but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a <code>.d</code> extension.
-MF <i>file</i>	When used with -M or -MM, specifies a file in which to write the dependencies. If no -MF switch is given, the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options, -MD or -MMD, -MF, overrides the default dependency output file.
-MG	Treat missing header files as generated files and assume they live in the same directory as the source file. If -MG is specified, then either -M or -MM must also be specified. -MG is not supported with -MD or -MMD.
-MM	Like -M but the output mentions only the user header files included with <code>#include “file”</code> . System header files included with <code>#include <file></code> are omitted.
-MMD	Like -MD except mention only user header files, not system header files.
-MP	This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors <code>make</code> gives if you remove header files without updating the make-file to match. This is typical output: test.o: test.c test.h test.h:
-MQ	Same as -MT, but it quotes any characters which are special to <code>make</code> . -MQ '\$(objpfx)foo.o' gives <code>\$(objpfx)foo.o: foo.c</code> The default target is automatically quoted, as if it were given with -MQ.
-MT <i>target</i>	Change the target of the rule emitted by dependency generation. By default, CPP takes the name of the main input file, including any path, deletes any file suffix such as <code>.c</code> , and appends the platform's usual object suffix. The result is the target. An -MT option will set the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to -MT, or use multiple -MT options. For example: -MT '\$(objpfx)foo.o' might give <code>\$(objpfx)foo.o: foo.c</code>

TABLE 5-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
<code>-nostdinc</code>	Do not search the standard system directories for header files. Only the directories you have specified with <code>-I</code> options (and the current directory, if appropriate) are searched. (See Section 5.7.10 “Options for Directory Search”) for information on <code>-I</code> . By using both <code>-nostdinc</code> and <code>-I-</code> , the include-file search path can be limited to only those directories explicitly specified.
<code>-P</code>	Tell the preprocessor not to generate <code>#line</code> directives. Used with the <code>-E</code> option (see Section 5.7.2 “Options for Controlling the Kind of Output”).
<code>-trigraphs</code>	Support ANSI C trigraphs. The <code>-ansi</code> option also has this effect.
<code>-Umacro</code>	Undefine macro <i>macro</i> . <code>-U</code> options are evaluated after all <code>-D</code> options, but before any <code>-include</code> and <code>-imacros</code> options.
<code>-undef</code>	Do not predefine any nonstandard macros (including architecture flags).

5.7.8 Options for Assembling

The following options control assembler operations. For more on available options, see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.

TABLE 5-15: ASSEMBLY OPTIONS

Option	Definition
<code>-Wa,option</code>	Pass <i>option</i> as an option to the assembler. If <i>option</i> contains commas, it is split into multiple options at the commas. For example, to generate an assembly list file, use <code>-Wa,-a</code> .

5.7.9 Options for Linking

If any of the options `-c`, `-S` or `-E` are used, the linker is not run and object file names should not be used as arguments. For more on available options, see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.

TABLE 5-16: LINKING OPTIONS

Option	Definition
<code>--fill=options</code>	<p>Fill unused program memory. The format is:</p> <pre>--fill=[wn:]expression[@address[:end_address] unused]</pre> <p><code>address</code> and <code>end_address</code> will specify the range of program memory addresses to fill. If <code>end_address</code> is not provided then the <code>expression</code> will be written to the specific memory location at address <code>address</code>. The optional literal value <code>unused</code> may be specified to indicate that all unused memory will be filled. If none of the location parameters are provided, all unused memory will be filled. <code>expression</code> will describe how to fill the specified memory. The following options are available:</p> <p>A single value</p> <pre>xc16-ld --fill=0x12345678@unused</pre> <p>Range of values</p> <pre>xc16-ld --fill=1,2,3,4,097@0x9d000650:0x9d000750</pre> <p>An incrementing value</p> <pre>xc16-ld --fill=7+=911@unused</pre> <p>By default, the linker will fill using data that is instruction-word length. For 16-bit devices, the default fill width is 24 bits. However, you may specify the value width using <code>[wn:]</code>, where <code>n</code> is the fill value's width and <code>n</code> belongs to [1, 3].</p> <p>Multiple fill options may be specified on the command line; the linker will always process fill options at specific locations first.</p>
<code>--gc-sections</code>	<p>Remove dead functions from code at link time.</p> <p>Support is for ELF projects only. In order to make the best use of this feature, add the <code>-ffunction-sections</code> option to the compiler command line.</p>
<code>-Ldir</code>	<p>Add directory <code>dir</code> to the list of directories to be searched for libraries specified by the command-line option <code>-l</code>.</p>
<code>-legacy-libc</code>	<p>Use legacy include files and libraries (v3.24 and before).</p> <p>The format of include file and libraries changed in v3.25 to match HI-TECH C compiler format.</p>

TABLE 5-16: LINKING OPTIONS (CONTINUED)

Option	Definition
<code>-llibrary</code>	<p>Search the library named <i>library</i> when linking. The linker searches a standard list of directories for the library, which is actually a file named <code>liblibrary.a</code>. The linker then uses this file as if it had been specified precisely by name.</p> <p>It makes a difference where in the command you write this option; the linker processes libraries and object files in the order they are specified. Thus, <code>foo.o -lz bar.o</code> searches library <i>z</i> after file <code>foo.o</code> but before <code>bar.o</code>. If <code>bar.o</code> refers to functions in <code>libz.a</code>, those functions may not be loaded.</p> <p>The directories searched include several standard system directories, plus any that you specify with <code>-L</code>.</p> <p>Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an <code>-l</code> option (e.g., <code>-lmylib</code>) and specifying a file name (e.g., <code>libmylib.a</code>) is that <code>-l</code> searches several directories, as specified. By default the linker is directed to search:</p> <pre><install-path>\lib</pre> <p>for libraries specified with the <code>-l</code> option. This behavior can be overridden using the environment variables defined in Section 19.4 “Predefined Macro Names”.</p>
<code>-nodefaultlibs</code>	Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The compiler may generate calls to <code>memcpy</code> , <code>memset</code> and <code>memcpy</code> . These entries are usually resolved by entries in the standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.
<code>-nostdlib</code>	Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker. The compiler may generate calls to <code>memcpy</code> , <code>memset</code> and <code>memcpy</code> . These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.
<code>-s</code>	Remove all symbol table and relocation information from the executable.
<code>-T script</code>	Specify the linker script file, <i>script</i> , to be used at link time. This option is translated into the equivalent <code>-T</code> linker option.
<code>-u symbol</code>	Pretend <i>symbol</i> is undefined to force linking of library modules to define the symbol. It is legitimate to use <code>-u</code> multiple times with different symbols to force loading of additional library modules.
<code>-Wl,option</code>	Pass <i>option</i> as an option to the linker. If <i>option</i> contains commas, it is split into multiple options at the commas. For example, to generate a map file, use <code>-Wl, -Map=Project.map</code> .
<code>-Xlinker option</code>	Pass <i>option</i> as an option to the linker. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

5.7.10 Options for Directory Search

The following options specify to the compiler where to find directories and files to search.

TABLE 5-17: DIRECTORY SEARCH OPTIONS

Option	Definition
<code>-specs=file</code>	Process file after the compiler reads in the standard <code>specs</code> file, in order to override the defaults that the <code>xc16-gcc</code> driver program uses when determining what switches to pass to <code>xc16-cc1</code> , <code>xc16-as</code> , <code>xc16-ld</code> , etc. More than one <code>-specs=file</code> can be specified on the command line, and they are processed in order, from left to right.

5.7.11 Options for Code Generation Conventions

Options of the form `-fflag` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. In the table below, only one of the forms is listed (the one that is not the default.)

TABLE 5-18: CODE GENERATION CONVENTION OPTIONS

Option	Definition
<code>-fargument-alias</code> <code>-fargument-noalias</code> <code>-fargument-noalias-global</code>	Specify the possible relationships among parameters and between parameters and global data. <code>-fargument-alias</code> specifies that arguments (parameters) may alias each other and may alias global storage. <code>-fargument-noalias</code> specifies that arguments do not alias each other, but may alias global storage. <code>-fargument-noalias-global</code> specifies that arguments do not alias each other and do not alias global storage. Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.
<code>-fcall-saved-reg</code>	Treat the register named <i>reg</i> as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register <i>reg</i> if they use it. It is an error to use this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results. A different sort of disaster will result from the use of this flag for a register in which function values may be returned. This flag should be used consistently through all modules.
<code>-fcall-used-reg</code>	Treat the register named <i>reg</i> as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register <i>reg</i> . It is an error to use this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model will produce disastrous results. This flag should be used consistently through all modules.
<code>-ffixed-reg</code>	Treat the register named <i>reg</i> as a fixed register; generated code should never refer to it (except perhaps as a Stack Pointer, Frame Pointer or in some other fixed role). <i>reg</i> must be the name of a register, e.g., <code>-ffixed-w3</code> .
<code>-fno-ident</code>	Ignore the <code>#ident</code> directive.

TABLE 5-18: CODE GENERATION CONVENTION OPTIONS (CONTINUED)

Option	Definition
<code>-fpack-struct</code>	Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code sub-optimal, and the offsets of structure members won't agree with system libraries. The dsPIC® DSC device requires that words be aligned on even byte boundaries, so care must be taken when using the packed attribute to avoid run time addressing errors.
<code>-fpcc-struct-return</code>	Return short <code>struct</code> and <code>union</code> values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing capability between the 16-bit compiler compiled files and files compiled with other compilers. Short structures and unions are those whose size and alignment match that of an integer type.
<code>-fno-short-double</code>	By default, the compiler uses a <code>double</code> type equivalent to <code>float</code> . This option makes <code>double</code> equivalent to <code>long double</code> . Mixing this option across modules can have unexpected results if modules share double data either directly through argument passage or indirectly through shared buffer space. Libraries provided with the product function with either switch setting.
<code>-fshort-enums</code>	Allocate to an <code>enum</code> type only as many bytes as it needs for the declared range of possible values. Specifically, the <code>enum</code> type will be equivalent to the smallest integer type which has enough room.
<code>-fverbose-asm</code> <code>-fno-verbose-asm</code>	Put extra commentary information in the generated assembly code to make it more readable. <code>-fno-verbose-asm</code> , the default, causes the extra information to be omitted and is useful when comparing two assembler files.

5.8 MPLAB X IDE TOOLCHAIN OR MPLAB IDE TOOLSUITE EQUIVALENTS

For information on related compiler options in MPLAB X IDE, see:

Chapter 4. “XC16 Toolchain and MPLAB X IDE”

Chapter 6. Device-Related Features

6.1 INTRODUCTION

The MPLAB XC16 C Compiler provides some features that are purely device-related.

- Device Support
- Device Header Files
- Stack
- Configuration Bit Access
- Using SFRs in MCUs
- Bit-Reversed and Modulo Addressing

6.2 DEVICE SUPPORT

As discussed in **Chapter 1. “Compiler Overview”**, the compiler supports all Microchip 16-bit devices; dsPIC30/33 digital signal controls (DSCs) and PIC24 microcontrollers (MCUs).

To determine the device support for your version of the compiler, consult the file `Readme_XC16.html` in the `docs` subfolder of the compiler installation folder. For example:

```
C:\Program Files (x86)\Microchip\xc16\v1.10\docs\Readme_XC16.html
```

6.3 DEVICE HEADER FILES

One header file that is typically included in each C source file you will write is `xc.h`, a generic header file that will include other device- and architecture-specific header files when you build your project.

Inclusion of this file will allow access to SFRs via special variables, as well as macros which allow special memory access or inclusion of special instructions.

Avoid including chip-specific header files into your code, as this will reduce portability. However, device-specific compiler header files are stored in the `support\family\h` directory for reference.

For information about assembly include files (`*.inc`), see the assembler documentation.

6.3.1 Register Definition Files

The processor header files described in **Section 6.5 “Configuration Bit Access”** name all SFRs for each part, but they do not define the addresses of the SFRs. A separate set of device-specific linker script files, one per part, is distributed in the `support\family\gld` directory. These linker script files define the SFR addresses. To use one of these files, specify the linker command-line option:

```
-T p30fxxxx.gld
```

where `xxxx` corresponds to the device part number.

For example, assuming that there is a file named `app2010.c` that contains an application for the dsPIC30F2010 part, then it may be compiled and linked using the following command line:

```
xc16-gcc -mcpu=30f2010 -o app2010.out -T p30f2010.gld app2010.c
```

The `-o` command-line option names the output executable file, and the `-T` option gives the linker script name for the dsPIC30F2010 part. If `p30f2010.gld` is not found in the current directory, the linker searches in its known library paths. The default search path includes all locations of preinstalled libraries and linker scripts.

You should copy the appropriate linker script file (supplied with the compiler) into your project directory before any project-specific modifications are made.

6.4 STACK

The 16-bit devices use what is referred to in this user's guide as a "software stack". This is the typical stack arrangement employed by most computers and is ordinary data memory accessed by a push-and-pop type instruction and a stack pointer register. The term "hardware stack" is used to describe the stack employed by Microchip 8-bit devices, which is only used for storing function return addresses.

The 16-bit devices dedicate register W15 for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. The stack grows upward, towards higher memory addresses.

The dsPIC DSC device also supports stack overflow detection. If the Stack Pointer Limit register, SPLIM, is initialized, the device will test for overflow on all stack operations. If an overflow should occur, the processor will initiate a stack error exception. By default, this will result in a processor Reset. Applications may also install a stack error exception handler by defining an interrupt function named `_StackError`. See **Chapter 14. "Interrupts"** for details.

The C run-time startup module initializes the Stack Pointer (W15) and the Stack Pointer Limit register during the startup and initialization sequence. The initial values are normally provided by the linker, which allocates the largest stack possible from unused data memory. The location of the stack is reported in the link map output file. Applications can ensure that at least a minimum-sized stack is available with the `--stack` linker command-line option. See the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for details.

Alternatively, a stack of specific size may be allocated with a user-defined section from an assembly source file. In the following example, 0x100 bytes of data memory are reserved for the stack:

```
.section *,data,stack
.space 0x100
```

The linker will allocate an appropriately sized section and initialize `__SP_init` and `__SPLIM_init` so that the run-time startup code can properly initialize the stack. Note that since this is a normal assembly code section, attributes such as `address` may be used to further define the stack. Please see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for more information.

6.5 CONFIGURATION BIT ACCESS

Microchip devices have several locations which contain the configuration bits or fuses. These bits specify fundamental device operation, such as the oscillator mode, watch-dog timer, programming mode and code protection. Failure to correctly set these bits may result in code failure or a non-running device.

6.5.1 Configuration Settings Using `#pragma config`

Configuration Settings may be made using the preprocessor directive `#pragma config` and settings macros specified under the `docs` subdirectory of the compiler install directory.

The directive format options are:

```
#pragma config setting = state|value  
#pragma config register = value
```

where `setting` is a configuration setting descriptor (e.g., `WDT`), `state` is a descriptive value (e.g., `ON`) and `value` is a numerical value. The register token may represent a whole configuration word register, e.g., `CONFIG1L`.

A list of all available settings by device may be found from MPLAB X IDE, Dashboard window, **Compiler Help** button or from the command-line under:

```
<MPLAB XC16 Installation folder>/vx.xx/docs/config_index.html
```

6.5.2 Configuration Settings Using Macros

Configuration Settings macros are provided that can be used to set configuration bits. For example, to set the `FOSC` bit using a macro, the following line of code can be inserted before the beginning of your C source code:

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

This would enable the external clock with the PLL set to 16x and enable clock switching and fail-safe clock monitoring.

Similarly, to set the `FBORPOR` bit:

```
_FBORPOR(PBOR_ON & BORV_27 & PWRT_ON_64 & MCLR_DIS);
```

This would enable Brown-out Reset at 2.7 Volts and initialize the Power-up timer to 64 milliseconds and configure the use of the `MCLR` pin for I/O.

Configuration Settings macros are defined in compiler header files for each device. Please refer to your device's header files for a complete listing of related macros. Header files are located, by default, in:

```
<MPLAB XC16 Installation folder>/vx.xx/support/device/h
```

where `vx.xx` is the compiler version and `device` is your 16-bit device family.

6.6 USING SFRS

The Special Function Registers (SFRs) are registers which control aspects of the MCU operation or that of peripheral modules on the device. These registers are device memory mapped, which means that they appear at, and can be accessed using, specific addresses in the device's data memory space. Individual bits within some registers control independent features. Some registers are read-only; some are write-only. See your device data sheet for more information.

Memory-mapped SFRs are accessed by special C variables that are placed at the address of the register. These variables can be accessed like any ordinary C variable so that no special syntax is required to access SFRs.

The SFR variable identifiers are predefined in header files and are accessible once you have included the `<xc.h>` header file (see **Section 6.3 “Device Header Files”**) into your source code. Structures with bit-fields are also defined so you may access bits within a register in your source code.

A linker script file for the appropriate device must be linked into your project to ensure the SFR variable identifiers are linked to the correct address. MPLAB IDE will link in a default linker script, but a linker script file must be explicitly specified if you are driving the command-line toolchain. Linker scripts have a `.gld` extension (e.g. `p30F6014.gld`) and basic files are provided with the compiler.

The convention in the processor header files is that each SFR is named, using the same name that appears in the data sheet for the part – for example, `CORCON` for the Core Control register. If the register has individual bits that might be of interest, then there will also be a structure defined for that SFR, and the name of the structure will be the same as the SFR name, with “bits” appended. For example, `CORCONbits` for the Core Control register. The individual bits (or bit-fields) are named in the structure using the names in the data sheet – for example `PSV` for the `PSV` bit of the `CORCON` register.

Here is the complete definition of `CORCON` (subject to change):

```
/* CORCON: CPU Mode control Register */
extern volatile unsigned int CORCON __attribute__((__sfr__));
typedef struct tagCORCONBITS {
    unsigned IF      :1;    /* Integer/Fractional mode          */
    unsigned RND      :1;    /* Rounding mode                    */
    unsigned PSV      :1;    /* Program Space Visibility enable  */
    unsigned IPL3      :1;
    unsigned ACCSAT    :1;    /* Acc saturation mode              */
    unsigned SATDW     :1;    /* Data space write saturation enable */
    unsigned SATB      :1;    /* Acc B saturation enable          */
    unsigned SATA      :1;    /* Acc A saturation enable          */
    unsigned DL        :3;    /* DO loop nesting level status     */
    unsigned           :4;
} CORCONBITS;
extern volatile CORCONBITS CORCONbits __attribute__((__sfr__));
```

Note: The symbols `CORCON` and `CORCONbits` refer to the same register and will resolve to the same address at link time.

See *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for more information on using linker scripts.

For example, the following is a sample real-time clock. It uses an SFR, e.g. `TMR1`, as well as bits within an SFR, e.g. `T1CONbits.TCS`. Descriptions for these SFRs are found in the `p30F6014.h` file (this file will automatically be included by `<xc.h>` so you do not need to include this into your source code). This file would be linked with the device specific linker script which is `p30F6014.gld`.

EXAMPLE 6-1: SAMPLE REAL-TIME CLOCK

```
/*
** Sample Real Time Clock for dsPIC
**
** Uses Timer1, TCY clock timer mode
** and interrupt on period match
**/

#include <xc.h>
```

```
/* Timer1 period for 1 ms with FOSC = 20 MHz */
#define TMR1_PERIOD 0x1388

struct clockType
{
    unsigned int timer;      /* countdown timer, milliseconds */
    unsigned int ticks;      /* absolute time, milliseconds */
    unsigned int seconds;    /* absolute time, seconds */
} volatile RTclock;

void reset_clock(void)
{
    RTclock.timer = 0;        /* clear software registers */
    RTclock.ticks = 0;
    RTclock.seconds = 0;

    TMR1 = 0;                 /* clear timer1 register */
    PR1 = TMR1_PERIOD;        /* set period1 register */
    T1CONbits.TCS = 0;        /* set internal clock source */
    IPC0bits.T1IP = 4;        /* set priority level */
    IFS0bits.T1IF = 0;        /* clear interrupt flag */
    IEC0bits.T1IE = 1;        /* enable interrupts */

    SRbits.IPL = 3;          /* enable CPU priority levels 4-7*/
    T1CONbits.TON = 1;        /* start the timer*/
}

void __attribute__((__interrupt__,__auto_psv__)) _T1Interrupt(void)
{ static int sticks=0;

    if (RTclock.timer > 0)    /* if countdown timer is active */
        RTclock.timer -= 1;  /* decrement it */
    RTclock.ticks++;          /* increment ticks counter */
    if (sticks++ > 1000)
    {
        /* if time to rollover */
        sticks = 0;          /* clear seconds ticks */
        RTclock.seconds++;    /* and increment seconds */
    }

    IFS0bits.T1IF = 0;        /* clear interrupt flag */
    return;
}
```

6.7 BIT-REVERSED AND MODULO ADDRESSING

Bit-reversed and modulo addressing is supported on all dsPIC DSC devices.

Bit-reversed addressing is used for simplifying and speeding-up the writes to X-space data arrays in FFT (Fast Fourier Transform) algorithms. When enabled, pre-increment or post-increment addressing modes will reverse the lower order address bits used by instructions.

Modulo, or circular, addressing provides an automated means to support circular data buffers using the dsPIC hardware. When used, software no longer needs to perform data address boundary checks on arrays.

The compiler does not directly support the use of bit-reversed and modulo addressing; that is, it cannot generate code from C source that assumes these addressing modes are enabled when accessing memory. If either of these addressing modes are set up on the target device, then it is the programmer's responsibility to ensure that the compiler does not use those registers that are specified to use either modulo or bit-reversed addressing as pointers. Particular care must be exercised if interrupts can occur while one of these addressing modes is enabled.

It is possible to define arrays in C that will be suitably aligned in memory for modulo addressing by hand-written assembly language functions. The `aligned` attribute may be used to define arrays that are positioned for use as incrementing modulo buffers. Initialization of the start and end addresses, as well as the registers that modulo address is applied must be written by hand to match the array specification. The `reverse` attribute may be used to define arrays that are positioned for use as decrementing modulo buffers. For more information on these attributes, see **Section 13.2.1 "Function Specifiers"**. For more information on bit-reversed or modulo addressing, see Chapter 3 of the "*dsPIC30F Family Reference Manual*" (DS70046).

Chapter 7. Differences Between MPLAB XC16 and ANSI C

This compiler conforms to the ANS X3.159-1989 Standard for programming languages. This is commonly called the C89 Standard. It is referred to as the ANSI C Standard in this manual. Some features from the later standard C99 are also supported.

- Divergence from the ANSI C Standard
- Extensions to the ANSI C Standard
- Implementation-Defined Behavior

7.1 DIVERGENCE FROM THE ANSI C STANDARD

There are no divergences from the ANSI C standard.

7.2 EXTENSIONS TO THE ANSI C STANDARD

The MPLAB XC16 C Compiler provides extensions to the ANSI C standard in these areas: keywords and expressions.

7.2.1 Keyword Differences

The new keywords are part of the base GCC implementation and the discussions in the referenced sections are based on the standard GCC documentation, tailored for the specific syntax and semantics of the 16-bit compiler port of GCC.

- Specifying Attributes of Variables – **Section 8.12 “Variable Attributes”**
- Specifying Attributes of Functions – **Section 13.2.1 “Function Specifiers”**
- Inline Functions – **Section 13.6 “Inline Functions”**
- Variables in Specified Registers – **Section 10.9 “Allocation of Variables to Registers”**
- Complex Numbers – **Section 8.8 “Complex Data Types”**

7.2.2 Expression Differences

Expression differences are:

- Binary Constants – **Section 8.9 “Literal Constant Types and Formats”**.

7.3 IMPLEMENTATION-DEFINED BEHAVIOR

Certain features of the ANSI C standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler.

The exact behavior of the MPLAB XC16 C Compiler is detailed throughout this documentation, and is fully summarized in **Appendix A. “Implementation-Defined Behavior”**.

NOTES:

Chapter 8. Supported Data Types and Variables

8.1 INTRODUCTION

The MPLAB XC16 C Compiler supports a variety of data types and qualifiers (attributes). These data types and variables are discussed here. For information on where variables are stored in memory, see **Chapter 10. “Memory Allocation and Access”**.

- Identifiers
- Integer Data Types
- Floating-Point Data Types
- Fixed-Point Data Types
- Structures and Unions
- Pointer Types
- Complex Data Types
- Literal Constant Types and Formats
- Standard Type Qualifiers
- Compiler-Specific type Qualifiers
- Variable Attributes

8.2 IDENTIFIERS

A C variable identifier (as well as a function identifier) is a sequence of letters and digits where the underscore character, “_”, counts as a letter. Identifiers cannot start with a digit. Although they may start with an underscore, such identifiers are reserved for the compiler's use and should not be defined by your programs. Such is not the case for assembly domain identifiers, which often begin with an underscore, see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.

Identifiers are case sensitive, so `main` is different from `Main`.

All characters are significant in an identifier, although identifiers longer than 31 characters in length are less portable.

8.3 INTEGER DATA TYPES

Table 8-1 shows integer data types that are supported in the compiler. All unspecified or signed integer data types are arithmetic type signed integer. All unsigned integer data types are arithmetic type unsigned integer.

TABLE 8-1: INTEGER DATA TYPES

Type	Bits	Min	Max
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int	16	-32768	32767
unsigned int	16	0	65535
long, signed long	32	-2 ³¹	2 ³¹ - 1
unsigned long	32	0	2 ³² - 1
long long*, signed long long*	64	-2 ⁶³	2 ⁶³ - 1
unsigned long long*	64	0	2 ⁶⁴ - 1

* ANSI-89 extension

There is no type for storing single bit quantities.

All integer values are specified in little endian format, which means:

- The least significant byte (LSB) is stored at the lowest address
- The least significant bit (LSb) is stored at the lowest-numbered bit position

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

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

As another example, the long value of 0x12345678 is stored in registers w4 and w5:

w4	w5
0x5678	0x1234

Signed values are stored as a two's complement integer value.

Preprocessor macros that specify integer minimum and maximum values are available after including <limits.h> in your source code, located by default in:

```
<install directory>\include
```

As the size of data types is not fully specified by the ANSI Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

For information on implementation-defined behavior of integers, see **Section A.6 "Integers"**.

8.3.1 Double-Word Integers

The compiler supports data types for integers that are twice as long as long int. Simply write long long int for a signed integer, or unsigned long long int for an unsigned integer. To make an integer constant of type long long int, add the suffix LL to the integer. To make an integer constant of type unsigned long long int, add the suffix ULL to the integer.

You can use these types in arithmetic like any other integer types.

8.3.2 char Types

The compiler supports data types for `char`, which defaults to `signed char`. An option can be used to use `unsigned char` as the default, see **Section 5.7.3 “Options for Controlling the C Dialect”**.

It is a common misconception that the C `char` types are intended purely for ASCII character manipulation. This is not true; indeed, the C language makes no guarantee that the default character representation is even ASCII (however, this implementation does use ASCII as the character representation). The `char` types are simply the smallest of the multi-bit integer sizes, and behave in all respects like integers. The reason for the name “char” is historical and does not mean that `char` can only be used to represent characters. It is possible to freely mix `char` values with values of other types in C expressions. With the MPLAB XC16 C Compiler, the `char` types will commonly be used for a number of purposes: as 8-bit integers, as storage for ASCII characters, and for access to I/O locations.

8.4 FLOATING-POINT DATA TYPES

The compiler uses the IEEE-754 format. Table 8-2 shows floating point data types that are supported. All floating point data types are arithmetic type real.

TABLE 8-2: FLOATING POINT DATA TYPES

Type	Bits	E Min	E Max	N Min	N Max
float	32	-126	127	2^{-126}	2^{128}
double*	32	-126	127	2^{-126}	2^{128}
long double	64	-1022	1023	2^{-1022}	2^{1024}

E = Exponent

N = Normalized (approximate)

* `double` is equivalent to `long double` if `-fno-short-double` is used.

All floating point values are specified in little endian format, which means:

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

As an example, the `double` value of 1.2345678 is stored at address `0x100` as follows:

0x100	0x101	0x102	0x103
0x51	0x06	0x9E	0x3F

As another example, the long value of 1.2345678 is stored in registers `w4` and `w5`:

w4	w5
0x0651	0x3F9E

Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type.

Preprocessor macros that specify valid ranges are available after including `<float.h>` in your source code.

For information on implementation-defined behavior of floating point numbers, see **Section A.7 “Floating Point”**.

8.5 FIXED-POINT DATA TYPES

Table 8-3 shows fixed-point data types that are supported by the compiler when the `-menable-fixed` command line option is specified. See **Chapter 9. “Fixed-Point Arithmetic Support”** for more details on the compiler's support for the fixed-point C language dialect. If the signed or unsigned type specifier is not present, the type is assumed to be signed.

TABLE 8-3: FIXED POINT INTEGER DATA TYPES

Type	Bits	Min	Max
<code>_Fract</code>	16	-1.0	$1.0 - 2^{-15}$
<code>short _Fract</code>	16	-1.0	$1.0 - 2^{-15}$
<code>signed _Fract</code>	16	-1.0	$1.0 - 2^{-15}$
<code>signed short _Fract</code>	16	-1.0	$1.0 - 2^{-15}$
<code>unsigned _Fract</code>	16	0.0	$1.0 - 2^{-15}$
<code>unsigned short _Fract</code>	16	0.0	$1.0 - 2^{-15}$
<code>long _Fract</code>	32	-1.0	$1.0 - 2^{-31}$
<code>signed long _Fract</code>	32	-1.0	$1.0 - 2^{-31}$
<code>unsigned long _Fract</code>	32	0.0	$1.0 - 2^{-31}$
<code>_Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>short _Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>long _Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>signed _Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>signed short _Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>signed long _Accum</code>	40	-256.0	$256.0 - 2^{-31}$
<code>unsigned _Accum</code>	40	0.0	$256.0 - 2^{-31}$
<code>unsigned short _Accum</code>	40	0.0	$256.0 - 2^{-31}$
<code>unsigned long _Accum</code>	40	0.0	$256.0 - 2^{-31}$

As with integer and floating point data types, all fixed-point values are represented in a little endian format, which means:

- The Least Significant Byte (LSB) is stored at the lowest address
- The Least Significant bit (LSb) is stored at the lowest-numbered bit position

8.6 STRUCTURES AND UNIONS

MPLAB XC16 C Compiler supports `struct` and `union` types. Structures and unions only differ in the memory offset applied to each member.

These types will be at least 1 byte wide. Bit-fields are fully supported in structures.

Structures and unions may be passed freely as function arguments and function return values. Pointers to structures and unions are fully supported.

Implementation-defined behavior of structures, unions and bit-fields is described in **Section A.10 “Structures, Unions, Enumerations and Bit-Fields”**.

8.6.1 Structure and Union Qualifiers

The MPLAB XC16 C Compiler supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example, the structure is qualified `const`.

```
const struct foo {
    int number;
    int *ptr;
} record = { 0x55, &i };
```

In this case, the entire structure may be placed into the program space where each member will be read-only. Remember that all members are usually initialized if a structure is `const` as they cannot be initialized at runtime.

If the members of the structure were individually qualified `const`, but the structure was not, then the structure would be positioned into RAM, but each member would be still be read-only. Compare the following structure with the one above.

```
struct {
    const int number;
    int * const ptr;
} record = { 0x55, &i};
```

8.6.2 Bit-fields in Structures

The MPLAB XC16 C Compiler fully supports bit-fields in structures.

Bit-fields are, by default, signed `int`. They may be made an unsigned `int` `bit-field` by using a command line option, see **Section 5.7.3 “Options for Controlling the C Dialect”**.

The first bit defined will be the LSb of the word in which it will be stored.

The compiler supports bit-fields with any bit size, up to the size of the underlying type. Any integral type can be made into a bit-field. The allocation does not normally cross a bit boundary natural to the underlying type.

For example:

```
struct foo {
    long long i:40;
    int j:16;
    char k:8;
} x;

struct bar {
    long long I:40;
    char J:8;
    int K:16;
} y;
```

`struct foo` will have a size of 10 bytes using the compiler. `i` will be allocated at bit offset 0 (through 39). There will be 8 bits of padding before `j`, allocated at bit offset 48. If `j` were allocated at the next available bit offset, 40, it would cross a storage boundary for a 16 bit integer. `k` will be allocated after `j`, at bit offset 64. The structure will contain 8 bits of padding at the end to maintain the required alignment in the case of an array. The alignment is 2 bytes because the largest alignment in the structure is 2 bytes.

`struct bar` will have a size of 8 bytes using the compiler. `I` will be allocated at bit offset 0 (through 39). There is no need to pad before `J` because it will not cross a storage boundary for a `char`. `J` is allocated at bit offset 40. `K` can be allocated starting at bit offset 48, completing the structure without wasting any space.

Unnamed bit-fields may be declared to pad out unused space between active bits in control registers. For example:

```
struct foo {
    unsigned lo : 1;
    unsigned    : 6;
    unsigned hi : 1;
} x;
```

A structure with bit-fields may be initialized by supplying a *comma-separated* list of initial values for each field. For example:

```
struct foo {
    unsigned lo : 1;
    unsigned mid : 6;
    unsigned hi : 1;
} x = {1, 8, 0};
```

Structures with unnamed bit-fields may be initialized. No initial value should be supplied for the unnamed members, for example:

```
struct foo {
    unsigned lo : 1;
    unsigned    : 6;
    unsigned hi : 1;
} x = {1, 0};
```

will initialize the members `lo` and `hi` correctly.

8.7 POINTER TYPES

There are two basic pointer types supported by the MPLAB XC16 C Compiler: data pointers and function pointers. Data pointers hold the addresses of variables which can be indirectly read, and possibly indirectly written, by the program. Function pointers hold the address of an executable function which can be called indirectly via the pointer.

8.7.1 Combining Type Qualifiers and Pointers

It is helpful to first review the ANSI C standard conventions for definitions of pointer types.

Pointers can be qualified like any other C object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual pointer itself, which is treated like any ordinary C variable and has memory reserved for it. The second is the target, or targets, that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following:

```
target_type_ &_qualifiers * pointer's_qualifiers pointer's_name;
```

Any qualifiers to the right of the * (i.e., next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the * relate to the pointer's targets. This makes sense since it is also the * operator that dereferences a pointer, which allows you to get from the pointer variable to its current target.

Here are three examples of pointer definitions using the `volatile` qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *          vip ;  
int          * volatile ivp ;  
volatile int * volatile vivp ;
```

The first example is a pointer called `vip`. It contains the address of `int` objects that are qualified `volatile`. The pointer itself – the variable that holds the address – is *not* `volatile`; however, the objects that are accessed when the pointer is dereferenced are treated as being `volatile`. In other words, the target objects accessible via the pointer may be externally modified.

The second example is a pointer called `ivp` which also contains the address of `int` objects. In this example, the pointer itself is `volatile`, that is, the address the pointer contains may be externally modified; however, the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile`, and which also holds the address of `volatile` objects.

Bear in mind that one pointer can be assigned the addresses of many objects; for example, a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

Note: Care must be taken when describing pointers. Is a “const pointer” a pointer that points to `const` objects, or a pointer that is `const` itself? You can talk about “pointers to `const`” and “const pointers” to help clarify the definition, but such terms may not be universally understood.

8.7.2 Data Pointers

All standard data pointers are 16 bits wide. This is sufficient to access the full data memory space.

These pointers are also able to access `const`-qualified objects, although in the program memory space, `const`-qualified objects appear in a unique memory range in the data space using the PSV window. In this case, the `-mconst-in-data` option should not be in force (see **Section 5.7.1 “Options Specific to 16-Bit Devices”**.)

Pointers which access the managed PSV space are 32-bits wide. The extra space allows these pointers to access any PSV page.

A set of special purpose, 32-bit data pointers are also available. See **Chapter 10. “Memory Allocation and Access”** for more information.

8.7.3 Function Pointers

The MPLAB XC16 C Compiler fully supports pointers to functions, which allows functions to be called indirectly. Function pointers are always 16 bits wide.

In the small code model (up to 32 kWords of code), 16-bit wide function pointers can access any function location. In the large code model, which supports more than 32 kWords of code, pointers hold the address of a `GOTO` instruction in a lookup table. These instructions are able to reach any memory location, but the lookup table itself is located in the lower program memory, thus allowing the pointers themselves to remain as 16-bit wide variables.

As function pointers are only 16-bits wide, these pointers cannot point beyond the first 64K of FLASH. Should the address of a function that is allocated beyond the first 64K of FLASH be taken, the linker will arrange for a `handle` section to be generated. The `handle` section will always be allocated within the first 64K. Each handle provides a level of indirection which allows 16-bit pointers to access the full range of FLASH. This operation may be disabled with the `--no-handles` linker option.

8.7.4 Special Pointer Targets

Pointers and integers are not interchangeable. Assigning an integer value to a pointer will generate a warning to this effect. For example:

```
const char * cp = 0x123; // the compiler will flag this as bad code
```

There is no information in the integer, 0x123, relating to the type, size or memory location of the destination. Avoid assigning an integer (whether it be a constant or variable) to a pointer at all times. Addresses assigned to pointers should be derived from the address operator “&” that C provides.

In instances where you need to have a pointer reference a seemingly arbitrary address or address range, consider defining an object or label at the desired location. If the object is defined in assembly code, use a C declaration (using the `extern` keyword) to create a C object which links in with the external object and whose address can be taken.

Take care when comparing (subtracting) pointers. For example:

```
if(cp1 == cp2)
    ; take appropriate action
```

The ANSI C standard only allows pointer comparisons when the two pointer targets are the same object. The address may extend to one element past the end of an array.

Comparisons of pointers to integer constants are even more risky, for example:

```
if(cp1 == 0x246)
    ; take appropriate action
```

A `NULL` pointer is the one instance where a constant value can be safely assigned to a pointer. A `NULL` pointer is numerically equal to 0 (zero), but since they do not guarantee to point to any valid object and should not be dereferenced, this is a special case imposed by the ANSI C standard. Comparisons with the macro `NULL` are also allowed.

8.8 COMPLEX DATA TYPES

The compiler supports complex data types. You can declare both complex integer types and complex floating types, using the keyword `__complex__`.

For example, `__complex__ float x;` declares `x` as a variable whose real part and imaginary part are both of type `float`. `__complex__ short int y;` declares `y` to have real and imaginary parts of type `short int`.

To write a constant with a complex data type, use the suffix 'i' or 'j' (either one; they are equivalent). For example, `2.5fi` has type `__complex__ float` and `3i` has type `__complex__ int`. Such a constant is a purely imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression `exp`, write `__real__ exp`. Similarly, use `__imag__` to extract the imaginary part. For example;

```
__complex__ float z;
float r;
float i;

r = __real__ z;
i = __imag__ z;
```

The operator, `~`, performs complex conjugation when used on a value with a complex type.

The compiler can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). The debugging information format has no way to represent noncontiguous allocations like these, so the compiler describes noncontiguous complex variables as two separate variables of noncomplex type. If the variable's actual name is `foo`, the two fictitious variables are named `foo$real` and `foo$imag`.

8.9 LITERAL CONSTANT TYPES AND FORMATS

A literal constant is used to represent a numerical value in the source code; for example, 123 is a constant. Like any value, a literal constant must have a C type. In addition to a literal constant's type, the actual value can be specified in one of several formats. The format of integral literal constants specifies their radix. MPLAB XC16 supports the ANSI standard radix specifiers as well as ones which enables binary constants to be specified in C code.

The formats used to specify the radices are given in Table 8-4. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

TABLE 8-4: RADIX FORMATS

Radix	Format	Example
binary	<code>0b number</code> or <code>0B number</code>	0b10011010
octal	<code>0 number</code>	0763
decimal	<code>number</code>	129
hexadecimal	<code>0x number</code> or <code>0X number</code>	0x2F

Any integral literal constant will have a type of `int`, `long int` or `long long int`, so that the type can hold the value without overflow. Literal constants specified in octal or hexadecimal may also be assigned a type of `unsigned int`, `unsigned long int` or `unsigned long long int` if the signed counterparts are too small to hold the value.

The default types of literal constants may be changed by the addition of a suffix after the digits, e.g. 23U, where U is the suffix. Table 8-5 shows the possible combination of suffixes and the types that are considered when assigning a type. So, for example, if the suffix `l` is specified and the value is a decimal literal constant, the compiler will assign the type `long int`, if that type will hold the literal constant; otherwise, it will assigned `long long int`. If the literal constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

TABLE 8-5: SUFFIXES AND ASSIGNED TYPES

Suffix	Decimal	Octal or Hexadecimal
<code>u</code> or <code>U</code>	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code>	<code>unsigned int</code> <code>unsigned long int</code> <code>unsigned long long int</code>
<code>l</code> or <code>L</code>	<code>long int</code> <code>long long int</code>	<code>long int</code> <code>unsigned long int</code> <code>long long int</code> <code>unsigned long long int</code>
<code>u</code> or <code>U</code> , and <code>l</code> or <code>L</code>	<code>unsigned long int</code> <code>unsigned long long int</code>	<code>unsigned long int</code> <code>unsigned long long int</code>
<code>ll</code> or <code>LL</code>	<code>long long int</code>	<code>long long int</code> <code>unsigned long long int</code>
<code>u</code> or <code>U</code> , and <code>ll</code> or <code>LL</code>	<code>unsigned long long int</code>	<code>unsigned long long int</code>

Here is an example of code that may fail because the default type assigned to a literal constant is not appropriate:

```
unsigned long int result;
unsigned char shifter;

void main(void)
{
    shifter = 20;
    result = 1 << shifter;
    // code that uses result
}
```


Supported Data Types and Variables

The literal constant 1 will be assigned an `int` type; hence the result of the shift operation will be an `int` and the upper bits of the `long` variable, `result`, can never be set, regardless of how much the literal constant is shifted. In this case, the value 1 shifted left 20 bits will yield the result 0, not 0x100000.

The following uses a suffix to change the type of the literal constant, hence ensure the shift result has an `unsigned long` type.

```
result = 1UL << shifter;
```

Floating-point literal constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type. In MPLAB XC16, the `double` type equates to a 32-bit `float` type. The command line option, `-fno-short-double`, may be used to specify `double` as a 64-bit `long double` type.

Fixed-point literal constants look like floating point numbers, suffixed with combinations of `[u][h,l]<r,k>`. The suffix `u` means unsigned. The suffixes `h` and `l` signify short and long respectively. The suffix `r` denotes a `_Fract` type and `k` specifies an `_Accum` type. So for example, `-1.0r` is a signed `_Fract` and `0.5uhk` is an unsigned short `_Accum`.

Character literal constants are enclosed by single quote characters, `'`, for example `'a'`. A character literal constant has `int` type, although this may be optimized to a `char` type later in the compilation.

Multi-byte character literal constants are supported by this implementation.

String constants, or string literals, are enclosed by double quote characters `"`, for example `"hello world"`. The type of string literal constants is `const char *` and the character that make up the string may be stored in the program memory.

To comply with the ANSI C standard, the compiler does not support the extended character set in characters or character arrays. Instead, they need to be escaped using the backslash character, as in the following example:

```
const char name[] = "Bj\xf8k";  
printf("%s's Resum\xe9", name); \\ prints "Björk's Resumé"
```

Defining and initializing a non-`const` array (i.e. not a pointer definition) with a string, for example:

```
char ca[] = "two";           // "two" different to the above
```

is a special case and produces an array in data space which is initialized at startup with the string `"two"`, whereas a string literal constant used in other contexts represents an unnamed array, accessed directly from its storage location.

The compiler will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialize an array residing in the data space as shown in the last statement in the previous example.

Two adjacent string literal constants (i.e. two strings separated *only* by white space) are concatenated by the C preprocessor. Thus:

```
const char * cp = "hello " "world";
```

will assign the pointer with the address of the string `"hello world"`.

8.10 STANDARD TYPE QUALIFIERS

Type qualifiers provide additional information regarding how an object may be used. The MPLAB XC16 compiler supports both ANSI C qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC MCU and dsPIC DSC architectures.

8.10.1 Const Type Qualifier

The compiler supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object is read only and will not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning or error.

User-defined objects declared `const` are placed, by default, in the program space and may be accessed via the program visibility space, see **Section 10.4 “Variables in Program Space”**. Usually a `const` object must be initialized when it is declared, as it cannot be assigned a value at any point at runtime. For example:

```
const int version = 3;
```

will define `version` as being an `int` variable that will be placed in the program memory, will always contain the value 3, and which can never be modified by the program.

The memory model `-mconst-in-data` will allocate `const`-qualified objects in data space, which may be writable.

8.10.2 Volatile Type Qualifier

The `volatile` type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because it may alter the behavior of the program to do so.

Any SFR which can be modified by hardware or which drives hardware is qualified as `volatile`, and any variables which may be modified by interrupt routines should use this qualifier as well. For example:

```
extern volatile unsigned int INTCON1 __attribute__((__sfr__));
```

The code produced by the compiler to access `volatile` objects may be different to that to access ordinary variables, and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. However failure to use this qualifier when it is required may lead to code failure.

Another use of the `volatile` keyword is to prevent variables being removed if they are not used in the C source. If a non-`volatile` variable is never used, or used in a way that has no effect on the program's function, then it may be removed before code is generated by the compiler.

A C statement that consists only of a `volatile` variable's name will produce code that reads the variable's memory location and discards the result. For example the entire statement:

```
PORTB;
```

will produce assembly code that reads `PORTB`, but does nothing with this value. This is useful for some peripheral registers that require reading to reset the state of interrupt flags. Normally such a statement is not encoded as it has no effect.

Some variables are treated as being `volatile` even though they may not be qualified in the source code. See **Chapter 16. “Mixing C and Assembly Code”** if you have assembly code in your project.

8.11 COMPILER-SPECIFIC TYPE QUALIFIERS

The MPLAB XC16 C Compiler supports special type qualifiers, all of which allow the user to control how variables are accessed.

8.11.1 `__psv__` Type Qualifier

The `__psv__` qualifier can be applied to variables or pointer targets that have been allocated to the program memory space. It indicates how the variable or pointer targets will be accessed/read. Allocation of variables to the program memory space is a separate process and is made using the `space` attribute, so this qualifier is often used in conjunction with that attribute when the variable is defined. For example:

```
__psv__ unsigned int __attribute__((space(psv))) myPSVvar = 0x1234;  
__psv__ char * myPSVpointer;
```

The pointer in this example does not use the `space` attribute as it is located in data memory, but the qualifier indicates how the pointer targets are to be accessed. For more information on the `space` attribute and how to allocate variables to the Flash memory, see **Section 8.12 “Variable Attributes”**. For basic information on the memory layout and how program memory is accessed by the device, see **Section 10.2 “Address Spaces”**.

When variables qualified as `__psv__` are read, the compiler will manage the selection of the program memory page visible in the data memory window. This means that you do not need to adjust the PSVPAG SFR explicitly in your source code, but the generated code may be slightly less efficient than that produced if this window was managed by hand.

The compiler will assume that any object or pointer target qualified with `__psv__` will wholly fit within a single PSV page. Such is the case for objects allocated memory using the `psv` or `auto_psv` space attribute. If this is not the case, then you should use the `__prog__` qualifier (see **Section 8.11.2 “`__prog__` Type Qualifier”**) and an appropriate space attribute.

8.11.2 `__prog__` Type Qualifier

The `__prog__` qualifier is similar to the `__psv__` qualifier (see **Section 8.11.1 “`__psv__` Type Qualifier”**), but indicates to the compiler that the qualified variable or pointer target may straddle PSV pages. As a result, the compiler will generate code so these qualified objects can be read correctly, regardless of which page they are allocated to. This code may be longer than that to access variables or pointer targets which are qualified `__psv__`. For example:

```
__prog__ unsigned int __attribute__((space(prog))) myPROGvar = 0x1234;  
__prog__ char * myPROGpointer;
```

The pointer in this example does not use the `space` attribute as it is located in data memory, but the qualifier indicates how the pointer targets are to be accessed. For more information on the `space` attribute and how to allocate variables to the Flash memory, see **Section 8.12 “Variable Attributes”**. And, see **Section 10.2 “Address Spaces”** for basic information on the memory layout and how program memory is accessed by the device.

8.11.3 `__eds__` Type Qualifier

The `__eds__` qualifier indicates that the qualified object has been located in an EDS accessible memory space and that the compiler should manage the appropriate registers used to access this memory.

When used with pointers, it implies that the compiler should make few assumptions as to the memory space in which the pointer target is located and that the target may be in one of several memory spaces, which include: `space(data)` (and its subsets), `eds`, `space(eedata)`, `space(prog)`, `space(psv)`, `space(auto_psv)`, and on some devices `space(pmp)`. Not all devices support all memory spaces. For example

```
__eds__ unsigned int __attribute__((eds)) myEDSvar;
__eds__ char * myEDSpointer;
```

The compiler will automatically assert the `page` attribute to scalar variable declarations; this allows the compiler to generate more efficient code when accessing larger data types. Remember, scalar variables do not include structures or arrays. To force paging of a structure or array, please manually use the `page` attribute and the compiler will prevent the object from crossing a page boundary.

For read access to `__eds__` qualified variables will automatically manipulate the `PSVPAG` or `DSRPAG` register (as appropriate). For devices that support extended data space memory, the compiler will also manipulate the `DSWPAG` register.

Note: Some devices use `DSRPAG` to represent extended read access to FLASH or the extended data space (EDS)

For more on this qualifier, see **Section 10.7 “Extended Data Space Access”**.

8.11.4 `__pack_upper_byte` Type Qualifier

This qualifier allows the use of the upper byte of Flash memory for data storage. For 16-bit devices, a 24-bit word is used in Flash memory. The architecture supports the mapping of areas of Flash into the data space, but this mapping is only 16 bits wide to fit in with data space dimensions, unless the `__pack_upper_byte` qualifier is used.

For more information on this qualifier, see **Section 10.8 “Packing Data Stored in Flash”**.

8.11.5 `__pmp__` Type Qualifier

This qualifier may be used with those devices that contain a Parallel Master Port (PMP) peripheral, which allows the connection of various memory and non-memory devices directly to the device. When variables or pointer targets qualified with `__pmp__` are accessed, the compiler will generate the appropriate sequence for accessing these objects via the PMP peripheral on the device. For example:

```
__pmp__ int auxDevice
__attribute__((space(pmp(external_PMP_memory))));
__pmp__ char * myPMPpointer;
```

In addition to the qualifier, the `int` variable uses a memory space which would need to be predefined. The pointer in this example does not use the `space` attribute as it is located in data memory, but the qualifier indicates how the pointer targets are to be accessed. For more information on the `space` attribute, see **Section 8.12 “Variable Attributes”**. For basic information on the memory layout and how program memory is accessed by the device, see **Section 10.2 “Address Spaces”**.

For more on the qualifier, see **Section 10.5 “Parallel Master Port Access”**.

8.11.6 `__external__` Type Qualifier

This qualifier is used to indicate that the compiler should access variables or pointer targets which have been located in external memory. These memories include any that have been attached to the device, but which are not, or cannot, be accessed using the parallel master port (PMP) peripheral (see **Section 8.11.5 “`__pmp__` Type Qualifier”**.) Access of objects in external memory is similar to that for PMP access, but the routines that do so are fully configurable and, indeed, need to be defined before any access can take place. See **Section 10.6 “External Memory Access”** for full information on how the memory space are configured and access routines are defined.

The qualifier is used as in the following example.

```
__external__ int external_array[256]
               __attribute__((space(external_memory)));
__external__ char * myExternalPointer;
```

In addition to the qualifier, the array uses a memory space which would need to be pre-defined. The pointer in this example does not use the `space` attribute as it is located in data memory, but the qualifier indicates how the pointer targets are to be accessed. For more information on the `space` attribute, see **Section 8.12 “Variable Attributes”**. For basic information on the memory layout and how program memory is accessed by the device, see **Section 10.2 “Address Spaces”**.

For more on the qualifier, see **Section 10.6 “External Memory Access”**.

8.12 VARIABLE ATTRIBUTES

The MPLAB XC16 C Compiler uses attributes to indicate memory allocation, type and other configuration for variables, structure members and types. Other attributes are available for functions, and these are described in **Section 13.2.2 “Function Attributes”**. Qualifiers, listed in **Section 8.11 “Compiler-Specific type Qualifiers”**, are used independently to attributes. They only indicate how objects are accessed, but must be used where necessary to ensure correct code operation.

The compiler keyword `__attribute__` allows you to specify the attributes of objects. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently supported for variables:

- `address (addr)`
- `aligned (alignment)`
- `boot`
- `deprecated`
- `eds`
- `fillupper`
- `far`
- `mode (mode)`
- `near`
- `noload`
- `page`
- `packed`
- `persistent`
- `reverse (alignment)`
- `section ("section-name")`
- `secure`
- `sfr (address)`
- `space (space)`
- `transparent_union`
- `unordered`
- `unsupported(message)`
- `unused`
- `weak`

You may also specify attributes with `__` (double underscore) preceding and following each keyword (e.g., `__aligned__` instead of `aligned`). This allows you to use them in header files without being concerned about a possible macro of the same name.

To specify multiple attributes, separate them by commas within the double parentheses, for example:

```
__attribute__ ((aligned (16), packed)).
```

<p>Note: It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in file A with the <code>far</code> attribute, and declared <code>extern</code> in file B without <code>far</code>, then a link error may result.</p>

Supported Data Types and Variables

address (addr)

The `address` attribute specifies an absolute address for the variable. This attribute can be used in conjunction with a `section` attribute. This can be used to start a group of variables at a specific address:

```
int foo __attribute__((section("mysection"),address(0x900)));
int bar __attribute__((section("mysection")));
int baz __attribute__((section("mysection")));
```

A variable with the `address` attribute cannot be placed into the `auto_psv` space (see the `space()` attribute or the `-mconst-in-code` option); attempts to do so will cause a warning and the compiler will place the variable into the PSV space. If the variable is to be placed into a PSV section, the address should be a program memory address.

aligned (alignment)

This attribute specifies a minimum alignment for the variable, measured in bytes. The alignment must be a power of two. For example, the declaration:

```
int x __attribute__((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On the dsPIC DSC device, this could be used in conjunction with an `asm` expression to access DSP instructions and addressing modes that require aligned operands.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable to the maximum useful alignment for the dsPIC DSC device. For example, you could write:

```
short array[3] __attribute__((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the declared variable to the largest alignment for any data type on the target machine – which in the case of the dsPIC DSC device is two bytes (one word).

The `aligned` attribute can only increase the alignment; you can decrease it by specifying `packed` (see below). The `aligned` attribute conflicts with the `reverse` attribute. It is an error condition to specify both.

The `aligned` attribute can be combined with the `section` attribute. This will allow the alignment to take place in a named section. By default, when no section is specified, the compiler will generate a unique section for the variable. This will provide the linker with the best opportunity for satisfying the alignment restriction without using internal padding that may happen if other definitions appear within the same aligned section.

boot

This attribute can be used to define protected variables in Boot Segment (BS) RAM:

```
int __attribute__((boot)) boot_dat[16];
```

Variables defined in BS RAM will not be initialized on startup. Therefore all variables in BS RAM must be initialized using inline code. A diagnostic will be reported if initial values are specified on a `boot` variable.

An example of initialization is as follows:

```
int __attribute__((boot)) time = 0; /* not supported */
int __attribute__((boot)) time2;
void __attribute__((boot)) foo()
{
    time2 = 55; /* initial value must be assigned explicitly */
}
```

deprecated

The `deprecated` attribute causes the declaration to which it is attached to be specially recognized by the compiler. When a `deprecated` function or variable is used, the compiler will emit a warning.

A `deprecated` definition is still defined and, therefore, present in any object file. For example, compiling the following file:

```
int __attribute__((__deprecated__)) i;
int main() {
    return i;
}
```

will produce the warning:

```
deprecated.c:4: warning: `i' is deprecated (declared
at deprecated.c:1)
```

`i` is still defined in the resulting object file in the normal way.

eds

In the attribute context, the `eds` (extended data space) attribute indicates to the compiler that the variable will be allocated anywhere within data memory. Variables with this attribute will likely also the `__eds__` type qualifier (see **Section 10.7 “Extended Data Space Access”**) for the compiler to properly generate the correct access sequence. Not that the `__eds__` qualifier and the `eds` attribute are closely related, but not identical. On some devices, `eds` may need to be specified when allocating variables into certain memory spaces such as `space (ymemory)` or `space (dma)` as this memory may only exist in the extended data space.

fillupper

This attribute can be used to specify the upper byte of a variable stored into a `space (prog)` section.

For example:

```
int foo[26] __attribute__((space(prog),fillupper(0x23))) = { 0xDEAD };
```

will fill the upper bytes of array `foo` with `0x23`, instead of `0x00`. `foo[0]` will still be initialized to `0xDEAD`.

The command line option `-mfillupper=0x23` will perform the same function.

far

The `far` attribute tells the compiler that the variable will not necessarily be allocated in near (first 8 KB) data space, (i.e., the variable can be located anywhere in data memory between `0x0000` and `0x7FFF`).

mode (mode)

This attribute specifies the data type for the declaration as whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating point type according to its width. Valid values for *mode* are as follows:

Mode	Width	Compiler Type
QI	8 bits	char
HI	16 bits	int
SI	32 bits	long
DI	64 bits	long long
SF	32 bits	float
DF	64 bits	long double

Supported Data Types and Variables

This attribute is useful for writing code that is portable across all supported compiler targets. For example, the following function adds two 32-bit signed integers and returns a 32-bit signed integer result:

```
typedef int __attribute__((__mode__(SI))) int32;
int32
add32(int32 a, int32 b)
{
    return(a+b);
}
```

You may also specify a mode of `byte` or `__byte__` to indicate the mode corresponding to a one-byte integer, `word` or `__word__` for the mode of a one-word integer, and `pointer` or `__pointer__` for the mode used to represent pointers.

near

The `near` attribute tells the compiler that the variable is allocated in near data space (the first 8 KB of data memory). Such variables can sometimes be accessed more efficiently than variables not allocated (or not known to be allocated) in near data space.

```
int num __attribute__((near));
```

noload

The `noload` attribute indicates that space should be allocated for the variable, but that initial values should not be loaded. This attribute could be useful if an application is designed to load a variable into memory at run time, such as from a serial EEPROM.

```
int table1[50] __attribute__((noload)) = { 0 };
```

page

The `page` attribute places variable definitions into a specific page of memory. The page size depends on the type of memory selected by a `space` attribute. Objects residing in RAM will be constrained to a 32K page while objects residing in Flash will be constrained to a 64K page (upper byte not included).

```
unsigned int var[10] __attribute__((space(auto_psv)));
```

The `space(auto_psv)` or `space(psv)` attribute will use a single memory page by default.

```
__eds__ unsigned int var[10] __attribute__((eds, page));
```

When dealing with `eds`, please refer to **Section 10.7 “Extended Data Space Access”** for more information.

packed

The `packed` attribute specifies that a structure member should have the smallest possible alignment unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the member `x` is packed, so that it immediately follows `a`, with no padding for alignment:

```
struct foo
{
    char a;
    int x[2] __attribute__((packed));
};
```

Note: The device architecture requires that words be aligned on even byte boundaries, so care must be taken when using the `packed` attribute to avoid run-time addressing errors.

persistent

The `persistent` attribute specifies that the variable should not be initialized or cleared at startup. A variable with the `persistent` attribute could be used to store state information that will remain valid after a device Reset.

```
int last_mode __attribute__((persistent));
```

Persistent data is not normally initialized by the C run-time. However, from a cold-restart, persistent data may not have any meaningful value. This code example shows how to safely initialize such data:

```
#include <p24Fxxxx.h>

int last_mode __attribute__((persistent));

int main()
{
    if ((RCONbits.POR == 0) &&
        (RCONbits.BOR == 0)) {
        /* last_mode is valid */
    } else {
        /* initialize persistent data */
        last_mode = 0;
    }
}
```

reverse (alignment)

The `reverse` attribute specifies a minimum alignment for the ending address of a variable, plus one. The alignment is specified in bytes and must be a power of two. Reverse-aligned variables can be used for decrementing modulo buffers in dsPIC DSC assembly language. This attribute could be useful if an application defines variables in C that will be accessed from assembly language.

```
int buf1[128] __attribute__((reverse(256)));
```

The `reverse` attribute conflicts with the `aligned` and `section` attributes. An attempt to name a section for a reverse-aligned variable will be ignored with a warning. It is an error condition to specify both `reverse` and `aligned` for the same variable. A variable with the `reverse` attribute cannot be placed into the `auto_psv` space (see the `space()` attribute or the `-mconst-in-code` option); attempts to do so will cause a warning and the compiler will place the variable into the PSV space.

section ("section-name")

By default, the compiler places the objects it generates in sections such as `.data` and `.bss`. The `section` attribute allows you to override this behavior by specifying that a variable (or function) lives in a particular section.

```
struct a { int i[32]; };
struct a buf __attribute__((section("userdata"))) = {{0}};
```

Supported Data Types and Variables

secure

This attribute can be used to define protected variables in Secure Segment (SS) RAM:

```
int __attribute__((secure)) secure_dat[16];
```

Variables defined in SS RAM will not be initialized on startup. Therefore all variables in SS RAM must be initialized using inline code. A diagnostic will be reported if initial values are specified on a `secure` variable.

String literals can be assigned to secure variables using inline code, but they require extra processing by the compiler. For example:

```
char *msg __attribute__((secure)) = "Hello!\n"; /* not supported */
char *msg2 __attribute__((secure));
void __attribute__((secure)) foo2()
{
    *msg2 = "Goodbye..\n"; /* value assigned explicitly */
}
```

In this case, storage must be allocated for the string literal in a memory space which is accessible to the enclosing secure function. The compiler will allocate the string in a psv constant section designated for the secure segment.

sfr (address)

The `sfr` attribute tells the compiler that the variable is an SFR and may also specify the run-time address of the variable, using the `address` parameter.

```
extern volatile int __attribute__((sfr(0x200))) ulmod;
```

The use of the extern specifier is required in order to not produce an error.

Note: By convention, the `sfr` attribute is used only in processor header files. To define a general user variable at a specific address use the `address` attribute in conjunction with `near` or `far` to specify the correct addressing mode.

space (space)

Normally, the compiler allocates variables in general data space. The `space` attribute can be used to direct the compiler to allocate a variable in specific memory spaces. Memory spaces are discussed further in **Section 10.2 “Address Spaces”**. The following arguments to the space attribute are accepted:

data

Allocate the variable in general data space. Variables in general data space can be accessed using ordinary C statements. This is the default allocation.

DD

xmemory - dsPIC30F, dsPIC33EP/F DSCs only

Allocate the variable in X data space. Variables in X data space can be accessed using ordinary C statements. An example of `xmemory` space allocation is:

```
int x[32] __attribute__((space(xmemory)));
```

DD

ymemory - dsPIC30F, dsPIC33EP/F DSCs only

Allocate the variable in Y data space. Variables in Y data space can be accessed using ordinary C statements. An example of `ymemory` space allocation is:

```
int y[32] __attribute__((space(ymemory)));
```

prog

Allocate the variable in program space, in a section designated for executable code. Variables in program space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, the program space visibility window, or by the methods described in **Section 10.4.2 “Access of objects in Program Memory”**.

auto_psv

Allocate the variable in program space, in a compiler-managed section designated for automatic program space visibility window access. Variables in `auto_psv` space can be read (but not written) using ordinary C statements, and are subject to a maximum of 32K total space allocated. When specifying `space(auto_psv)`, it is not possible to assign a section name using the `section` attribute; any section name will be ignored with a warning. A variable in the `auto_psv` space cannot be placed at a specific address or given a reverse alignment.

Note: Variables placed in the `auto_psv` section are not loaded into data memory at startup. This attribute may be useful for reducing RAM usage.

DD

dma - PIC24E/H MCUs, dsPIC33E/F DSCs only

Allocate the variable in DMA memory. Variables in DMA memory can be accessed using ordinary C statements and by the DMA peripheral.

`__builtin_dmaoffset()` and `__builtin_dmapage()` can be used to find the correct offset for configuring the DMA peripheral. See **Appendix G. “Built-in Functions”** for details.

```
#include <p24Hxxxx.h>
unsigned int BufferA[8] __attribute__((space(dma)));
unsigned int BufferB[8] __attribute__((space(dma)));

int main()
{
    DMA1STA = __builtin_dmaoffset(BufferA);
    DMA1STB = __builtin_dmaoffset(BufferB);
    /* ... */
}
```

psv

Allocate the variable in program space, in a section designated for program space visibility window access. The linker will locate the section so that the entire variable can be accessed using a single setting of the PSVPAG register. Variables in PSV space are not managed by the compiler and can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

DD

eedata - PIC24F, dsPIC30F/33F DSCs only

Allocate the variable in EEData space. Variables in EEData space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

pmp

Allocate the variable in off chip memory associated with the PMP peripheral. For complete details please see **Section 10.5 “Parallel Master Port Access”**.

Supported Data Types and Variables

external

Allocate the variable in a user defined memory space. For complete details please see **Section 10.6 “External Memory Access”**.

transparent_union

This attribute, attached to a function parameter which is a `union`, means that the corresponding argument may have the type of any union member, but the argument is passed as if its type were that of the first union member. The argument is passed to the function using the calling conventions of the first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

unordered

The `unordered` attribute indicates that the placement of this variable may move relative to other variables within the current C source file.

```
const int __attribute__((unordered)) i;
```

unsupported(message)

This attribute will display a custom message when the object is used.

```
int foo __attribute__((unsupported("This object is unsupported")));
```

Access to `foo` will generate a warning message.

unused

This attribute, attached to a variable, means that the variable is meant to be possibly unused. The compiler will not produce an unused variable warning for this variable.

weak

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol may be superseded by a global definition. When `weak` is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((__weak__)) s;
int foo() {
    if (&s) return s;
    return 0; /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that `s` has been defined (and returns its value if it has). Otherwise '0' is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

The `weak` attribute may be applied to functions as well as variables:

```
extern int __attribute__((__weak__)) compress_data(void *buf);
int process(void *buf) {
    if (compress_data) {
        if (compress_data(buf) == -1) /* error */
        }
    /* process buf */
}
```

In the above code, the function `compress_data` will be used only if it is linked in from some other module. Deciding whether or not to use the feature becomes a link-time decision, not a compile time decision.

The affect of the `weak` attribute on a definition is more complicated and requires multiple files to describe:

```
/* weak1.c */
int __attribute__((__weak__)) i;

void foo() {
    i = 1;
}

/* weak2.c */
int i;
extern void foo(void);

void bar() {
    i = 2;
}

main() {
    foo();
    bar();
}
```

Here the definition in `weak2.c` of `i` causes the symbol to become a strong definition. No link error is emitted and both `i`'s refer to the same storage location. Storage is allocated for `weak1.c`'s version of `i`, but this space is not accessible.

There is no check to ensure that both versions of `i` have the same type; changing `i` in `weak2.c` to be of type `float` will still allow a link, but the behavior of function `foo` will be unexpected. `foo` will write a value into the least significant portion of our 32-bit float value. Conversely, changing the type of the weak definition of `i` in `weak1.c` to type `float` may cause disastrous results. We will be writing a 32-bit floating point value into a 16-bit integer allocation, overwriting any variable stored immediately after our `i`.

In the cases where only `weak` definitions exist, the linker will choose the storage of the first such definition. The remaining definitions become in-accessible.

The behavior is identical, regardless of the type of the symbol; functions and variables behave in the same manner.

Chapter 9. Fixed-Point Arithmetic Support

9.1 INTRODUCTION

The MPLAB XC16 C compiler supports fixed-point arithmetic according to the N1169 draft of ISO/IEC TR 18037, the ISO C99 technical report on Embedded C, available here:

<http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037>

This appendix describes the implementation-specific details of the types and operations supported by the compiler under this draft standard.

- Enabling Fixed-Point Arithmetic Support
- Data Types
- Rounding
- External Definitions
- Mixing C and Assembly Language Code

9.2 ENABLING FIXED-POINT ARITHMETIC SUPPORT

Fixed-point arithmetic support is not enabled by default in the MPLAB XC16 C compiler; it must be explicitly enabled by the `-menable-fixed` compiler switch, described in **Section 5.7 “Driver Option Descriptions”**.

9.3 DATA TYPES

All 12 of the primary fixed-point types and their aliases, described in section 4.1 “Overview and principles of the fixed-point data types” of N1169, are supported via three fixed point formats corresponding to the intrinsic hardware capabilities of Microchip 16-bit devices.

TABLE 9-1: FIXED POINT FORMATS - 16-BIT DEVICES

Format	Description
1.15	1 bit sign, 15 bits fraction
1.31	1 bit sign, 31 bits fraction
9.31	9 bit signed integer, 31 bits fraction

These formats represent the fixed-point C data types, shown below.

TABLE 9-2: FIXED POINT FORMATS - C DATA TYPES

Type	Format
<code>_Fract</code>	1.15
<code>short _Fract</code>	1.15
<code>signed _Fract</code>	1.15
<code>signed short _Fract</code>	1.15
<code>unsigned _Fract</code>	1.15 (sign bit 0)
<code>unsigned short _Fract</code>	1.15 (sign bit 0)
<code>long _Fract</code>	1.31

TABLE 9-2: FIXED POINT FORMATS - C DATA TYPES (CONTINUED)

Type	Format
signed long _Fract	1.31
unsigned long _Fract	1.31 (sign bit 0)
_Accum	9.31
short _Accum	9.31
long _Accum	9.31
signed _Accum	9.31
signed short _Accum	9.31
signed long _Accum	9.31
unsigned _Accum	9.31 (sign bit 0)
unsigned short _Accum	9.31 (sign bit 0)
unsigned long _Accum	9.31 (sign bit 0)

The `_Sat` type specifier, indicating that the values are saturated, may be used with any type as described in N1169.

Unsigned types are represented identically to signed types, but negative numbers (sign bit 1) are not valid values in the unsigned types. Signed types saturate at the most negative and positive numbers representable in the underlying format. Unsigned types saturate at 0 and the most positive number representable in the format.

The default behavior of overflow on signed or unsigned types is not saturation (as defined by the pragmas described in section 4.1.3 “Rounding and Overflow” of N1169). Therefor variables in signed or unsigned types that are not declared as saturating with the `_Sat` specifier may receive invalid values when assigned the result of an expression in which an overflow may occur (the results of non-saturating overflows are not defined.)

9.4 ROUNDING

Three rounding modes are supported, corresponding to the three rounding modes supported by the 16-bit device fixed-point multiplication facilities.

TABLE 9-3: ROUNDING MODES

Mode	Description
Truncation	Truncate signed result - round toward -saturation
Conventional	Round signed result to nearest, ties toward +saturation
Convergent	Round signed result to nearest, ties to even

All operations on fixed point variables, whether intrinsically supported by the hardware or not, are performed according to the prevailing rounding mode chosen. The rounding mode may be specified globally via the `-menable-fixed` compiler switch, as described in **Section 5.7 “Driver Option Descriptions”**, or on a function-by-function basis, via the `-round` attribute, as described in **Section 13.2.2 “Function Attributes”**

These modes are described in more detail in the “16-bit MCU and DSC Programmer’s Reference Manual” (DS70157).

9.5 DIVISION BY ZERO

The result of a division of a `_Fract` or `_Accum` typed value by zero is not defined, and may or may not result in an arithmetic error trap. Regardless of the presence of the `_Sat` keyword, division by zero does NOT produce the most negative or most positive saturation value for the result type.

9.6 EXTERNAL DEFINITIONS

The MPLAB XC16 C compiler provides an include file, `stdfix.h`, which provides constant, pragma, typedef, and function definitions as described in section 7.18a of N1169.

Fixed point conversion specifiers for formatted I/O, as described in section 4.1.9 “Formatted I/O functions for fixed-point arguments” of N1169, are not supported in the current MPLAB XC16 standard C libraries. Fixed-point variables may be displayed via `(s)printf` by casting them to the appropriate floating point representation (`double` for `_Fract`, `long double` for `long _Fract` and `_Accum`), and then displaying the value in that format. To scan a fixed-point number via `(s)scanf`, first scan it as the appropriate `double` or `long double` floating point number, and then cast the value obtained to the desired fixed-point type.

The fixed point functions described in section 4.1.7 of N1169 are not provided in the current MPLAB XC16 standard C libraries.

Fixed point constants, with suffixes of `k` (K) and `r` (R), as described in section 4.1.5 of N1169, are supported by the MPLAB XC16 C compiler.

9.7 MIXING C AND ASSEMBLY LANGUAGE CODE

The MPLAB XC16 C compiler generates fixed-point code that assumes that certain 16-bit device resources are managed by the compiler's start-up and run-time code. Hand-written assembly code built into the same program could interfere with the state of the CPU assumed by the code the compiler generates.

MPLAB XC16 programs may contain both fixed-point C and assembly language code that utilizes 16-bit device intrinsic fixed-point capabilities directly, but in order for these two kinds of code to inter-operate safely, the compiler must save certain dsPIC registers around calls to assembly language functions that may change their state. The C compiler can be instructed to do so by providing prototypes for assembly language functions for which this is necessary. These prototypes should specify the `save(CORCON)` attribute for the target assembly language function, as described in **Section 13.2.2 “Function Attributes”**. Programs constructed in this manner will operate correctly, at the expense of some state saves and restores around calls to the indicated assembly routines.

NOTES:

Chapter 10. Memory Allocation and Access

10.1 INTRODUCTION

There are two broad groups of RAM-based variables: auto/parameter variables, which are allocated to some form of stack, and global/static variables, which are positioned freely throughout the data memory space. The memory allocation of these two groups is discussed separately in the following sections.

- Address Spaces
- Variables in Data Space Memory
- Variables in Program Space
- Parallel Master Port Access
- External Memory Access
- Extended Data Space Access
- Packing Data Stored in Flash
- Allocation of Variables to Registers
- Variables in EEPROM
- Dynamic Memory Allocation
- Memory Models

10.2 ADDRESS SPACES

The 16-bit devices are a combination of traditional PIC[®] Microcontroller (MCU) features (peripherals, Harvard architecture, RISC) and new DSP capabilities (dsPIC DSC devices). These devices have two distinct memory regions:

- Program Memory contains executable code and optionally constant data.
- Data Memory contains external variables, static variables, the system stack and file registers. Data memory consists of near data, which is memory in the first 8 KB of the data memory space, and far data, which is in the upper 56 KB of data memory space.

Although the program and data memory regions are distinctly separate, the dsPIC30F/33F and PIC24F/H families of processors contain hardware support for accessing data from within program Flash using a hardware feature that is commonly called Program Space Visibility (PSV). More detail about how PSV works can be found in device data sheets or family reference manuals. Also, see **Section 10.3 “Variables in Data Space Memory”** and **Section 14.8.2 “PSV Usage with Interrupt Service Routines”**.

Briefly, the architecture allows the mapping of one 32K page of Flash into the upper 32K of the data address space via the Special Function Register (SFR) PSVPAG. Devices that support Extended Data Space (EDS) map using the DSRPAG register instead. Also it is possible to map FLASH and other areas. See **Section 10.7 “Extended Data Space Access”** for more details.

By default the compiler only supports direct access to one single PSV page, referred to as the `auto_psv` space. In this model, 16-bit data pointers can be used. However, on larger devices, this can make it difficult to manage large amounts of constant data stored in Flash.

The extensions presented here allow the definition of a variable as being a 'managed' PSV variable. This means that the compiler will manipulate both the offset (within a PSV page) and the page itself. As a consequence, data pointers must be 32 bits. The compiler will probably generate more instructions than the single PSV page model, but that is the price being paid to buy more flexibility and shorter coding time to access larger amounts of data in Flash.

10.3 VARIABLES IN DATA SPACE MEMORY

Most variables are ultimately positioned into the data space memory. The exceptions are non-`auto` variables which are qualified as `const` and may be placed in the program memory space.

Due to the fundamentally different way in which `auto` variables and non-`auto` variables are allocated memory, they are discussed separately. To use the C language terminology, these two groups of variables are those with 'automatic storage duration' and those with 'permanent storage duration', respectively.

The terms "local" and "global" are commonly used to describe variables, but are not ones defined by the language standard. The term "local variable" is often taken to mean a variable which has scope inside a function, and "global variable" is one which has scope throughout the entire program. However, the C language has three common scopes: block, file (i.e. internal linkage) and program (i.e. external linkage). So using only two terms to describe these can be confusing.

For example, a `static` variable defined outside a function has scope only in that file, so it is not globally accessible, but it can be accessed by more than one function inside that file, so it is not local to any one function either.

In terms of memory allocation, variables are allocated space based on whether it is an `auto` or not; hence the grouping in the following sections.

10.3.1 Non-Auto Variable Allocation and Access

Non-`auto` (`static` and `external`) variables have permanent storage duration and are located by the compiler into the data space memory. The compiler will also allocate non-`auto` `const`-qualified variables (see **Section 8.10.1 "Const Type Qualifier"**) into the data space memory if the constants-in-data memory model is selected; otherwise, they are located in program memory.

10.3.1.1 DEFAULT ALLOCATION OF NON-AUTO VARIABLES

The compiler considers several categories of `static` and `external` variable, which all relate to the value the variable should contain at the time the program begins, that is, those that should be cleared at program startup (uninitialized variables), and those that should hold a non-zero value (initialized variables), and those that should not be altered at all at program startup (persistent variables). Those objects qualified as `const` are usually assigned an initial value since they are read-only. If they are not assigned an initial value, they are grouped with the other uninitialized variables.

Data placed in RAM may be initialized at startup by copying initialized values from program memory.

10.3.1.2 STATIC VARIABLES

All `static` variables have permanent storage duration, even those defined inside a function which are “local static” variables. Local `static` variables only have scope in the function or block in which they are defined, but unlike `auto` variables, their memory is reserved for the entire duration of the program. Thus they are allocated memory like other non-`auto` variables. Static variables may be accessed by other functions via pointers since they have permanent duration.

Variables which are `static` are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer.

Variables which are `static` and which are initialized only have their initial value assigned once during the program's execution. Thus, they may be preferable over initialized `auto` objects which are assigned a value every time the block in they are defined begins execution. Any initialized static variables are initialized in the same way as other non-`auto` initialized objects by the runtime startup code, see **Section 5.4.2 “Startup and Initialization”**.

10.3.1.3 NON-AUTO VARIABLE SIZE LIMITS

The compiler option `-mlarge-arrays` allows you to define and access arrays larger than 32K. You must ensure that there is enough space to allocate such an array by nominating a memory space large enough to contain such an object.

Using this option will have some effect on how code is generated as it effects the definition of the `size_t` type, increasing it to an `unsigned long int`. If used as a global option, this will affect many operations used in indexing (making the operation more complex). Using this option locally may effect how variables can be accessed. With these considerations in mind, using large arrays is requires careful planning. This section discusses some techniques for its use.

Two things occur when the `-mlarge-arrays` option is selected:

1. The compiler generates code in a different way for accessing arrays.
2. The compiler defines the `size_t` type to be `unsigned long int`.

Item 1 can have a negative effect on code size, if used throughout the whole program. It is possible to only compile a single module with this option and have it work, but there are limitations which will be discussed shortly.

Item 2 affects the calling convention when external functions receive or return objects of type `size_t`. The compiler provides libraries built to handle a larger `size_t` and these objects will be selected automatically by the linker (provided they exist).

Mixing `-mlarge-arrays` and normal-sized arrays together is relatively straightforward and might be the best way to make use of this feature. There are a few usage restrictions: functions defined in such a module should not call external routines that use `size_t`, and functions defined in such a module should not receive `size_t` as a parameter.

For example, one could define a large array and an accessor function which is then used by other code modules to access the array. The benefit is that only one module needs to be compiled with `-mlarge-array` with the defect that an accessor is required to access the array. This is useful in cases where compiling the whole program with `-mlarge-arrays` will have negative effect on code size and speed.

A code example for this would be:

file1.c

```
/* to be compiled -mlarge-arrays */
__prog__ int array1[48000] __attribute__((space(prog)));
__prog__ int array2[48000] __attribute__((space(prog)));

int access_large_array(__prog__ int *array, unsigned long index) {
    return array[index];
}
```

file2.c

```
/* to be compiled without -mlarge-arrays */
extern __prog__ int array1[] __attribute__((space(prog)));
extern __prog__ int array2[] __attribute__((space(prog)));

extern int access_large_array(__prog__ int *array,
    unsigned long index);

main() {
    fprintf(stderr, "Answer is: %d\n", access_large_array(array1,
        39543));
    fprintf(stderr, "Answer is: %d\n", access_large_array(array2,
        16));
}
```

10.3.1.4 CHANGING NON-AUTO VARIABLE ALLOCATION

As described in **Section 10.2 “Address Spaces”**, the compiler arranges for data to be placed into sections, depending on the memory model used and whether or not the data is initialized. When modules are combined at link time, the linker determines the starting addresses of the various sections based on their attributes.

Cases may arise when a specific variable must be located at a specific address, or within some range of addresses. The easiest way to accomplish this is by using the `address` attribute, described in **Chapter 7. “Differences Between MPLAB XC16 and ANSI C”**. For example, to locate variable `Mabonga` at address `0x1000` in data memory:

```
int __attribute__((address(0x1000))) Mabonga = 1;
```

A group of common variables may be allocated into a named section, complete with address specifications:

```
int __attribute__((section("mysection"), address(0x1234))) foo;
```

10.3.1.5 DATA MEMORY ALLOCATION MACROS

Macros that may be used to allocate space in data memory are discussed below. There are two types: those that require an argument and those that do not.

The following macros require an argument `N` that specifies alignment. `N` must be a power of two, with a minimum value of 2.

```
#define _XBSS(N)    __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N)  __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N)    __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N)  __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata), aligned(N)))
```

For example, to declare an uninitialized array in X memory that is aligned to a 32-byte address:

```
int _XBSS(32) xbuf[16];
```

To declare an initialized array in data EEPROM without special alignment:

```
int _EEDATA(2) table1[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

The following macros do not require an argument. They can be used to locate a variable in persistent data memory or in near data memory.

```
#define _PERSISTENT __attribute__((persistent))
#define _NEAR      __attribute__((near))
```

For example, to declare two variables that retain their values across a device Reset:

```
int _PERSISTENT var1, var2;
```

10.3.2 Auto Variable Allocation and Access

This section discusses allocation of `auto` variables (those with automatic storage duration). This also includes function parameter variables, which behave like `auto` variables, as well as temporary variables defined by the compiler.

The `auto` (short for *automatic*) variables are the default type of local variable. Unless explicitly declared to be `static`, a local variable will be made `auto`. The `auto` keyword may be used if desired.

`auto` variables, as their name suggests, automatically come into existence when a block is executed and then disappear once the block exits. Since they are not in existence for the entire duration of the program, there is the possibility to reclaim memory they use when the variables are not in existence and allocate it to other variables in the program.

Typically such variables are stored on some sort of a data stack, which can easily allocate then deallocate memory as required by each function. The stack is discussed in **Section 10.3.2.1 “Software Stack”**.

The standard qualifiers: `const` and `volatile` may both be used with `auto` variables and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an `auto` object and, as such, will be allocated memory on the stack, not in the program memory like with non-`auto` `const` objects.

10.3.2.1 SOFTWARE STACK

The dsPIC DSC device dedicates register W15 for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. The stack grows upward, towards higher memory addresses.

The dsPIC DSC device also supports stack overflow detection. If the Stack Pointer Limit register, SPLIM, is initialized, the device will test for overflow on all stack operations. If an overflow should occur, the processor will initiate a stack error exception. By default, this will result in a processor Reset. Applications may also install a stack error exception handler by defining an interrupt function named `_StackError`. See **Chapter 14. “Interrupts”** for details.

The C run-time startup module initializes the Stack Pointer (W15) and the Stack Pointer Limit register during the startup and initialization sequence. The initial values are normally provided by the linker, which allocates the largest stack possible from unused data memory. The location of the stack is reported in the link map output file. Applications can ensure that at least a minimum-sized stack is available with the `--stack` linker command-line option. See the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for details.

Alternatively, a stack of specific size may be allocated with a user-defined section from an assembly source file. In the following example, 0x100 bytes of data memory are reserved for the stack:

```
.section *,data,stack  
.space 0x100
```

The linker will allocate an appropriately sized section and initialize `__SP_init` and `__SPLIM_init` so that the run-time startup code can properly initialize the stack. Note that since this is a normal assembly code, section attributes such as `address` may be used to further define the stack. Please see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for more information.

10.3.2.2 THE C STACK USAGE

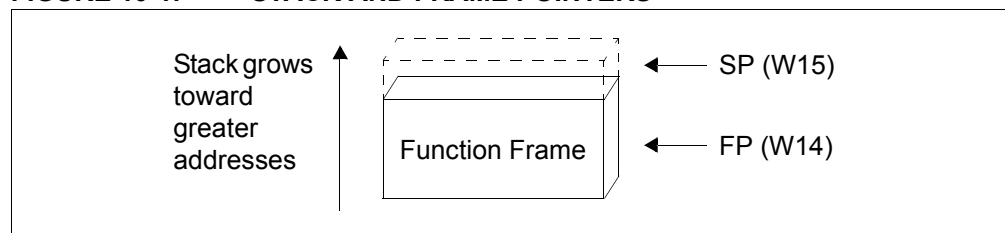
The C compiler uses the software stack to:

- Allocate automatic variables
- Pass arguments to functions
- Save the processor status in interrupt functions
- Save function return address
- Store temporary results
- Save registers across function calls

The run-time stack grows upward from lower addresses to higher addresses. The compiler uses two working registers to manage the stack:

- W15 – This is the Stack Pointer (SP). It points to the top of stack which is defined to be the first unused location on the stack.
- W14 – This is the Frame Pointer (FP). It points to the current function's frame. Each function, if required, creates a new frame at the top of the stack from which automatic and temporary variables are allocated. The compiler option `-fomit-frame-pointer` can be used to restrict the use of the FP.

FIGURE 10-1: STACK AND FRAME POINTERS

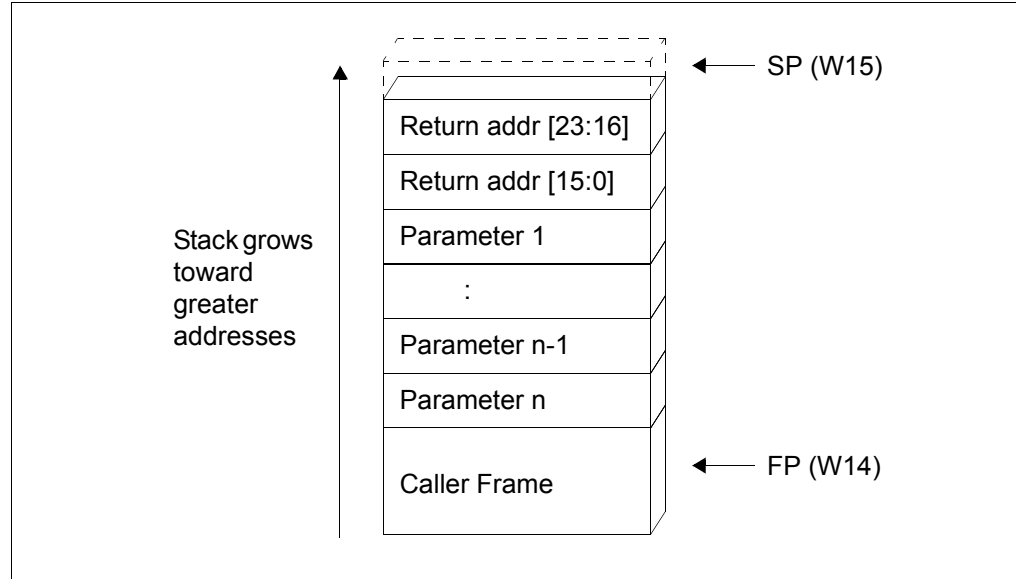


The C run-time startup modules in `libpic30-omf.a` initialize the Stack Pointer W15 to point to the bottom of the stack and initialize the Stack Pointer Limit register to point to the top of the stack. The stack grows up and if it should grow beyond the value in the Stack Pointer Limit register, then a stack error trap will be taken. The user may initialize the Stack Pointer Limit register to further restrict stack growth.

The following diagrams illustrate the steps involved in calling a function. Executing a `CALL` or `RCALL` instruction pushes the return address onto the software stack. See Figure 10-2.

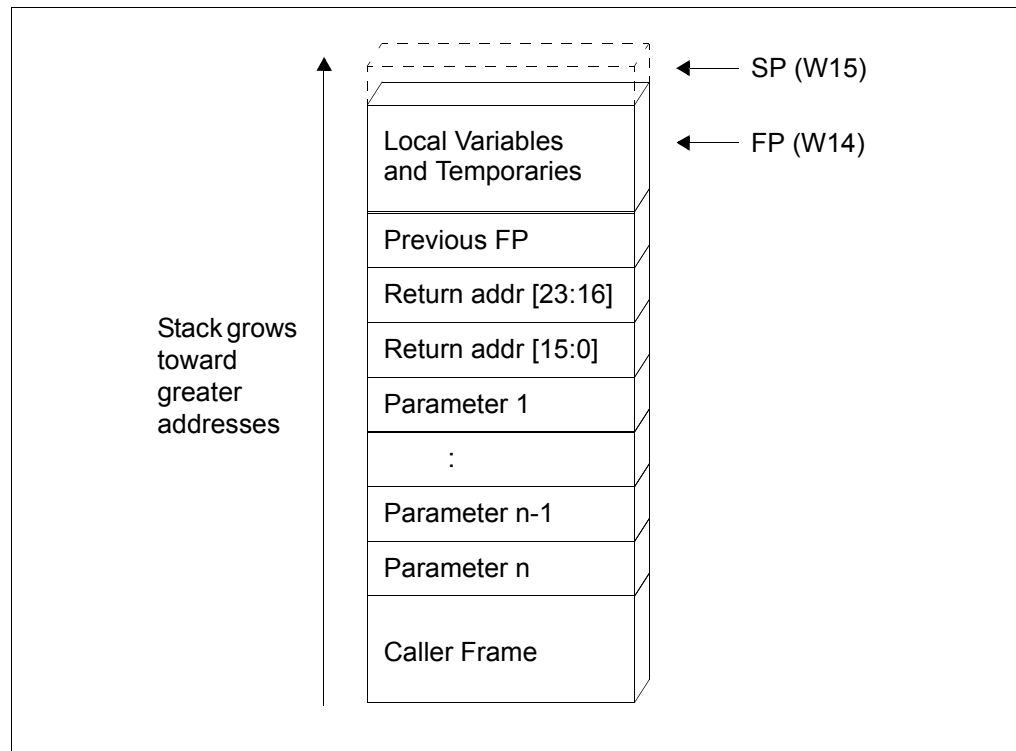
Memory Allocation and Access

FIGURE 10-2: CALL OR RCALL



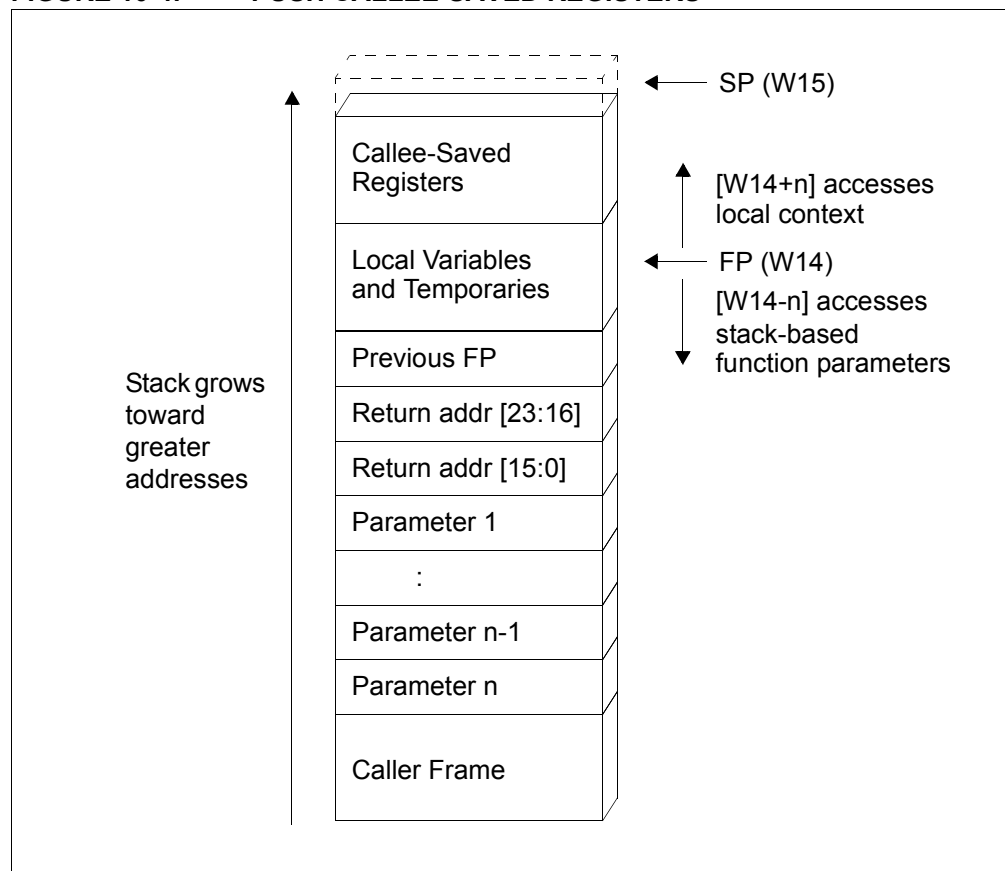
The called function (callee) can now allocate space for its local context (Figure 10-3).

FIGURE 10-3: CALLEE SPACE ALLOCATION



Finally, any callee-saved registers that are used in the function are pushed (Figure 10-4).

FIGURE 10-4: PUSH CALLEE-MAINTAINED REGISTERS



10.3.2.3 AUTO VARIABLE SIZE LIMITS

If a program requires large objects that should not be accessible to the entire program, consider leaving them as local objects, but using the `static` specifier. Such variables are still local to a function, but are no longer `auto` and are allocated permanent storage which is not in the software stack.

The `auto` objects are subject to the similar constraints as non-`auto` objects in terms of maximum size, but they are allocated to the software stack rather than fixed memory locations. **Section 10.3.1.3 “Non-Auto Variable Size Limits”** which describes defining and using large arrays is also applicable to `auto` objects.

10.3.3 Changing Auto Variable Allocation

As `auto` variables are dynamically allocated space in the software stack, using the `address` attribute or other mechanisms to have them allocated at a non-default location is not permitted.

10.4 VARIABLES IN PROGRAM SPACE

The 16-bit core families of processors contain hardware support for accessing data from within program Flash using a hardware feature that is commonly called Program Space Visibility (PSV). More detail about how PSV works can be found in device data sheets or family reference manuals. Also, see **Section 10.4.1 “Allocation and Access of Program Memory Objects”** and **Section 14.8.2 “PSV Usage with Interrupt Service Routines”**.

Briefly, the architecture allows the mapping of one 32K page of Flash into the upper 32K of the data address space via the Special Function Register (SFR) PSVPAG or DSRPAG. By default the compiler only supports direct access to one single PSV page, referred to as the `auto_psv` space. In this model, 16-bit data pointers can be used. However, this can make it difficult to manage large amounts of constant data stored in Flash on larger devices.

When the option `-mconst-in-code` is enabled, `const`-qualified variables that are not `auto` are placed in program memory. Any `auto` variables qualified `const` are placed on the stack along with other `auto` variables.

Any `const`-qualified (`auto` or non-`auto`) variable will always be read-only and any attempt to write to these in your source code will result in an error being issued by the compiler.

A `const` object is usually defined with initial values, as the program cannot write to these objects at runtime. However this is not a requirement. An uninitialized `const` object is allocated space along with other uninitialized RAM variables, but is still read-only. Here are examples of `const` object definitions.

```
const char IObtype = 'A'; // initialized const object
const char buffer[10];    // I reserve memory in RAM
```

See **Chapter 16. “Mixing C and Assembly Code”** for the equivalent assembly symbols that are used to represent `const`-qualified variables in program memory.

10.4.1 Allocation and Access of Program Memory Objects

There are many objects that are allocated to program memory by the compiler. The following sections indicate those objects and how they are allocated to their final memory location by the compiler and how they are accessed.

10.4.1.1 STRING AND CONST OBJECTS

By default, the compiler will automatically arrange for strings and `const`-qualified initialized variables to be allocated in the `auto_psv` section, which is mapped into the PSV window. Specify the `-mconst-in-data` option to direct the compiler not to use the PSV window and these objects will be allocated along with other RAM-based variables.

In the default memory model, the PSV page is fixed to one page which is represented by the `auto_psv` memory space. Accessing the single `auto PSV` page is efficient as no page manipulation is required. Additional FLASH may be accessed using the techniques introduced in section **Section 10.4.2.1 “Managed PSV Access”**.

10.4.1.2 CONST-QUALIFIED VARIABLES IN SECURE FLASH

`const`-qualified variables with initializers can be supported in secure Flash segments using PSV constant sections managed by the compiler. For example:

```
const int __attribute__((boot)) time_delay = 55;
```

If the `const` qualifier was omitted from the definition of `time_delay`, this statement would be rejected with an error message. (Initialized variables in secure RAM are not supported).

Since the `const` qualifier has been specified, variable `time_delay` can be allocated in a PSV constant section that is owned by the boot segment. It is also possible to specify the PSV constant section explicitly with the `space(auto_psv)` attribute:

```
int __attribute__((boot, space(auto_psv))) bebop = 20;
```

Pointer variables initialized with string literals require special processing. For example:

```
char * const foo __attribute__((boot)) = "eek";
```

The compiler will recognize that string literal "eek" must be allocated in the same PSV constant section as pointer variable `foo`.

Regardless of whether you have selected the constants-in-code or constants-in-data memory model, the compiler will create and manage PSV constant sections as needed for secure segments. Support for user-managed PSV sections is maintained through an object compatibility model explained below.

Upon entrance to a boot or secure function, `PSVPAG` will be set to the correct value. This value will be restored after any external function call.

10.4.1.3 STRING LITERALS AS ARGUMENTS

In addition to being used as initializers, string literals may also be used as function arguments. For example:

```
myputs("Enter the Dragon code:\n");
```

Here allocation of the string literal depends on the surrounding code. If the statement appears in a boot or secure function, the literal will be allocated in a corresponding PSV constant section. Otherwise it will be placed in general (non-secure) memory, according to the constants memory model.

Recall that data stored in a secure segment cannot be read by any other segment. For example, it is not possible to call the standard C library function `puts()` with a string that has been allocated in a secure segment. Therefore literals which appear as function arguments can only be passed to functions in the same security segment. This is also true for objects referenced by pointers and arrays. Simple scalar types such as `char`, `int`, and `float`, which are passed by value, may be passed to functions in different segments.

10.4.2 Access of objects in Program Memory

Allocating objects to program memory and accessing them are considered as two separate issues. The compiler requires that you qualify variables to indicate how they are accessed. You can choose to have the compiler manage access of these objects, or do this yourself, which can be more efficient, but more complex.

10.4.2.1 MANAGED PSV ACCESS

The compiler introduces several new qualifiers (CV-qualifiers for the language lawyers in the audience). Like a `const volatile` qualifier, the new qualifiers can be applied to objects or pointer targets. These qualifiers are:

- `__psv__` for accessing objects that do not cross a PSV boundary, such as those allocated in `space(auto_psv)` or `space(psv)`
- `__prog__` for accessing objects that may cross a PSV boundary, specifically those allocated in `space(prog)`, but it may be applied to any object in Flash
- `__eds__` for accessing objects that may be in FLASH or the extended data space (for devices with > 32K of RAM), see `__eds__` in **Section 10.7 “Extended Data Space Access”**.

Typically there is no need to specify `__psv__` or `__prog__` for an object placed in `space(auto_psv)`.

Defining a variable in a compiler managed Flash space is accomplished by:

```
__psv__ unsigned int FLASH_variable __attribute__((space(psv)));
```

Reading from the variable now will cause the compiler to generate code that adjusts the appropriate PSV page SFR as necessary to access the variable correctly. These qualifiers can equally decorate pointers:

```
__psv__ unsigned int *pFLASH;
```

produces a pointer to something in PSV, which can be assigned to a managed PSV object in the normal way. For example:

```
pFLASH = &FLASH_variable;
```

10.4.2.2 OBJECT COMPATIBILITY MODEL

Since functions in secure segments set PSVPAG to their respective `psv` constant sections, a convention must be established for managing multiple values of the PSVPAG register. In previous versions of the compiler, a single value of PSVPAG was set during program startup if the default `constants-in-code` memory model was selected. The compiler relied upon that preset value for accessing `const` variables and string literals, as well as any variables specifically nominated with `space(auto_psv)`.

MPLAB XC16 provides support for multiple values of PSVPAG. Variables declared with `space(auto_psv)` may be combined with secure segment constant variables and/or managed `psv` variables in the same source file. Precompiled objects that assume a single, pre-set value of PSVPAG are link-compatible with objects that define secure segment `psv` constants or managed `psv` variables.

Even though PSVPAG is considered to be a compiler-managed resource, there is no change to the function calling conventions.

10.4.2.3 ISR CONSIDERATIONS

A data access using managed PSV pointers is definitely not atomic, meaning it can take several instructions to complete the access. Care should be taken if an access should not be interrupted.

Furthermore an Interrupt Service Routine (ISR) never really knows what the current state of the PSVPAG register will be. Unfortunately the compiler is not really in any position to determine whether or not this is important in all cases.

The compiler will make the simplifying assumption that the writer of the interrupt service routine will know whether or not the automatic, compiler managed PSVPAG is required by the ISR. This is required to access any constant data in the `auto_psv` space or any string literals or constants when the default `-mconst-in-code` option is selected. When defining an interrupt service routine, it is best to specify whether or not it is necessary to assert the default setting of the PSVPAG SFR.

This is achieved by adding a further attribute to the interrupt function definition:

- `auto_psv` - the compiler will set the PSVPAG register to the correct value for accessing the `auto_psv` space, ensuring that it is restored when exiting the ISR
- `no_auto_psv` - the compiler will not set the PSVPAG register

For example:

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {  
    IFS0bits.T1IF = 0;  
}
```

The choice is provided so that, if you are especially conscious of interrupt latency, you may select the best option. Saving and setting the PSVPAG will consume approximately 3 cycles at the entry to the function and one further cycle to restore the setting upon exit from the function.

Note that `boot` or `secure` interrupt service routines will use a different setting of the PSVPAG register for their constant data.

10.4.3 Size Limitations of Program Memory Variables

Arrays of any type (including arrays of aggregate types) can be qualified `const` and placed in the program memory. So too can structure and union aggregate types, see **Section 8.6 “Structures and Unions”**. These objects can often become large in size and may affect memory allocation.

For objects allocated in a compiler-managed PSV window (`auto_psv` space) the total memory available for allocation is limited by the size of the PSV window itself. Thus no single object can be larger than the size of the PSV window, and all such objects must not total larger than this window.

The variables allocated to program memory are subject to similar constraints as data space objects in terms of maximum size, but they are allocated to the larger program space rather than data space memory. **Section 10.3.1.3 “Non-Auto Variable Size Limits”** which describes defining and using large arrays is also applicable to objects in program space memory.

10.4.4 Changing Program Memory Variable Allocation

The variables allocated to program memory can, to some degree, be allocated to alternate memory locations. **Section 10.3.1.4 “Changing Non-Auto Variable Allocation”** describes alternate addresses and sections also applicable to objects in the program memory space. Note that you cannot use the address attribute for objects that are in the `auto_psv` space.

The `space` attribute can be used to define variables that are positioned for use in the PSV window. To specify certain variables for allocation in the compiler-managed PSV space, use attribute `space(auto_psv)`. To allocate variables for PSV access in a section not managed by the compiler, use attribute `space(psv)`. For more information on these attributes, see **Chapter 7. “Differences Between MPLAB XC16 and ANSI C”**.

For example, to place a variable in the `auto_psv` space, which will cause storage to be allocated in Flash in a convenient way to be accessed by a *single* PSVPAG setting, specify:

```
unsigned int FLASH_variable __attribute__((space(auto_psv)));
```

Other user spaces that relate to Flash are available:

- `space(psv)` - a PSV space that the compiler does not access automatically
- `space(prog)` - any location in Flash that the compiler does not access automatically

Note that both the `psv` and `auto_psv` spaces are appropriately blocked or aligned so that a single PSVPAG setting is suitable for accessing the entire variable.

For more on PSV usage, see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*.

10.5 PARALLEL MASTER PORT ACCESS

Some devices contain a Parallel Master Port (PMP) peripheral which allows the connection of various memory and non-memory devices directly to the device. Access to the peripheral is controlled via a selection of peripherals. More information about this peripheral can be found in the Family Reference Manual or device-specific data sheets.

Note: PMP attributes are not supported on devices with EPMP. Use Extended Data Space (EDS) instead. See **Section 10.7 “Extended Data Space Access”**.

The peripheral can require a substantial amount of configuration, depending upon the type and brand of memory device that is connected. This configuration is not done automatically by the compiler.

The extensions presented here allow the definition of a variable as PMP. This means that the compiler will communicate with the PMP peripheral to access the variable.

To use this feature:

- Initialize PMP - define the initialization function: `void __init_PMP(void)`
- Declare a New Memory Space
- Define Variables within PMP Space

10.5.1 Initialize PMP

The PMP peripheral requires initialization before any access can be properly processed. Consult the appropriate documentation for the device you are interfacing to and the data sheet for 16-bit device you are using.

If PMP is used, the toolsuite will call `void __init_PMP(void)` during normal C run-time initialization. If a customized initialization is being used, please ensure that this function is called.

This function should make the necessary settings in the PMMODE and PMCON SFRs. In particular:

- The peripheral should not be configured to generate interrupts:
`PMMODEbits.IRQM = 0`
- The peripheral should not be configured to generate increments:
`PMMODEbits.INCM = 0`
The compiler will modify this setting during run-time as needed.
- The peripheral should be initialized to 16-bit mode:
`PMMODEbits.MODE16 = 1`
The compiler will modify this setting during run-time as needed.
- The peripheral should be configured for one of the MASTER modes:
`PMMODEbits.MODE = 2` or `PMMODEbits.MODE = 3`
- Set the wait-states `PMMODEbits.WAITB`, `PMMODEbits.WAITM`, and `PMMODEbits.WAITE` as appropriate for the device being connected.
- The PMCON SFR should be configured as appropriate making sure that the chip select function bits `PMCONbits.CSF` match the information communicated to the compiler when defining memory spaces.

A partial example might be:

```
void __init_PMP(void) {
    PMMODEbits.IRQM = 0;
    PMMODEbits.INCM = 0;
    PMMODEbits.MODE16 = 1;
    PMMODEbits.MODE = 3;
    /* device specific configuration of PMMODE and PMCCON follows */
}
```


10.5.2 Declare a New Memory Space

The compiler toolsuite requires information about each additional memory being attached via the PMP. In order for the 16-bit device linker to be able to properly assign memory, information about the size of memory available and the number of chip-selects needs to be provided.

In **Chapter 7. “Differences Between MPLAB XC16 and ANSI C”** the new `pmp` memory space was introduced. This attribute serves two purposes: declaring extended memory spaces and assigning C variable declarations to external memory (this will be covered in the next subsection).

Declaring an extended memory requires providing the size of the memory. You may optionally assign the memory to a particular chip-select pin; if none is assigned it will be assumed that chip-selects are not being used. These memory declarations look like normal external C declarations:

```
extern int external_PMP_memory
__attribute__((space(pmp(size(1024),cs(0)))));
```

Above we defined an external memory of size 1024 bytes and there are no chip-selects. The compiler only supports one PMP memory unless chip-selects are being used:

```
extern int PMP_bank1 __attribute__((space(pmp(size(1024),cs(1)))));
extern int PMP_bank2 __attribute__((space(pmp(size(2048),cs(2)))));
```

Above `PMP_bank1` will be activated using chip-select pin 1 (address pin 14 will be asserted when accessing variables in this bank). `PMP_bank2` will be activated using chip-select pin 2 (address pin 15 will be asserted).

Note that when using chip-selects, the largest amount of memory is 16 Kbytes per bank. It is recommended that the declaration appear in a common header file so that the declaration is available to all translation units.

10.5.3 Define Variables within PMP Space

The `pmp` space attribute is also used to assign individual variables to the space. This requires that the memory space declaration to be present. Given the declarations in the previous subsection, the following variable declarations can be made:

```
__pmp__ int external_array[256]
__attribute__((space(pmp(external_PMP_memory))));
```

`external_array` will be allocated in the previously declared memory `external_PMP_memory`. If there is only one PMP memory, and chip-selects are not being used, it is possible to leave out the explicit reference to the memory. It is good practice, however, to always make the memory explicit which would lead to code that is more easily maintained.

Note that, like managed PSV pointers, we have qualified the variable with a new type qualifier `__pmp__`. When attached to a variable or pointer it instructs the compiler to generate the correct sequence for access via the PMP peripheral.

Now that a variable has been declared it may be accessed using normal C syntax. The compiler will generate code to correctly communicate with the PMP peripheral.

10.6 EXTERNAL MEMORY ACCESS

Not all of Microchip's 16-bit devices have a parallel master port peripheral (see **Section 10.5 "Parallel Master Port Access"**), and not all memories are suitable for attaching to the PMP (serial memories sold by Microchip, for example). The toolsuite provides a more general interface to, what is known as, external memory, although, as will be seen, the memory does not have to be external.

Like PMP access, the tool-chain needs to learn about external memories that are being attached. Unlike PMP access, however, the compiler does not know how to access these memories. A mechanism is provided by which an application can specify how such memories should be accessed.

Addresses of external objects are all 32 bits in size. The largest attachable memory is 64K (16 bits); the other 16 bits in the address is used to uniquely identify the memory. A total of 64K (16 bits) of these may be (theoretically) attached.

To use this feature, work through the following sections.

10.6.1 Declare a New Memory Space

This is very similar to declaring a new memory space for PMP access.

The 16-bit toolsuite requires information about each external memory. In order for 16-bit device linker to be able to properly assign memory, information about the size of memory available and, optionally the origin of the memory, needs to be provided.

In **Chapter 7. "Differences Between MPLAB XC16 and ANSI C"** the `external` memory space was introduced. This attribute serves two purposes: declaring extended memory spaces and assigning C variable declarations to external memory (this will be covered in the next subsection).

Declaring an extended memory requires providing the size of the memory. You may optionally specify an origin for this memory; if none is specified 0x0000 will be assumed.

```
extern int external_memory
__attribute__((space(external(size(1024)))));
```

Above an external memory of size 1024 bytes is defined. This memory can be uniquely identified by its given name of `external_memory`.

10.6.2 Define Variables Within an External Space

The `external` space attribute is also used to assign individual variables to the space. This requires that the memory space declaration to be present. Given the declarations in the previous subsection, the following variable declarations can be made:

```
__external__ int external_array[256]
__attribute__((space(external(external_memory))));
```

`external_array` will be allocated in the previous declared memory `external_memory`.

Note that, like managed PSV objects, we have qualified the variable with a new type qualifier `__external__`. When attached to a variable or pointer target, it instructs the compiler to generate the correct sequence to access these objects.

Once an external memory variable has been declared, it may be accessed using normal C syntax. The compiler will generate code to access the variable via special helper functions that the programmer must define. These are covered in the next subsection.

10.6.3 Define How to Access Memory Spaces

References to variables placed in external memories are controlled via the use of several helper functions. Up to five functions may be defined for reading and five for writing. One of these functions is a generic routine and will be called if any of the other four are not defined but are required. If none of the functions are defined, the compiler will generate an error message. A brief example will be presented in the next subsection. Generally, defining the individual functions will result in more efficient code generation.

10.6.3.1 FUNCTIONS FOR READING

read_external

```
void __read_external(unsigned int address,
    unsigned int memory_space,
    void *buffer,
    unsigned int len)
```

This function is a generic Read function and will be called if one of the next functions are required but not defined. This function should perform the steps necessary to fill `len` bytes of memory in the `buffer` from the external memory named `memory_space` starting at address `address`.

read_external8

```
unsigned char __read_external8(unsigned int address,
    unsigned int memory_space)
```

Read 8 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access an 8-bit sized object.

read_external16

```
unsigned int __read_external16(unsigned int address,
    unsigned int memory_space)
```

Read 16 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access an 16-bit sized object.

read_external32

```
unsigned long __read_external32(unsigned int address,
    unsigned int memory_space)
```

Read 32 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access a 32-bit sized object, such as a `long` or `float` type.

read_external64

```
unsigned long long __read_external64(unsigned int address,
    unsigned int memory_space)
```

Read 64 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access a 64-bit sized object, such as a `long long` or `long double` type.

10.6.3.2 FUNCTIONS FOR WRITING

write_external

```
void __write_external(unsigned int address,
    unsigned int memory_space,
    void *buffer,
    unsigned int len)
```

This function is a generic Write function and will be called if one of the next functions are required but not defined. This function should perform the steps necessary to write `len` bytes of memory from the `buffer` to the external memory named `memory_space` starting at address `address`.

write_external8

```
void __write_external8(unsigned int address,
    unsigned int memory_space,
    unsigned char data)
```

Write 8 bits of data to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write an 8-bit sized object.

write_external16

```
void __write_external16(unsigned int address,
    unsigned int memory_space,
    unsigned int data)
```

Write 16 bits of data to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write an 16-bit sized object.

write_external32

```
void __write_external32(unsigned int address,
    unsigned int memory_space,
    unsigned long data)
```

Write 32 bits of data to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write a 32-bit sized object, such as a `long` or `float` type.

write_external64

```
void __write_external64(unsigned int address,
    unsigned int memory_space,
    unsigned long long data)
```

Write 64 bits of data to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write a 64-bit sized object, such as a `long long` or `long double` type.

10.6.4 An External Example

The following snippets of example come from a working example (in the Examples folder.)

This example implements, using external memory, addressable bit memory. In this case each bit is stored in real data memory, not off chip. The code will define an external memory of 512 units and map accesses using the appropriate read and write function to one of 64 bytes in local data memory.

First the external memory is defined:

```
extern unsigned int bit_memory
__attribute__((space(external(size(512)))));
```

Next appropriate read and write functions are defined. These functions will make use of an array of memory that is reserved in the normal way.

```
static unsigned char real_bit_memory[64];
unsigned char __read_external8(unsigned int address,
                               unsigned int memory_space) {
    if (memory_space == bit_memory) {
        /* an address within our bit memory */
        unsigned int byte_offset, bit_offset;
        byte_offset = address / 8;
        bit_offset = address % 8;
        return (real_bit_memory[byte_offset] >> bit_offset) & 0x1;
    } else {
        fprintf(stderr, "I don't know how to access memory space: %d\n",
                memory_space);
    }
    return 0;
}

void __write_external8(unsigned int address,
                      unsigned int memory_space,
                      unsigned char data) {
    if (memory_space == bit_memory) {
        /* an address within our bit memory */
        unsigned int byte_offset, bit_offset;
        byte_offset = address / 8;
        bit_offset = address % 8;
        real_bit_memory[byte_offset] &= (~(1 << bit_offset));
        if (data & 0x1) real_bit_memory[byte_offset] |=
            (1 << bit_offset);
    } else {
        fprintf(stderr, "I don't know how to access memory space: %d\n",
                memory_space);
    }
}
```

These functions work in a similar fashion:

- if accessing `bit_memory`, then
 - determine the correct byte offset and bit offset
 - read or write the appropriate place in the `real_bit_memory`
- otherwise access another memory (whose access is unknown)

With the two major pieces of the puzzle in place, generate some variables and accesses:

```
__external__ unsigned char bits[NUMBER_OF_BITS]
__attribute__((space(external(bit_memory)))));

// inside main
__external__ unsigned char *bit;
bit = bits;
for (i = 0; i < 512; i++) {
    printf("%d ", *bit++);
}
```

Apart from the `__external__` CV-qualifiers, ordinary C statements can be used to define and access variables in the external memory space.

10.7 EXTENDED DATA SPACE ACCESS

Qualifying a variable or pointer target as being accessible through the extended data space window allows you to easily access objects that have been placed in a variety of different memory spaces. These include: `space(data)` (and its subsets), `eds`, `space(eedata)`, `space(prog)`, `space(psv)`, `space(auto_psv)`, and on some devices `space(pmp)`. Not all devices support all memory spaces.

To use this feature:

- declare an object in an appropriate memory space
- qualify the object with the `__eds__` qualifier

For example:

```
__eds__ int var_a __attribute__((space(prog)));
__eds__ int var_b [10] __attribute__((eds));
__eds__ int *var_c;
__eds__ int *__eds__ *var_d __attribute__((space(psv)));
```

`var_a` - declares an `int` in Flash that is automatically accessed

`var_b` - declares an array of `ints`, located in `eds`; the elements of the array are automatically accessed

`var_c` - declares a pointer to an `int`, where the destination may exist in any one of the memory spaces supported by Extended Data Space pointers and will be automatically accessed upon dereference; the pointer itself must live in a normal data space

`var_d` - declares a pointer to an `int`, where the destination may exist in any one of the memory spaces supported by Extended Data Space pointers and will be automatically accessed upon dereference; the pointer value exists in Flash and is also automatically accessed.

The compiler will automatically assert the `page` attribute to scalar variable declarations; this allows the compiler to generate more efficient code when accessing larger data types. Remember, scalar variables do not include structures or arrays. To force paging of a structure or array, please manually use the `page` attribute and the compiler will prevent the object from crossing a page boundary.

For read access to `__eds__` qualified variables, the compiler will automatically manipulate the `PSVPAG` or `DSRPAG` register as appropriate. For devices that support extended data space memory, the compiler will also manipulate the `DSWPAG` register.

Note: Some devices use <code>DSRPAG</code> to represent extended read access to FLASH or the extended data space (EDS)

10.8 PACKING DATA STORED IN FLASH

The 16-bit core families use a 24-bit Flash word size. The architecture supports the mapping of areas of Flash into the data space, as discussed in **Section 10.4 “Variables in Program Space”**. Unfortunately this mapping is only 16 bits wide to fit in with data space dimensions.

The compiler supports using the upper byte of Flash via packed storage. Use of this upper byte can offer a code-size savings for large structures, but this is more expensive to access. The type-qualifier `__pack_upper_byte` added to a declaration indicates that the variable should be placed into Flash memory and use the upper byte. Unlike other qualifiers in use with MPLAB XC16 C Compiler, such as `__psv__`, this qualifier combines placement and access control.

10.8.1 Packed Example

```
__pack_upper_byte char message[] = "Hello World!\n";
```

will allocate the message string into Flash memory in such a way that the upper byte of memory contains valid data.

There are no restrictions to the types of `__pack_upper_byte` data. The compiler will 'pack' structures as if `__attribute__((packed))` had also been specified. This further eliminates wasted space due to padding.

Like other extended type qualifiers, the `__pack_upper_byte` type qualifier enforces a unique addressing space on the compiler; therefore, it is important to maintain this qualifier when passing values as parameters. Do not be tempted to cast away the `__pack_upper_byte` qualifier – it won't work.

10.8.2 Usage Considerations

When using this qualifier, consider the following:

1. `__pack_upper_byte` data is best used for large data sets that do not need to be accessed frequently or that do not have important access timing.
2. Sequential accesses to `__pack_upper_byte` data objects will improve access performance.
3. A version of `memcpy` is defined in `libpic30.h`, and its prototype is:

```
void _memcpy_packed(void *dst, __pack_upper_byte void *src,  
                    unsigned int len);
```
4. The following style of declaration is invalid for packed memory:

```
__pack_upper_byte char *message = "Hello World!\n";
```

Here, `message` is a pointer to `__pack_upper_byte` space, but the string "Hello World!\n", is in normal const data space, which is not compatible with `__pack_upper_byte`. There is no standard C way to specify a different source address space for the literal string. Instead declare `message` as an object (such as an array declaration in **Section 10.8.1 “Packed Example”**).
5. The TBLPAG SFR may be corrupted during access of a packed variable.

10.8.3 Addressing Information

The upper byte of Flash does not have a unique address, which is a requirement for C. Therefore, the compiler has to invent one. The tool chain remaps Flash to linear addresses for all bytes starting with program address word 0. This means that the real Flash address of a `__pack_upper_byte` variable will not be the address that is stored in a pointer or symbol table. The Flash address can be determined by:

1. word offset = address div 3
2. program address offset = word offset * 2
3. byte offset = address mod 3

The byte to reference is located in Flash at *program address offset*.

The remapped addressing scheme for `__pack_upper_byte` objects prevents the compiler from accepting fixed address requests.

10.9 ALLOCATION OF VARIABLES TO REGISTERS

Note: Using variables specified in compiler-allocated registers - fixed registers - is usually unnecessary and occasionally dangerous. This feature is deprecated and not recommended.

You may specify a fixed register assignment for a particular C variable. It is not recommended that this be done.

DD 10.10 VARIABLES IN EEPROM

The compiler provides some convenience macro definitions to allow placement of data into the device's EE data area. This can be done quite simply:

```
int _EEDATA(2) user_data[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

`user_data` will be placed in the EE data space (`space(eedata)`) reserving 10 words with the given initial values.

The device provides two ways for programmers to access this area of memory. The first is via the program space visibility window. The second is by using special machine instructions (TBLRDx).

10.10.1 Accessing EEDATA via User Managed PSV

The compiler normally manages the PSV window to access constants stored in program memory. If this is not the case, the PSV window can be used to access EEDATA memory.

To use the PSV window:

- The `psv` page register must be set to the appropriate address for the program memory to be accessed. For EE data this will be 0xFF, but it is best to use the `__builtin_psvpage()` function.
- In some devices, the PSV window should also be enabled by setting the PSV bit in the CORCON register. If this bit is not set, uses of the PSV window will always read 0x0000.

EXAMPLE 10-1: EEDATA ACCESS VIA PSV

```
#include <xc.h>
int main(void) {
    PSVPAG = __builtin_psvpage(&user_data);
    CORCONbits.PSV = 1;

    /* ... */

    if (user_data[2]) /* do something */

}
```

These steps need only be done once. Unless `psv` page is changed, variables in EE data space may be read by referring to them as normal C variables, as shown in the example.

Note: This access model is not compatible with the compiler-managed PSV (`-mconst-in-code`) model. You should be careful to prevent conflict.

10.10.2 Accessing EEDATA Using TBLRDx Instructions

The TBLRDx instructions are not directly supported by the compiler, but they can be used via inline assembly or compiler built-in functions. Like PSV accesses, a 23-bit address is formed from an SFR value and the address encoded as part of the instruction. To access the same memory as given in the previous example, the following code may be used:

To use the TBLRDx instructions:

- The TBLPAG register must be set to the appropriate address for the program memory to be accessed. For EE data, this will be 0x7F, but it is best to use the `__builtin_tblpage()` function.
- The TBLRDx instruction can be accessed from an `__asm__` statement or through one of the `__builtin_tblrd` functions; refer to the “*dsPIC30F/33F Programmer's Reference Manual*” (DS70157) for information on this instruction.

EXAMPLE 10-2: EEDATA ACCESS VIA TABLE READ

```
#include <xc.h>
#define eedata_read(src, offset, dest) { \
    register int eedata_addr;           \
    register int eedata_val;            \
                                         \
    eedata_addr = __builtin_tbloffset(&src)+offset; \
    eedata_val = __builtin_tblrdl(eedata_addr); \
    dest = eedata_val;                  \
}

char user_data[] __attribute__((space(eedata))) = { /* values */ };

int main(void) {
    int value;

    TBLPAG = __builtin_tblpage(&user_data);

    eedata_read(user_data, 2*sizeof(user_data[0]), value);
    if (value) ; /* do something */

}
```

10.10.3 Accessing EEDATA Using Managed Access

On most device the EE Data space is part of the program address space. Therefore EEData can be accessed automatically using one of the managed access qualifiers `__psv__` or `__eds__`.

EXAMPLE 10-3: EXAMPLE 6-2 USING MANAGED PSV ACCESS

```
#include <xc.h>

__eds__ char user_data[] __attribute__((space(eedata))) = { /* values
*/ };

int main(void) {
    int value;

    value = user_data[0];
    if (value) ; /* do something */
}
```

10.10.4 Additional Sources of Information

The device Family Reference Manuals have an excellent discussion on using the Flash program memory and EE data memory provided. These manuals also have information on run-time programming of both program memory and EE data memory.

There are many library routines provided with the compiler. See the *16-Bit Language Tools Libraries* (DS51456) manual for more information.

10.11 DYNAMIC MEMORY ALLOCATION

The C run-time heap is an uninitialized area of data memory that is used for dynamic memory allocation using the standard C library dynamic memory management functions, `calloc`, `malloc` and `realloc`. If you do not use any of these functions, then you do not need to allocate a heap. By default, a heap is not created.

If you do want to use dynamic memory allocation, either directly, by calling one of the memory allocation functions, or indirectly, by using a standard C library input/output function, then a heap must be created. A heap is created by specifying its size on the linker command line, using the `--heap` linker command-line option. An example of allocating a heap of 512 bytes using the command line is:

```
xc16-gcc foo.c -Wl,--heap=512
```

The linker allocates the heap immediately below the stack.

You can use a standard C library input/output function to create open files (`fopen`). If you open files, then the heap size must include 40 bytes for each file that is simultaneously open. If there is insufficient heap memory, then the `open` function will return an error indicator. For each file that should be buffered, 4 bytes of heap space is required. If there is insufficient heap memory for the buffer, then the file will be opened in unbuffered mode. The default buffer can be modified with `setvbuf` or `setbuf`.

10.12 MEMORY MODELS

The compiler supports several memory models. Command-line options are available for selecting the optimum memory model for your application, based on the specific device that you are using and the type of memory usage.

TABLE 10-1: MEMORY MODEL COMMAND LINE OPTIONS

Option	Memory Definition	Description
<code>-msmall-data</code>	Up to 8 KB of data memory. This is the default.	Permits use of PIC18 like instructions for accessing data memory.
<code>-msmall-scalar</code>	Up to 8 KB of data memory. This is the default.	Permits use of PIC18 like instructions for accessing scalars in data memory.
<code>-mlarge-data</code>	Greater than 8 KB of data memory.	Uses indirection for data references.
<code>-msmall-code</code>	Up to 32 kWords of program memory. This is the default.	Function pointers will not go through a jump table. Function calls use <code>RCALL</code> instruction.
<code>-mlarge-code</code>	Greater than 32 kWords of program memory.	Function pointers might go through a jump table. Function calls use <code>CALL</code> instruction.
<code>-mconst-in-data</code>	Constants located in data memory.	Values copied from program memory by startup code.
<code>-mconst-in-code</code>	Constants located in program memory. This is the default.	Values are accessed via Program Space Visibility (PSV) data window.
<code>-mconst-in-aux-flash</code>	Constants in auxiliary FLASH	Values are accessed via Program Space visibility window.

The command-line options apply globally to the modules being compiled. Individual variables and functions can be declared as `near`, `far` or in `eds` to better control the code generation. For information on setting individual variable or function attributes, see **Section 8.12 “Variable Attributes”** and **Section 13.2.1 “Function Specifiers”**.

10.12.1 Near or Far Data

If variables are allocated in the near data space, the compiler is often able to generate better (more compact) code than if the variables are not allocated in near data. If all variables for an application can fit within the 8 KB of near data, then the compiler can be requested to place them there by using the default `-msmall-data` command line option when compiling each module. If the amount of data consumed by scalar types (no arrays or structures) totals less than 8 KB, the default `-msmall-scalar`, combined with `-mlarge-data`, may be used. This requests that the compiler arrange to have just the scalars for an application allocated in the near data space.

If neither of these global options is suitable, then the following alternatives are available.

1. It is possible to compile some modules of an application using the `-mlarge-data` or `-mlarge-scalar` command line options. In this case, only the variables used by those modules will be allocated in the far data section. If this alternative is used, then care must be taken when using externally defined variables. If a variable that is used by modules compiled using one of these options is defined externally, then the module in which it is defined must also be compiled using the same option, or the variable declaration and definition must be tagged with the far attribute.
2. If the command line options `-mlarge-data` or `-mlarge-scalar` have been used, then an individual variable may be excluded from the far data space by tagging it with the near attribute.
3. Instead of using command-line options, which have module scope, individual variables may be placed in the far data section by tagging them with the `far` attribute.

The linker will produce an error message if all near variables for an application cannot fit in the 8K near data space.

NOTES:

Chapter 11. Operators and Statements

11.1 INTRODUCTION

The MPLAB XC16 C Compiler supports all the ANSI operators. The exact results of some of these are implementation defined and this behavior is fully documented in **Appendix A. "Implementation-Defined Behavior"**. The following sections illustrate code operations that are often misunderstood as well as additional operations that the compiler is capable of performing.

- Built-In Functions
- Integral Promotion

11.2 BUILT-IN FUNCTIONS

Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using inline assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and usually do not involve function calls or library routines.

For more on built-in functions, see **Appendix G. "Built-in Functions"**.

11.3 INTEGRAL PROMOTION

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they do have the same type. The conversion is to a "larger" type so there is no loss of information; however, the change in type can cause different code behavior to what is sometimes expected. These form the standard type conversions.

Prior to these type conversions, some operands are unconditionally converted to a larger type, even if both operands to an operator have the same type. This conversion is called *integral promotion* and is part of Standard C behavior. The compiler performs these integral promotions where required, and there are no options that can control or disable this operation. If you are not aware that the type has changed, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, signed or unsigned varieties of `char`, `short int` or bit-field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example.

```
unsigned char count, a=0, b=50;
if(a - b < 10)
    count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which is less than 10) and hence the body of the `if()` statement is executed.

If the result of the subtraction is to be an `unsigned` quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
    count++;
```

The comparison is then done using `unsigned int`, in this case, and the body of the `if()` would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, `~`. This operator toggles each bit within a value. Consider the following code.

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
    count++;
```

If `c` contains the value `0x55`, it is often assumed that `~c` will produce `0xAA`, however the result is `0xFFAA` and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with `char`-type operands, but with `int`-type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type `char` or `int`. In these cases, the compiler will not perform the integral promotion so as to increase the code efficiency. Consider the following example.

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of `b` and `c` should be promoted to `unsigned int`, the addition performed, the result of the addition cast to the type of `a`, and then the assignment can take place. Even if the result of the `unsigned int` addition of the promoted values of `b` and `c` was different to the result of the `unsigned char` addition of these values without promotion, after the `unsigned int` result was converted back to `unsigned char`, the final result would be the same. If an 8-bit addition is more efficient than a 16-bit addition, the compiler will encode the former.

If, in the above example, the type of `a` was `unsigned int`, then integral promotion would have to be performed to comply with the ANSI C standard.

Chapter 12. Register Usage

12.1 INTRODUCTION

Certain registers play import roles in the C runtime environment. Therefor creating code concerning these registers requires knowledge about their use by the compiler.

- Register Variables
- Changing Register Contents

12.2 REGISTER VARIABLES

Register variables use one or more working registers, as shown in Table 12.1.

TABLE 12.1: REGISTER CONVENTIONS

Variable	Working Register
char, signed char, unsigned char	W0-W13, and W14 if not used as a Frame Pointer.
short, signed short, unsigned short	W0-W13, and W14 if not used as a Frame Pointer.
int, signed int, unsigned int	W0-W13, and W14 if not used as a Frame Pointer.
void * (or any pointer)	W0-W13, and W14 if not used as a Frame Pointer.
long, signed long, unsigned long	A pair of contiguous registers, the first of which is a register from the set {W0, W2, W4, W6, W8, W10, W12}.
long long, signed long long, unsigned long long	A quadruplet of contiguous registers, the first of which is a register from the set {W0, W4, W8}. Successively higher-numbered registers contain successively more significant bits.
float	A pair of contiguous registers, the first of which is a register from the set {W0, W2, W4, W6, W8, W10, W12}.
double ¹	A pair of contiguous registers, the first of which is a register from the set {W0, W2, W4, W6, W8, W10, W12}.
long double	A quadruplet of contiguous registers, the first of which is a register from the set {W0, W4, W8}.
structure	1 contiguous register per 2 bytes in the structure.
_Fract _Sat _Fract	W0-W13, and W14 if not used as a Frame Pointer.
long _Fract _Sat long _Fract	A pair of contiguous registers, the first of which is a register from the set {W0, W2, W4, W6, W8, W10, W12}.
_Accum _Sat _Accum	Three contiguous registers where the first register starts in the set {W0, W4, W8} and W12 if W14 is not used as a frame pointer.

Note 1: double is equivalent to long double if -fno-short-double is used.

12.3 CHANGING REGISTER CONTENTS

The assembly generated from C source code by the compiler will use certain registers that are present on the 16-bit device. Most importantly, the compiler assumes that nothing other than code it generates can alter the contents of these registers. So if the assembly loads a register with a value and no subsequent code generation requires this register, the compiler will assume that the contents of the register are still valid later in the output sequence.

The registers that are special and which are managed by the compiler are: W0-W15, RCOUNT, STATUS (SR), PSVPAG and DSRPAG. If fixed point support is enabled, the compiler may allocate A and B, in which case the compiler may adjust CORCON.

The state of these register must never be changed directly by C code, or by any assembly code in-line with C code. The following example shows a C statement and in-line assembly that violates these rules and changes the ZERO bit in the STATUS register.

```
#include <xc.h>

void badCode(void)
{
    asm ("mov #0, w8");
    WREG9 = 0;
}
```

The compiler is unable to interpret the meaning of in-line assembly code that is encountered in C code. Nor does it associate a variable mapped over an SFR to the actual register itself. Writing to an SFR register using either of these two methods will not flag the register as having changed and may lead to code failure.

Chapter 13. Functions

13.1 INTRODUCTION

The compiler supports C code functions and handles assembly code functions, as discussed in the following topics:

- Writing Functions
- Function Size Limits
- Allocation of Function Code
- Changing the Default Function Allocation
- Inline Functions
- Memory Models
- Function Call Conventions

13.2 WRITING FUNCTIONS

Implementation and special features associated with functions are discussed in the following sections.

13.2.1 Function Specifiers

The only specifier that has any effect on functions is `static`.

A function defined using the `static` specifier only affects the scope of the function, i.e. limits the places in the source code where the function may be called. Functions that are `static` may only be directly called from code in the file in which the function is defined. This specifier does not change the way the function is encoded.

13.2.2 Function Attributes

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently supported for functions:

- `address (addr)`
- `alias ("target")`
- `auto_psv, no_auto_psv`
- `boot`
- `const`
- `deprecated`
- `far`
- `format (archetype, string-index, first-to-check)`
- `format_arg (string-index)`
- `interrupt [([save(list)] [, irq(irqid)] [, altirq(altirqid)] [, preprologue(asm)])]`
- `keep`
- `naked`
- `near`
- `no_instrument_function`
- `noload`
- `noreturn`
- `round(mode)`
- `save(list)`
- `section ("section-name")`
- `secure`
- `shadow`
- `unsupported("message")`
- `unused`
- `user_init`
- `weak`

You may also specify attributes with `__` (double underscore) preceding and following each keyword (e.g., `__shadow__` instead of `shadow`). This allows you to use them in header files without being concerned about a possible macro of the same name.

Multiple attributes may be specified in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

address (*addr*)

The `address` attribute specifies an absolute address for the function.

```
void __attribute__((address(0x100))) foo() {  
    ...  
}
```

Alternatively, you may define the address in the function prototype:

```
void foo() __attribute__((address(0x100)));
```

alias ("*target*")

The `alias` attribute causes the declaration to be emitted as an alias for another symbol, which must be specified.

Use of this attribute results in an external reference to `target`, which must be resolved during the link phase.

auto_psv, no_auto_psv

The `auto_psv` attribute, when combined with the `interrupt` attribute, will cause the compiler to generate additional code in the function prologue to set the `psv` page SFR to the correct value for accessing `space(auto_psv)` (or constants in the constants-in-code memory model) variables. Use this option when using 24-bit pointers and an interrupt may occur while the `psv` page has been modified and the interrupt routine, or a function it calls, uses an `auto_psv` variable. Compare this with `no_auto_psv`.

The `no_auto_psv` attribute, when combined with the `interrupt` attribute, will cause the compiler to not generate additional code for accessing `space(auto_psv)` (or constants in the constants-in-code memory model) variables. Use this option if none of the conditions under `auto_psv` hold true.

If neither `auto_psv` nor `no_auto_psv` option is specified for an interrupt routine, the compiler will issue a warning and assume `auto_psv`.

boot

This attribute directs the compiler to allocate a function in the `boot` segment of program Flash.

For example, to declare a protected function:

```
void __attribute__((boot)) func();
```

An optional argument can be used to specify a protected access entry point within the `boot` segment. The argument may be a literal integer in the range 0 to 31 (except 16), or the word `unused`. Integer arguments correspond to 32 instruction slots in the segment access area, which occupies the lowest address range of each secure segment. The value 16 is excluded because access entry 16 is reserved for the secure segment interrupt vector. The value `unused` is used to specify a function for all of the unused slots in the access area.

Access entry points facilitate the creation of application segments from different vendors that are combined at run time. They can be specified for external functions as well as locally defined functions.

For example:

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
    boot_service3();
}
```

Note: In order to allocate functions with the `boot` or `secure` attribute, memory for the boot and/or secure segment must be reserved. This can be accomplished by setting configuration words in source code, or by specifying linker command options. For more information, see Chapter 8.8, “Options that Specify CodeGuard Security Features”, in the linker manual (DS51317).

If attributes `boot` or `secure` are used, and memory is not reserved, then a link error will result.

To specify a secure interrupt handler, use the `boot` attribute in combination with the interrupt attribute:

```
void __attribute__((boot,interrupt)) boot_interrupts();
```

When an access entry point is specified for an external secure function, that function need not be included in the project for a successful link. All references to that function will be resolved to a fixed location in Flash, depending on the security model selected at link time.

When an access entry point is specified for a locally defined function, the linker will insert a branch instruction into the secure segment access area. The exception is for access entry 16, which is represented as a vector (i.e, an instruction address) rather than an instruction. The actual function definition will be located beyond the access area; therefore the access area will contain a jump table through which control can be transferred from another security segment to functions with defined entry points.

Automatic variables are owned by the enclosing function and do not need the `boot` attribute. They may be assigned initial values, as shown:

```
void __attribute__((boot)) chuck_cookies()
{
    int hurl;
    int them = 55;
    char *where = "far";
    splat(where);
    /* ... */
}
```

Note that the initial value of `where` is based on a string literal which is allocated in the PSV constant section `.boot_const`. The compiler will set the `psv` page SFR to the correct value upon entrance to the function. If necessary, the compiler will also restore it after the call to `splat()`.

const

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `const`. For example:

```
int square (int) __attribute__((const));
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to have a `void` return type.

deprecated

See **Section 8.12 “Variable Attributes”** for information on the `deprecated` attribute.

far

The `far` attribute tells the compiler that the function may be located too far away to use short call instruction.

format (archetype, string-index, first-to-check)

The `format` attribute specifies that a function takes `printf`, `scanf` or `strftime` style arguments which should be type-checked against a format string. For example, consider the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
__attribute__((format (printf, 2, 3)));
```

This causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter *archetype* determines how the format string is interpreted, and should be one of `printf`, `scanf` or `strftime`. The parameter *string-index* specifies which argument is the format string argument (arguments are numbered from the left, starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case, the compiler only checks the format string for consistency.

In the previous example, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions that take format strings as arguments, so that the compiler can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `strftime`, `vprintf`, `vfprintf` and `vsprintf`, whenever such warnings are requested (using `-Wformat`), so there is no need to modify the header file `stdio.h`.

format_arg (string-index)

The `format_arg` attribute specifies that a function takes `printf` or `scanf` style arguments, modifies it (for example, to translate it into another language), and passes it to a `printf` or `scanf` style function. For example, consider the declaration:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
```

```
__attribute__ ((format_arg (2)));
```

This causes the compiler to check the arguments in calls to `my_dgettext`, whose result is passed to a `printf`, `scanf` or `strftime` type function for consistency with the `printf` style format string argument `my_format`.

The parameter *string-index* specifies which argument is the format string argument (starting from 1).

The `format-arg` attribute allows you to identify your own functions which modify format strings, so that the compiler can check the calls to `printf`, `scanf` or `strftime` function, whose operands are a call to one of your own functions.

```
interrupt [ ( [ save(list) ] [, irq(irqid) ]  
[, altirq(altirqid) ] [, preprologue(asm) ] ) ]
```

Use this option to indicate that the specified function is an interrupt handler. The compiler will generate function prologue and epilogue sequences suitable for use in an interrupt handler when this attribute is present. The optional parameter `save` specifies a list of variables to be saved and restored in the function prologue and epilogue, respectively. The optional parameters `irq` and `altirq` specify interrupt vector table IDs to be used. The optional parameter `preprologue` specifies assembly code that is to be emitted before the compiler-generated prologue code. See **Chapter 14. “Interrupts”** for a full description, including examples.

When using the `interrupt` attribute, please specify either `auto_psv` or `no_auto_psv`. If none is specified a warning will be produced and `auto_psv` will be assumed.

keep

The `keep` attribute will prevent the linker from removing the function with the ELF linker option `--gc-sections`, even if it is unused.

```
void __attribute__((keep)) func();
```

naked

The `naked` attribute will prevent the compiler from saving or restoring any registers. This attribute should be applied with caution as failing to save or restore registers may cause issues. Consider using this attribute with `noreturn` for safety - any attempt to return will cause a reset.

```
void __attribute__((naked)) func();
```

near

The `near` attribute tells the compiler that the function can be called using a more efficient form of the call instruction.

no_instrument_function

If the command line option `-finstrument-function` is given, profiling function calls will be generated at entry and exit of most user-compiled functions. Functions with this attribute will not be so instrumented.

noload

The `noload` attribute indicates that space should be allocated for the function, but that the actual code should not be loaded into memory. This attribute could be useful if an application is designed to load a function into memory at run time, such as from a serial EEPROM.

```
void bar() __attribute__((noload)) {  
...  
}
```


noreturn

A few standard library functions, such as `abort` and `exit`, cannot return. The compiler knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example:

```
void fatal (int i) __attribute__((noreturn));

void
fatal (int i)
{
    /* Print error message. */
    exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. Also, it helps avoid spurious warnings of uninitialized variables.

It does not make sense for a `noreturn` function to have a return type other than `void`.

A `noreturn` function will reset if it attempts to return.

round(mode)

The `round` attribute controls the rounding mode of C language fixed-point support (`_Fract`, `_Accum` variable types) dialect code (`-menable-fixed` command-line option) within a function. Specify `mode` as one of `truncation`, `conventional`, or `convergent`. This attribute overrides the default rounding mode set by `-menable-fixed` for C language code within the attributed function, but has no effect on functions that may be called by that function.

save(list)

Functions declared with the `save(list)` attribute will direct the compiler to save and restore the C variables expressed in `list`.

section ("section-name")

Normally, the compiler places the code it generates in the `.text` section. Sometimes, however, you need additional sections, or you need certain functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, consider the declaration:

```
extern void foobar (void) __attribute__((section (".libtext")));
```

This puts the function `foobar` in the `.libtext` section.

The linker will allocate the saved named section sequentially. This might allow you to ensure code is locally referent to each other, even across modules. This can ensure that calls are near enough to each other for a more efficient call instruction.

secure

This attribute directs the compiler to allocate a function in the `secure` segment of program Flash.

For example, to declare a protected function:

```
void __attribute__((secure)) func();
```

An optional argument can be used to specify a protected access entry point within the `secure` segment. The argument may be a literal integer in the range 0 to 31 (except 16), or the word `unused`. Integer arguments correspond to 32 instruction slots in the segment access area, which occupies the lowest address range of each secure segment. The value 16 is excluded because access entry 16 is reserved for the secure segment interrupt vector. The value `unused` is used to specify a function for all of the unused slots in the access area.

Access entry points facilitate the creation of application segments from different vendors that are combined at run time. They can be specified for external functions as well as locally defined functions. For example:

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
    boot_service3();
}
```

Note: In order to allocate functions with the `boot` or `secure` attribute, memory for the boot and/or secure segment must be reserved. This can be accomplished by setting configuration words in source code, or by specifying linker command options. For more information, see Chapter 8.8, “Options that Specify CodeGuard Security Features”, in the linker manual (DS51317).

If attributes `boot` or `secure` are used, and memory is not reserved, then a link error will result.

To specify a secure interrupt handler, use the `secure` attribute in combination with the interrupt attribute:

```
void __attribute__((secure,interrupt)) secure_interrupts();
```

When an access entry point is specified for an external secure function, that function need not be included in the project for a successful link. All references to that function will be resolved to a fixed location in Flash, depending on the security model selected at link time.

When an access entry point is specified for a locally defined function, the linker will insert a branch instruction into the secure segment access area. The exception is for access entry 16, which is represented as a vector (i.e., an instruction address) rather than an instruction. The actual function definition will be located beyond the access area; therefore the access area will contain a jump table through which control can be transferred from another security segment to functions with defined entry points.

Automatic variables are owned by the enclosing function and do not need the `secure` attribute. They may be assigned initial values, as shown:

```
void __attribute__((secure)) chuck_cookies()
{
    int hurl;
    int them = 55;
    char *where = "far";
    splat(where);
    /* ... */
}
```

Note that the initial value of `where` is based on a string literal which is allocated in the PSV constant section `.secure_const`. The compiler will set PSVPAG to the correct value upon entrance to the function. If necessary, the compiler will also restore PSVPAG after the call to `splat()`.

shadow

The `shadow` attribute causes the compiler to use the shadow registers rather than the software stack for saving registers. This attribute is usually used in conjunction with the `interrupt` attribute.

```
void __attribute__((interrupt, shadow)) _T1Interrupt (void);
```

unsupported ("message")

This attribute will display a custom message when the object is used.

```
int foo __attribute__((unsupported("This object is unsupported")));
```

Access to `foo` will generate a warning message.

unused

This attribute, attached to a function, means that the function is meant to be possibly unused. The compiler will not produce an unused function warning for this function.

user_init

The `user_init` attribute may be applied to any non-interrupt function with `void` parameter and return types. Applying this attribute will cause default C start-up modules to call this function before the user main is executed. There is no guarantee of ordering, so these functions cannot rely on other `user_init` functions having been previously run; these functions will be called after PSV and data initialization. A `user_init` may still be called by the executing program. For example:

```
void __attribute__((user_init)) initialize_me(void) {
    // perform initialization sequence alpha alpha beta
}
```

weak

See **Section 8.12 “Variable Attributes”** for information on the `weak` attribute.

13.3 FUNCTION SIZE LIMITS

For all devices, the code generated for a function may become larger than one page in size, limited only by the available program memory. However, functions that yield code larger than a page may not be as efficient due to longer call sequences to jump to and call destinations in other pages. See **13.4 “Allocation of Function Code”** for more details.

13.4 ALLOCATION OF FUNCTION CODE

Code associated with functions is always placed in the program memory of the target device. As described in **Section 10.2 “Address Spaces”**, the compiler arranges for code to be placed in the `.text` section, depending on the memory model used and whether or not the data is initialized. When modules are combined at link time, the linker determines the starting addresses of the various sections based on their attributes.

13.5 CHANGING THE DEFAULT FUNCTION ALLOCATION

Cases may arise when a specific function must be located at a specific address, or within some range of addresses. The easiest way to accomplish this is by using the `address` attribute, described in **Section 13.2.1 “Function Specifiers”**. For example, to locate function `PrintString` at address `0x8000` in program memory:

```
int __attribute__((address(0x8000))) PrintString (const char *s);
```

Another way to locate code is by placing the function into a user-defined section, and specifying the starting address of that section in a custom linker script. This is done as follows:

1. Modify the code declaration in the C source to specify a user-defined section.
2. Add the user-defined section to a custom linker script file to specify the starting address of the section.

For example, to locate the function `PrintString` at address `0x8000` in program memory, first declare the function as follows in the C source:

```
int __attribute__((__section__(".myTextSection")))
PrintString(const char *s);
```

The section attribute specifies that the function should be placed in a section named `.myTextSection`, rather than the default `.text` section. It does not specify where the user-defined section is to be located. That must be done in a custom linker script, as follows. Using the device-specific linker script as a base, add the following section definition:

```
.myTextSection 0x8000 :
{
    *(.myTextSection);
} >program
```

This specifies that the output file should contain a section named `.myTextSection` starting at location `0x8000` and containing all input sections named `.myTextSection`. Since, in this example, there is a single function `PrintString` in that section, then the function will be located at address `0x8000` in program memory.

13.6 INLINE FUNCTIONS

By declaring a function `inline`, you can direct the compiler to integrate that function's code into the code for its callers. This usually makes execution faster by eliminating the function-call overhead. In addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time, so that not all of the inline function's code needs to be included. The effect on code size is less predictable. Machine code may be larger or smaller with inline functions, depending on the particular case.

Note: Function inlining will only take place when the function's definition is visible at the call site (not just the prototype). In order to have a function inlined into more than one source file, the function definition may be placed into a header file that is included by each of the source files.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
    (*a)++;
}
```

(If you are using the `-traditional` option or the `-ansi` option, write `__inline` instead of `inline`.) You can also make all "simple enough" functions inline with the command-line option `-finline-functions`. The compiler heuristically decides which functions are simple enough to be worth integrating in this way, based on an estimate of the function's size.

Note: The `inline` keyword will only be recognized with `-finline` or optimizations enabled.

Certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of `varargs`, use of `alloca`, use of variable-sized data, use of computed `goto` and use of nonlocal `goto`. Using the command-line option `-Winline` will warn when a function marked `inline` could not be substituted, and will give the reason for the failure.

In compiler syntax, the `inline` keyword does not affect the linkage of the function.

When a function is both `inline` and `static`, if all calls to the function are integrated into the caller and the function's address is never used, then the function's own assembler code is never referenced. In this case, the compiler does not actually output assembler code for the function, unless you specify the command-line option `-fkeep-inline-functions`. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated and neither can recursive calls within the definition). If there is a non-integrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined. The compiler will only eliminate inline functions if they are declared to be static and if the function definition precedes all uses of the function.

When an `inline` function is not `static`, then the compiler must assume that there may be calls from other source files. Since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function and had not defined it.

This combination of `inline` and `extern` has a similar effect to a macro. Put a function definition in a header file with these keywords and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

Inline, like regular, is a suggestion and may be ignored.

13.7 MEMORY MODELS

The compiler supports several memory models. Command-line options are available for selecting the optimum memory model for your application, based on the specific dsPIC DSC device part that you are using and the type of memory usage.

TABLE 13-1: MEMORY MODEL COMMAND LINE OPTIONS

Option	Memory Definition	Description
<code>-msmall-data</code>	Up to 8 KB of data memory. This is the default.	Permits use of PIC18 like instructions for accessing data memory.
<code>-msmall-scalar</code>	Up to 8 KB of data memory. This is the default.	Permits use of PIC18 like instructions for accessing scalars in data memory.
<code>-mlarge-data</code>	Greater than 8 KB of data memory.	Uses indirection for data references.
<code>-msmall-code</code>	Up to 32 Kwords of program memory. This is the default.	Function pointers will not go through a jump table. Function calls use <code>RCALL</code> instruction.
<code>-mlarge-code</code>	Greater than 32 Kwords of program memory.	Function pointers might go through a jump table. Function calls use <code>CALL</code> instruction.
<code>-mconst-in-data</code>	Constants located in data memory.	Values copied from program memory by startup code.
<code>-mconst-in-code</code>	Constants located in program memory. This is the default.	Values are accessed via Program Space Visibility (PSV) data window.
<code>-mconst-in-aux-flash</code>	Constants in auxiliary FLASH	Values are accessed via Program Space visibility window.

The command-line options apply globally to the modules being compiled. Individual variables and functions can be declared as `near`, `far` or `eds` to better control the code generation. For information on setting individual variable or function attributes, see **Section 8.12 “Variable Attributes”** and **Section 13.2.1 “Function Specifiers”**.

13.7.1 Near or Far Code

Functions that are near (within a radius of 32 kWords of each other) may call each other more efficiently than those which are not. If it is known that all functions in an application are near, then the default `-msmall-code` command line option can be used when compiling each module to direct the compiler to use a more efficient form of the function call.

If this default option is not suitable, then the following alternatives are available:

1. It is possible to compile some modules of an application using the `-msmall-code` command line option. In this case, only function calls in those modules will use a more efficient form of the function call.
2. If the `-msmall-code` command-line option has been used, then the compiler may be directed to use the long form of the function call for an individual function by tagging it with the `far` attribute.
3. Instead of using command-line options, which have module scope, the compiler may be directed to call individual functions using a more efficient form of the function call by tagging their declaration and definition with the `near` attribute.
4. Group locally referent code together by using named sections or keep this code in common translation units.

The linker will produce an error message if the function declared to be near cannot be reached by one of its callers using a more efficient form of the function call.

13.8 FUNCTION CALL CONVENTIONS

When calling a function:

- Registers W0-W7 are caller saved. The calling function must preserve these values before the function call if their value is required subsequently from the function call. The stack is a good place to preserve these values.
- Registers W8-W14 are callee saved. The function being called must save any of these registers it will modify.
- Registers W0-W4 are used for function return values.
- Registers W0-W7 are used for argument transmission.
- DBRPAG/PSVPAG should be preserved if the `-mconst-in-code (auto_psv)` memory model is being used.

TABLE 13-2: REGISTERS REQUIRED

Data Type	Number of Working Registers Required
char	1
int	1
short	1
pointer	1
long	2 (contiguous – aligned to even numbered register)
float	2 (contiguous – aligned to even numbered register)
double*	2 (contiguous – aligned to even numbered register)
long double	4 (contiguous – aligned to quad numbered register)
structure	1 register per 2 bytes in structure
_Fract	1
long _Fract	2 (contiguous – aligned to even numbered register)
_Accum	3 (contiguous – aligned to quad numbered register)

* double is equivalent to long double if `-fno-short-double` is used.

Parameters are placed in the first aligned contiguous register(s) that are available. The calling function must preserve the parameters, if required. Structures do not have any alignment restrictions; a structure parameter will occupy registers if there are enough registers to hold the entire structure. Function results are stored in consecutive registers, beginning with W0.

13.8.1 Function Parameters

The first eight working registers (W0-W7) are used for function parameters. Parameters are allocated to registers in left-to-right order, and a parameter is assigned to the first available register that is suitably aligned.

In the following example, all parameters are passed in registers, although not in the order that they appear in the declaration. This format allows the compiler to make the most efficient use of the available parameter registers.

EXAMPLE 13-1: FUNCTION CALL MODEL

```
void
params0(short p0, long p1, int p2, char p3, float p4, void *p5)
{
    /*
    ** W0          p0
    ** W1          p2
    ** W3:W2       p1
    ** W4          p3
    ** W5          p5
    ** W7:W6       p4
    */
    ...
}
```

The next example demonstrates how structures are passed to functions. If the complete structure can fit in the available registers, then the structure is passed via registers; otherwise the structure argument will be placed onto the stack.

EXAMPLE 13-2: FUNCTION CALL MODEL, PASSING STRUCTURES

```
typedef struct bar {
    int i;
    double d;
} bar;

void
params1(int i, bar b) {
    /*
    ** W0          i
    ** W1          b.i
    ** W5:W2       b.d
    */

}
```

Parameters corresponding to the ellipses (...) of a variable-length argument list are not allocated to registers. Any parameter not allocated to registers is pushed onto the stack, in right-to-left order.

In the next example, the structure parameter cannot be placed in registers because it is too large. However, this does not prevent the next parameter from using a register spot.

EXAMPLE 13-3: FUNCTION CALL MODEL, STACK BASED ARGUMENTS

```
typedef struct bar {
    double d,e;
} bar;

void
params2(int i, bar b, int j) {
    /*
    ** W0          i
    ** stack      b
    ** W1          j
    */
}
```

Accessing arguments that have been placed onto the stack depends upon whether or not a Frame Pointer has been created. Generally the compiler will produce a Frame Pointer (unless told not to do so), and stack-based parameters will be accessed via the Frame Pointer register (W14). In the preceding example, *b* will be accessed from W14-22. The Frame Pointer offset of negative 22 has been calculated (refer to Figure 10-4) by removing 2 bytes for the previous FP, 4 bytes for the return address, followed by 16 bytes for *b*.

When no Frame Pointer is used, the assembly programmer must know how much stack space has been used since entry to the procedure. If no further stack space is used, the calculation is similar to Example 13-3. *b* would be accessed via W15-20; 4 bytes for the return address and 16 bytes to access the start of *b*.

13.8.2 Return Value

Function return values are returned in W0 for 8- or 16-bit scalars, W1:W0 for 32-bit scalars, and W3:W2:W1:W0 for 64-bit scalars. Aggregates are returned indirectly through W0, which is set up by the function caller to contain the address of the aggregate value.

13.8.3 Preserving Registers Across Function Calls

The compiler arranges for registers W8-W15 to be preserved across ordinary function calls. Registers W0-W7 are available as scratch registers. For interrupt functions, the compiler arranges for all necessary registers to be preserved, namely W0-W15 and RCOUNT.

Chapter 14. Interrupts

14.1 INTRODUCTION

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

This chapter presents an overview of interrupt processing. The following items are discussed:

- **Interrupt Operation** – An overview of how interrupts operate.
- **Writing an Interrupt Service Routine** – You can designate one or more C functions as Interrupt Service Routines (ISRs) to be invoked by the occurrence of an interrupt. For best performance in general, place lengthy calculations or operations that require library calls in the main application. This strategy optimizes performance and minimizes the possibility of losing information when interrupt events occur rapidly.
- **Specifying the Interrupt Vector** – The 16-bit devices use interrupt vectors to transfer application control when an interrupt occurs. An interrupt vector is a dedicated location in program memory that specifies the address of an ISR. Applications must contain valid function addresses in these locations to use interrupts.
- **Interrupt Service Routine Context Saving** – To handle returning from an interrupt to code in the same conditional state as before the interrupt, context information from specific registers must be saved.
- **Nesting Interrupts** – The time between when an interrupt is called and when the first ISR instruction is executed is the latency of the interrupt.
- **Enabling/Disabling Interrupts** – How interrupt priorities are determined. Enabling and disabling interrupt sources occurs at two levels: globally and individually.
- **ISR Considerations** – Sharing memory with mainline code, PSV usage with ISRs, and calling functions within ISRs.

14.2 INTERRUPT OPERATION

The compiler incorporates features allowing interrupts to be fully handled from C code. Interrupt functions are often called ISRs.

The 16-bit devices allow interrupts to be generated from many interrupt sources. Most sources have their own dedicated interrupt vector collated in an interrupt vector table (IVT). Each vector consists of an address at which is found the entry point of the interrupt service routine. Some of the interrupt table vector locations are for traps, which are non-maskable interrupts which deal with erroneous operation of the device, such as an address error.

On some devices, an alternate interrupt vector table (AIVT) is provided, which allow independent interrupt vectors to be specified. This table can be enabled when required, forcing ISR addresses to be read from the AIVT rather than the IVT.

Interrupts have a priority associated with them. This can be independently adjusted for each interrupt source. When more than interrupt with the same priority are pending at the same time, the intrinsic priority, or natural order priority, of each source comes into play. The natural order priority is typically the same as the order of the interrupt vectors in the IVT.

The compiler provides full support for interrupt processing in C or inline assembly code.

Interrupt code is the name given to any code that executes as a result of an interrupt occurring. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed.

This contrasts with main-line code, which, for a freestanding application, is usually the main part of the program that executes after Reset.

14.3 WRITING AN INTERRUPT SERVICE ROUTINE

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

All ISR code will be placed into a named section that starts with `.isr`. A function with a `section` attribute will prepend `.isr` to the name given. Code compiled with `-ffunction-sections` will also prepend `.isr` to the section name.

If you have created your own linker script file, and that file is older than an MPLAB C30 v3.30 project, you will need to modify your linker script as per the `Readme_XC16.html` file found in the `docs` subdirectory of the MPLAB XC16 install directory.

14.3.1 Guidelines for Writing ISRs

The following guidelines are suggested for writing ISRs:

- declare ISRs with no parameters and a `void` return type (mandatory)
- do not let ISRs be called by main line code (mandatory)
- do not let ISRs call other functions (recommended)

A 16-bit device ISR is like any other C function in that it can have local variables and access global variables. However, an ISR needs to be declared with no parameters and no return value. This is necessary because the ISR, in response to a hardware interrupt or trap, is invoked asynchronously to the mainline C program (that is, it is not called in the normal way, so parameters and return values don't apply).

ISRs should only be invoked through a hardware interrupt or trap and not from other C functions. An ISR uses the return from interrupt (`RETFIE`) instruction to exit from the function rather than the normal `RETURN` instruction. Using a `RETFIE` instruction out of context can corrupt processor resources, such as the Status register.

Finally, ISRs should avoid calling other functions. This is recommended because of latency issues. See **Section 14.6 “Nesting Interrupts”** for more information.

14.3.2 Syntax for Writing ISRs

To declare a C function as an interrupt handler, tag the function with the interrupt attribute (see **Section 13.2.2 “Function Attributes”** for a description of the `__attribute__` keyword). The syntax of the interrupt attribute is:

```
__attribute__((interrupt [ (
    [ save(symbol-list) ]
    [, irq(irqid) ]
    [, altirq(altirqid) ]
    [, preprologue(asm) ]
    ) ]
    ) )
```

The `interrupt` attribute name and the parameter names may be written with a pair of underscore characters before and after the name. Thus, `interrupt` and `__interrupt__` are equivalent, as are `save` and `__save__`.

The optional `save` parameter names a list of one or more variables that are to be saved and restored on entry to and exit from the ISR. The list of names is written inside parentheses, with the names separated by commas.

You should arrange to save global variables that may be modified in an ISR if you do not want the value to be exported. Global variables accessed by an ISR should be qualified `volatile`.

The optional `irq` parameter allows you to place an interrupt vector at a specific interrupt, and the optional `altirq` parameter allows you to place an interrupt vector at a specified alternate interrupt. Each parameter requires a parenthesized interrupt ID

number. (See **Section 14.4 “Specifying the Interrupt Vector”** for a list of interrupt IDs.)

The optional `preprologue` parameter allows you to insert assembly-language statements into the generated code immediately before the compiler-generated function prologue.

When using the `interrupt` attribute, please specify either `auto_psv` or `no_auto_psv`. If none is specified a warning will be produced and `auto_psv` will be assumed.

14.3.3 Coding ISRs

The following prototype declares function `isr0` to be an interrupt handler:

```
void __attribute__((__interrupt__, __auto_psv__)) isr0(void);
```

As this prototype indicates, interrupt functions must not take parameters nor may they return a value. The compiler arranges for all working registers to be preserved, as well as the Status register and the Repeat Count register, if necessary. Other variables may be saved by naming them as parameters of the `interrupt` attribute. For example, to have the compiler automatically save and restore the variables, `var1` and `var2`, use the following prototype:

```
void __attribute__((__interrupt__, __auto_psv__,  
    (__save__(var1, var2)))) isr0(void);
```

To request the compiler to use the fast context save (using the `push.s` and `pop.s` instructions), tag the function with the `shadow` attribute (see **Section 13.2.1 “Function Specifiers”**). For example:

```
void __attribute__((__interrupt__, __auto_psv__, __shadow__))  
    isr0(void);
```

14.3.4 Using Macros to Declare Simple ISRs

If an interrupt handler does not require any of the optional parameters of the `interrupt` attribute, then a simplified syntax may be used. The following macros are defined in the device-specific header files:

```
#define _ISR __attribute__((interrupt))  
#define _ISRFAST __attribute__((interrupt, shadow))
```

For example, to declare an interrupt handler for external interrupt 0:

```
#include <xc.h>  
void _ISR _INT0Interrupt(void);
```

To declare an interrupt handler for the SPI1 interrupt with fast context save:

```
#include <xc.h>  
void _ISRFAST _SPI1Interrupt(void);
```

14.4 SPECIFYING THE INTERRUPT VECTOR

Many 16-bit devices have two interrupt vector tables – a primary and an alternate table – each containing several exception vectors.

The exception sources have associated with them a primary and alternate exception vector, each occupying a program word, as shown in the tables below. The alternate vector name is used when the `ALTIVT` bit is set in the `INTCON2` register.

Note: A device Reset is not handled through the interrupt vector table. Instead, on device Reset, the program counter is cleared. This causes the processor to begin execution at address zero. By convention, the linker script constructs a `GOTO` instruction at that location which transfers control to the C run-time startup module.

To field an interrupt, a function's address must be placed at the appropriate address in one of the vector tables, and the function must preserve any system resources that it uses. It must return to the foreground task using a `RETFIE` processor instruction. Interrupt functions may be written in C. When a C function is designated as an interrupt handler, the compiler arranges to preserve all the system resources that the compiler uses, and to return from the function using the appropriate instruction. The compiler can optionally arrange for the interrupt vector table to be populated with the interrupt function's address.

To arrange for the compiler to fill in the interrupt vector to point to the interrupt function, name the function as denoted in the preceding table. For example, the stack error vector will automatically be filled if the following function is defined:

```
void __attribute__((__interrupt__,__auto_psv__)) _StackError(void);
```

Note the use of the leading underscore. Similarly, the alternate stack error vector will automatically be filled if the following function is defined:

```
void __attribute__((__interrupt__,__auto_psv__))
    _AltStackError(void);
```

Again, note the use of the leading underscore.

For all interrupt vectors without specific handlers, a default interrupt handler will be installed. The default interrupt handler is supplied by the linker and simply resets the device. An application may also provide a default interrupt handler by declaring an interrupt function with the name `_DefaultInterrupt`.

The last nine interrupt vectors in each table do not have predefined hardware functions. The vectors for these interrupts may be filled by using the names indicated in the preceding table, or, names more appropriate to the application may be used, while still filling the appropriate vector entry by using the `irq` or `altirq` parameter of the interrupt attribute. For example, to specify that a function should use primary interrupt vector 52, use the following:

```
void __attribute__((__interrupt__,__auto_psv__,__irq__(52)))
    MyIRQ(void);
```

Similarly, to specify that a function should use alternate interrupt vector 52, use the following:

```
void __attribute__((__interrupt__,__auto_psv__,__altirq__(52)))
    MyAltIRQ(void);
```

The `irq/altirq` number can be one of the interrupt request numbers 45 to 53. If the `irq` parameter of the interrupt attribute is used, the compiler creates the external symbol name `__Interruptn`, where `n` is the vector number. Therefore, the C identifiers `_Interrupt45` through `_Interrupt53` are reserved by the compiler. In the same way, if the `altirq` parameter of the interrupt attribute is used, the compiler creates the external symbol name `_AltInterruptn`, where `n` is the vector number. Therefore, the C identifiers `_AltInterrupt45` through `_AltInterrupt53` are reserved by the compiler.

For tables of interrupt vectors by device family:

- In MPLAB X IDE, for newer versions of the compiler, open the Dashboard window and click on the **Compiler Help** button.
- On the command-line, see the `docs` subdirectory of the MPLAB XC16 C compiler install directory (**Section 4.2 “MPLAB X IDE and Tools Installation”**). Open the `XC16MasterIndex` file and click on the “Interrupt Vector Tables Reference” link.

14.5 INTERRUPT SERVICE ROUTINE CONTEXT SAVING

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

To properly handle a return from interrupt, the setup (prologue) code for an ISR function automatically saves the compiler-managed working and special function registers on the stack for later restoration at the end of the ISR. You can use the optional `save` parameter of the `interrupt` attribute to specify additional variables and SFRs to be saved and restored.

In certain applications, it may be necessary to insert assembly statements into the ISR immediately prior to the compiler-generated function prologue. For example, it may be required that a semaphore be incremented immediately on entry to an interrupt service routine. This can be done as follows:

```
void __attribute__((__interrupt__,__auto_psv__,__preprologue__  
    ("inc _semaphore")))) isr0(void);
```

The context switch leads to latency in interrupt code execution, as described in **Section 14.8.3 “Latency”**.

14.6 NESTING INTERRUPTS

The 16-bit devices support nested interrupts. Since processor resources are saved on the stack in an ISR, nested ISRs are coded in just the same way as non-nested ones. Nested interrupts are enabled by clearing the NSTDIS (nested interrupt disable) bit in the INTCON1 register. Note that this is the default condition as the 16-bit device comes out of Reset with nested interrupts enabled. Each interrupt source is assigned a priority in the Interrupt Priority Control registers (IPCn).

An interrupt is vectored if the priority of the interrupt source is greater than the current CPU priority level.

14.7 ENABLING/DISABLING INTERRUPTS

Each interrupt source can be individually enabled or disabled. One interrupt enable bit for each IRQ is allocated in the Interrupt Enable Control registers (IECn). Setting an interrupt enable bit to one (1) enables the corresponding interrupt; clearing the interrupt enable bit to zero (0) disables the corresponding interrupt. When the device comes out of Reset, all interrupt enable bits are cleared to zero. In addition, the processor has a disable interrupt instruction (DISI) that can disable all interrupts for a specified number of instruction cycles.

Note: Traps, such as the address error trap, cannot be disabled. Only IRQs can be disabled.

The DISI instruction can be used in a C program through the use of:

```
__builtin_disi
```

For example:

```
__builtin_disi(16);
```

will emit the specified DISI instruction at the point it appears in the source program. A disadvantage of using DISI in this way is that the C programmer cannot always be sure how the C compiler will translate C source to machine instructions, so it may be difficult to determine the cycle count for the DISI instruction. It is possible to get around this difficulty by bracketing the code that is to be protected from interrupts by DISI instructions, the first of which sets the cycle count to the maximum value, and the second of which sets the cycle count to zero. For example,

```
__builtin_disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
__builtin_disi(0x0000); /* enable interrupts */
```

An alternative approach is to write directly to the DISICNT register to enable interrupts. The DISICNT register may be modified only after a DISI instruction has been issued and if the contents of the DISICNT register are not zero.

```
__builtin_disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

For some applications, it may be necessary to disable level 7 interrupts as well. These can only be disabled through the modification of the COROCON IPL field. The provided support files contain some useful preprocessor macro functions to help you safely modify the IPL value. These macros are:

```
SET_CPU_IPL(ipl)
SET_AND_SAVE_CPU_IPL(save_to, ipl)
RESTORE_CPU_IPL(saved_to)
```

For example, you may wish to protect a section of code from interrupt. The following code will adjust the current IPL setting and restore the IPL to its previous value.

```
void foo(void) {
    int current_cpu_ipl;

    SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7); /* disable interrupts */
    /* protected code here */
    RESTORE_CPU_IPL(current_cpu_ipl);
}
```

14.8 ISR CONSIDERATIONS

The following sections describe how to ensure your interrupt code works as expected.

14.8.1 Sharing Memory with Mainline Code

Exercise caution when modifying the same variable within a main or low-priority ISR and a high-priority ISR. Higher priority interrupts, when enabled, can interrupt a multiple instruction sequence and yield unexpected results when a low-priority function has created a multiple instruction Read-Modify-Write sequence accessing that same variable. Therefore, embedded systems must implement an “atomic” operation to ensure that the intervening high-priority ISR will not write to the variable from which the low-priority ISR has just read, but not yet completed its write.

An atomic operation is one that cannot be broken down into its constituent parts – it cannot be interrupted. Not all C expressions translate into an atomic operation. On dsPIC DSC devices, these expressions mainly fall into the following categories: 32-bit expressions, floating point arithmetic, division, operations on multi-bit bit-fields, and fixed point operations. Other factors will determine whether or not an atomic operation will be generated, such as memory model settings, optimization level and resource availability. In other words, C does not guarantee atomicity of operations.

Consider the general expression:

```
foo = bar op baz;
```

The operator (`op`) may or may not be atomic, based on the architecture of the device. In any event, the compiler may not be able to generate the atomic operation in all instances, depending on factors that may include the following:

- availability of an appropriate atomic machine instruction
- resource availability - special registers or other constraints
- optimization level, and other options that affect data/code placement

Without knowledge of the architecture, it is reasonable to assume that the general expression requires two reads, one for each operand and one write to store the result. Several difficulties may arise in the presence of interrupt sequences, depending on the particular application.

14.8.1.1 DEVELOPMENT ISSUES

Here are some examples of the issues that should be considered:

EXAMPLE 14-1: **bar** MUST MATCH **baz**

When it is required that `bar` and `baz` match (i.e., are updated synchronously with each other), there is a possible hazard if either `bar` or `baz` can be updated within a higher priority interrupt expression. Here are some sample flow sequences:

1. Safe:
 read `bar`
 read `baz`
 perform operation
 write back result to `foo`
2. Unsafe:
 read `bar`
 interrupt modifies `baz`
 read `baz`
 perform operation
 write back result to `foo`
3. Safe:
 read `bar`
 read `baz`
 interrupt modifies `bar` or `baz`
 perform operation
 write back result to `foo`

The first is safe because any interrupt falls outside the boundaries of the expression. The second is unsafe because the application demands that `bar` and `baz` be updated synchronously with each other. The third is probably safe; `foo` will possibly have an old value, but the value will be consistent with the data that was available at the start of the expression.

EXAMPLE 14-2: TYPE OF `foo`, `bar` AND `baz`

Another variation depends upon the type of `foo`, `bar` and `baz`. The operations, “read `bar`”, “read `baz`”, or “write back result to `foo`”, may not be atomic, depending upon the architecture of the target processor. For example, dsPIC DSC devices can read or write an 8-bit, 16-bit, or 32-bit quantity in 1 (atomic) instruction. But, a 32-bit quantity may require two instructions depending upon instruction selection (which in turn will depend upon optimization and memory model settings). Assume that the types are `long` and the compiler is unable to choose atomic operations for accessing the data. Then the access becomes:

```
read lsw bar
read msw bar
read lsw baz
read msw baz
perform operation (on lsw and on msw)
perform operation
write back lsw result to foo
write back msw result to foo
```

Now there are more possibilities for an update of `bar` or `baz` to cause unexpected data.

EXAMPLE 14-3: BIT-FIELDS

A third cause for concern are bit-fields. C allows memory to be allocated at the bit level, but does not define any bit operations. In the purest sense, any operation on a bit will be treated as an operation on the underlying type of the bit-field and will usually require some operations to extract the field from `bar` and `baz` or to insert the field into `foo`. The important consideration to note is that (again depending upon instruction architecture, optimization levels and memory settings) an interrupted routine that writes to any portion of the bit-field where `foo` resides may be corruptible. This is particularly apparent in the case where one of the operands is also the destination.

The dsPIC DSC instruction set can operate on 1 bit atomically. The compiler may select these instructions depending upon optimization level, memory settings and resource availability.

EXAMPLE 14-4: CACHED MEMORY VALUES IN REGISTERS

Finally, the compiler may choose to cache memory values in registers. These are often referred to as register variables and are particularly prone to interrupt corruption, even when an operation involving the variable is not being interrupted. Ensure that memory resources shared between an ISR and an interruptible function are designated as `volatile`. This will inform the compiler that the memory location may be updated out-of-line from the serial code sequence. This will not protect against the effect of non-atomic operations, but is never-the-less important.

14.8.1.2 DEVELOPMENT SOLUTIONS

Here are some strategies to remove potential hazards:

- Design the software system such that the conflicting event cannot occur. Do not share memory between ISRs and other functions. Make ISRs as simple as possible and move the real work to main code.
- Use care when sharing memory and, if possible, avoid sharing bit-fields which contain multiple bits.
- Protect non-atomic updates of shared memory from interrupts as you would protect critical sections of code. The following macro can be used for this purpose:

```
#define INTERRUPT_PROTECT(x) { \
    char saved_ipl; \
    \
    SET_AND_SAVE_CPU_IPL(saved_ipl,7); \
    x; \
    RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

This macro disables interrupts by increasing the current priority level to 7, performing the desired statement and then restoring the previous priority level.

14.8.1.3 APPLICATION EXAMPLE

The following example highlights some of the points discussed in this section:

```
void __attribute__((interrupt))
HigherPriorityInterrupt(void) {
    /* User Code Here */
    LATGbits.LATG15 = 1; /* Set LATG bit 15 */
    IPC0bits.INT0IP = 2; /* Set Interrupt 0
                          priority (multiple
                          bits involved) to 2 */
}

int main(void) {
    /* More User Code */
    LATGbits.LATG10 ^= 1; /* Potential HAZARD -
                          First reads LATG into a W reg,
                          implements XOR operation,
                          then writes result to LATG */

    LATG = 0x1238;        /* No problem, this is a write
                          only assignment operation */

    LATGbits.LATG5 = 1;   /* No problem likely,
                          this is an assignment of a
                          single bit and will use a single
                          instruction bit set operation */

    LATGbits.LATG2 = 0;   /* No problem likely,
                          single instruction bit clear
                          operation probably used */

    LATG += 0x0001;       /* Potential HAZARD -
                          First reads LATG into a W reg,
                          implements add operation,
                          then writes result to LATG */

    IPC0bits.T1IP = 5;    /* HAZARD -
                          Assigning a multiple bitfield
                          can generate a multiple
                          instruction sequence */
}
```

A statement can be protected from interrupt using the `INTERRUPT_PROTECT` macro provided above. For this example:

```
INTERRUPT_PROTECT(LATGbits.LATG15 ^= 1); /* Not interruptible by
                                          level 1-7 interrupt
                                          requests and safe
                                          at any optimization
                                          level */
```

14.8.2 PSV Usage with Interrupt Service Routines

The introduction of managed psv pointers and CodeGuard Security psv constant sections in compiler v3.0 means that ISRs cannot make any assumptions about the setting of PSVPAG. This is a migration issue for existing applications with ISRs that reference the `auto_psv` constants section. In previous versions of the compiler, the ISR could assume that the correct value of PSVPAG was set during program startup (unless the programmer had explicitly changed it.)

To help mitigate this problem, two new function attributes will be introduced: `auto_psv` and `no_auto_psv`. If an ISR references const variables or string literals using the `constants-in-code` memory model, the `auto_psv` attribute should be added to the function definition. This attribute will cause the compiler to preserve the previous contents of PSVPAG and set it to section `.const`. Upon exit, the previous value of PSVPAG will be restored. For example:

```
void __attribute__((interrupt, auto_psv)) myISR()
{
    /* This function can reference const variables and
       string literals with the constants-in-code memory model. */
}
```

The `no_auto_psv` attribute is used to indicate that an ISR does not reference the `auto_psv` constants section. If neither attribute is specified, the compiler assumes `auto_psv` and inserts the necessary instructions to ensure correct operation at run time. A warning diagnostic message is also issued that alerts the user to the migration issue, and to the possibility of reducing interrupt latency by specifying the `no_auto_psv` attribute.

14.8.3 Latency

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

- **Processor Servicing of Interrupt** – the amount of time it takes the processor to recognize the interrupt and branch to the first address of the interrupt vector. To determine this value refer to the processor data sheet for the specific processor and interrupt source being used.
- **ISR Code** – although an interrupt function may call other functions, whether they be user-defined functions, library functions or implicitly called functions to implement a C operation, the compiler cannot know, in general, which resources are used by the called function. As a result, the compiler will save all the working registers and RCOUNT, even if they are not all used explicitly in the ISR itself. The increased latency associated with the call does not lend itself to fast response times.

Chapter 15. Main, Runtime Startup and Reset

15.1 INTRODUCTION

When creating C code, there are elements that are required to ensure proper program operation: a `main` function must be present; startup code to initialize and clear variables, to set up registers and the processor; and Reset conditions need to be handled. The following topics are discussed in this section:

- The main Function
- Runtime Startup and Initialization

15.2 THE `main` FUNCTION

The identifier `main` is special. It must be used as the name of a function that will be the first function to execute in a program. You must always have one and only one function called `main()` in your programs. Code associated with `main()`, however, is not the first code to execute after Reset. Additional code provided by the compiler and known as the runtime startup code is executed first and is responsible for transferring control to the `main()` function.

The prototype that should be used for `main()` is as follows.

```
int main(void);
```

15.3 RUNTIME STARTUP AND INITIALIZATION

A C program requires certain objects to be initialized and the processor to be in a particular state before it can begin execution of its function `main()`. It is the job of the *runtime startup* code to perform these tasks, specifically (and in no particular order):

- Initialization of global variables assigned a value when defined
- Initialization of the stack
- Clearing of non-initialized global variables
- General setup of registers or processor state

Two C run-time startup modules are included in the `libpic30-omf.a` archive/library. The entry point for both startup modules is `__reset`. The linker scripts construct a `GOTO __reset` instruction at location 0 in program memory, which transfers control upon device Reset.

The primary startup module is linked by default and performs the following:

1. The Stack Pointer (W15) and Stack Pointer Limit register (SPLIM) are initialized, using values provided by the linker or a custom linker script. For more information, see **Section 6.4 “Stack”**.
2. If a `.const` section is defined, it is mapped into the program space visibility window by initializing the PSV page and CORCON registers, as appropriate, if `const-in-code` memory mode is used or variables have been explicitly allocated to `space(auto_psv)`.

3. The data initialization template is read, causing all uninitialized objects to be cleared, and all initialized objects to be initialized with values read from program memory. The data initialization template is created by the linker.

Note: Persistent data is never cleared or initialized.

4. If the application has defined `user_init` functions (see **Section 13.2.2 “Function Attributes”**), these are invoked. The order of execution depends on link order.
5. The function `main()` is called with no parameters.
6. If `main()` returns, the processor will reset.

The alternate startup module is linked when the `-Wl, --no-data-init` option is specified. It performs the same operations, except for step (3), which is omitted. The alternate startup module is smaller than the primary module, and can be selected to conserve program memory if data initialization is not required.

Zippped source code (in dsPIC DSC assembly language) for both modules is provided in the `<xc16 install directory>\src\libpic30.zip`. The startup modules may be modified if necessary. For example, if an application requires `main` to be called with parameters, a conditional assembly directive may be changed to provide this support.

Chapter 16. Mixing C and Assembly Code

16.1 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.

Items discussed are:

- **Mixing Assembly Language and C Variables and Functions** – separate assembly language modules may be assembled, then linked with compiled C modules.
- **Using Inline Assembly Language** – assembly language instructions may be embedded directly into the C code. The inline assembler supports both simple (non-parameterized) assembly language statement, as well as extended (parameterized) statements (where C variables can be accessed as operands of an assembler instruction).
- **Predefined Assembly Macros** – a list of predefined assembly-code macros to be used in C code is provided.

16.2 MIXING ASSEMBLY LANGUAGE AND C VARIABLES AND FUNCTIONS

The following guidelines indicate how to interface separate assembly language modules with C modules.

- Follow the register conventions described in **12.2 “Register Variables”**. In particular, registers W0-W7 are used for parameter passing. An assembly language function will receive parameters, and should pass arguments to called functions, in these registers.
- Functions not called during interrupt handling must preserve registers W8-W15. That is, the values in these registers must be saved before they are modified and restored before returning to the calling function. Registers W0-W7 may be used without restoring their values.
- Interrupt functions must preserve all registers. Unlike a normal function call, an interrupt may occur at any point during the execution of a program. When returning to the normal program, all registers must be as they were before the interrupt occurred.
- Variables or functions declared within a separate assembly file that will be referenced by any C source file should be declared as global using the assembler directive `.global`. External symbols should be preceded by at least one underscore. The C function `main` is named `_main` in assembly and conversely an assembly symbol `_do_something` will be referenced in C as `do_something`. Undeclared symbols used in assembly files will be treated as externally defined.

The following example shows how to use variables and functions in both assembly language and C regardless of where they were originally defined.

The file `ex1.c` defines `foo` and `cVariable` to be used in the assembly language file. The C file also shows how to call an assembly function, `asmFunction`, and how to access the assembly defined variable, `asmVariable`.

Examples in this Section:

- Mixing C and Assembly
- Calling an Assembly Function in C

EXAMPLE 16-1: MIXING C AND ASSEMBLY

```
/*
** file: ex1.c
*/
extern unsigned int asmVariable;
extern void asmFunction(void);
unsigned int cVariable;
void foo(void)
{
    asmFunction();
    asmVariable = 0x1234;
}
```

The file `ex2.s` defines `asmFunction` and `asmVariable` as required for use in a linked application. The assembly file also shows how to call a C function, `foo`, and how to access a C defined variable, `cVariable`.

```
;
; file: ex2.s
;
    .text
    .global _asmFunction
_asmFunction:
    mov #0,w0
    mov w0,_cVariable
    return

    .global _main
_main:
    call _foo
    return

    .bss
    .global _asmVariable
    .align 2
_asmVariable: .space 2
.end
```

In the C file, `ex1.c`, external references to symbols declared in an assembly file are declared using the standard `extern` keyword; note that `asmFunction`, or `_asmFunction` in the assembly source, is a `void` function and is declared accordingly.

In the assembly file, `ex1.s`, the symbols `_asmFunction`, `_main` and `_asmVariable` are made globally visible through the use of the `.global` assembler directive and can be accessed by any other source file. The symbol `_main` is only referenced and not declared; therefore, the assembler takes this to be an external reference.

The following compiler example shows how to call an assembly function with two parameters. The C function `main` in `call1.c` calls the `asmFunction` in `call2.s` with two parameters.

EXAMPLE 16-2: CALLING AN ASSEMBLY FUNCTION IN C

```
/*
** file: call1.c
*/
extern int asmFunction(int, int);
int x;
void
main(void)
{
    x = asmFunction(0x100, 0x200);
}
```

The assembly-language function sums its two parameters and returns the result.

```
;
; file: call2.s
;
.global _asmFunction
_asmFunction:
    add w0,w1,w0
    return
.end
```

Parameter passing in C is detailed in **Section 13.8.2 “Return Value”**. In the preceding example, the two integer arguments are passed in the W0 and W1 registers. The integer return result is transferred via register W0. More complicated parameter lists may require different registers and care should be taken in the hand-written assembly to follow the guidelines.

16.3 USING INLINE ASSEMBLY LANGUAGE

Within a C function, the `asm` statement may be used to insert a line of assembly language code into the assembly language that the compiler generates. Inline assembly has two forms: simple and extended.

In the **simple** form, the assembler instruction is written using the syntax:

```
asm ("instruction");
```

where *instruction* is a valid assembly-language construct. If you are writing inline assembly in ANSI C programs, write `__asm__` instead of `asm`.

Note: Only a single string can be passed to the simple form of inline assembly.

In an **extended** assembler instruction using `asm`, the operands of the instruction are specified using C expressions. The extended syntax is:

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
               [ : [ "constraint"(input-operand) [ , ... ] ]
               [ "clobber" [ , ... ] ]
           ]
    );
```

You must specify an assembler instruction *template*, plus an operand *constraint* string for each operand. The *template* specifies the instruction mnemonic, and optionally placeholders for the operands. The *constraint* strings specify operand constraints, for example, that an operand must be in a register (the usual case), or that an operand must be an immediate value.

Constraint letters and modifiers supported by the compiler are listed in Table 16-1 and Table respectively.

TABLE 16-1: CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
a	Claims WREG
b	Divide support register W1
c	Multiply support register W2
d	General purpose data registers W1-W14
e	Non-divide support registers W2-W14
g	Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.
i	An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time.
r	A register operand is allowed provided that it is in a general register.
v	AWB register W13
w	Accumulator register A-B
x	x prefetch registers W8-W9
y	y prefetch registers W10-W11
z	MAC prefetch registers W4-W7
0, 1, ... , 9	An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last. By default, <code>%n</code> represents the first register for the operand (<i>n</i>). To access the second, third, or fourth register, use a modifier letter.
T	A near or far data operand.
U	A near data operand.

TABLE 16-2: CONSTRAINT MODIFIERS SUPPORTED BY THE COMPILER

Modifier	Constraint
=	Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.
+	Means that this operand is both read and written by the instruction.
&	Means that this operand is an <i>earlyclobber</i> operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.
d	Second register for operand number <i>n</i> , i.e., %dn..
q	Fourth register for operand number <i>n</i> , i.e., %qn..
t	Third register for operand number <i>n</i> , i.e., %tn..

Examples in this Section:

- Passing C Variables
- Clobbering Registers
- Using Multiple Assembler Instructions
- Using ‘&’ to Prevent Input Register Clobbering
- Matching Operands
- Naming Operands
- Volatile asm Statements
- Handling Values Larger Than int

EXAMPLE 16-3: PASSING C VARIABLES

This example demonstrates how to use the `swap` instruction (which the compiler does not generally use):

```
asm ("swap %0" : "+r"(var));
```

Here `var` is the C expression for the operand, which is both an input and an output operand. The operand is constrained to be of type `r`, which denotes a register operand. The `+` in `+r` indicates that the operand is both an input and output operand.

Each operand is described by an operand-constraint string that is followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs.

If there are no output operands, but there are input operands; then, there must be two consecutive colons surrounding the place where the output operands would go. The compiler requires that the output operand expressions must be L-values. The input operands need not be L-values. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that the compiler itself does not know exist. If the output expression cannot be directly addressed (for example, it is a bit-field), the constraint must allow a register. In that case, the compiler will use the register as the output of the `asm`, and then store that register into the output. If output operands are write-only, the compiler will assume that the values in these operands before the instruction are dead and need not be generated.

EXAMPLE 16-4: CLOBBERING REGISTERS

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings separated by commas). Here is an example:

```
asm volatile ("mul.b %0"
: /* no outputs */
: "U" (nvar)
: "w2");
```

In this case, the operand `nvar` is a character variable declared in near data space, as specified by the “U” constraint. If the assembler instruction can alter the flags (condition code) register, add “cc” to the list of clobbered registers. If the assembler instruction modifies memory in an unpredictable fashion, add “memory” to the list of clobbered registers. This will cause the compiler to not keep memory values cached in registers across the assembler instruction.

EXAMPLE 16-5: USING MULTIPLE ASSEMBLER INSTRUCTIONS

You can put multiple assembler instructions together in a single `asm` template, separated with newlines (written as `\n`). The input operands and the output operands' addresses are ensured not to use any of the clobbered registers, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine `_foo` accepts arguments in registers W0 and W1:

```
asm ("mov %0,w0\nmov %1,w1\ncall _foo"
: /* no outputs */
: "g" (a), "g" (b)
: "W0", "W1");
```

In this example, the constraint strings “g” indicate a general operand.

EXAMPLE 16-6: USING ‘&’ TO PREVENT INPUT REGISTER CLOBBERING

Unless an output operand has the `&` constraint modifier, the compiler may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&` for each output operand that may not overlap an input operand. For example, consider the following function:

```
int
exprbad(int a, int b)
{
    int c;

    __asm__ ("add %1,%2,%0\n sl %0,%1,%0"
: "=r" (c) : "r" (a), "r" (b));

    return(c);
}
```

The intention is to compute the value $(a + b) \ll a$. However, as written, the value computed may or may not be this value. The correct coding informs the compiler that the operand `c` is modified before the `asm` instruction is finished using the input operands, as follows:

```
int
exprgood(int a, int b)
{
    int c;
```

```
__asm__ ("add %1,%2,%0\n sl %0,%1,%0"
: "=&r" (c) : "r" (a), "r" (b));

return(c);
}
```

EXAMPLE 16-7: MATCHING OPERANDS

When the assembler instruction has a read-write operand, or an operand in which only some of the bits are to be changed, you must logically split its function into two separate operands: one input operand and one write-only output operand. The connection between them is expressed by constraints that say they need to be in the same location when the instruction executes. You can use the same C expression for both operands or different expressions. For example, here is the `add` instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "0" (foo), "r" (bar));
```

The constraint "0" for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand and must refer to an output operand. Only a digit in the constraint can ensure that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to ensure that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "r" (foo), "r" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address).

EXAMPLE 16-8: NAMING OPERANDS

It is also possible to specify input and output operands using symbolic names that can be referenced within the assembler code template. These names are specified inside square brackets preceding the constraint string, and can be referenced inside the assembler code template using `%[name]` instead of a percentage sign followed by the operand number. Using named operands, the above example could be coded as follows:

```
asm ("add %[foo],[bar],[foo]"
: [foo] "=r" (foo)
: "0" (foo), [bar] "r" (bar));
```

EXAMPLE 16-9: VOLATILE ASM STATEMENTS

You can prevent an `asm` instruction from being deleted, moved significantly, or combined, by writing the keyword `volatile` after the `asm`. For example:

```
#define disi(n) \
asm volatile ("disi #0" \
: /* no outputs */ \
: "i" (n))
```

In this case, the constraint letter "i" denotes an immediate operand, as required by the `disi` instruction.

EXAMPLE 16-10: HANDLING VALUES LARGER THAN INT

Constraint letters and modifiers may be used to identify various entities with which it is acceptable to replace a particular operand, such as %0 in:

```
asm("mov %1, %0" : "r"(foo) : "r"(bar));
```

This example indicates that the value stored in `foo` should be moved into `bar`. The example code performs this task unless `foo` or `bar` are larger than an `int`.

By default, %0 represents the first register for the operand (0). To access the second, third, or fourth register, use a modifier letter specified in Table .

16.4 PREDEFINED ASSEMBLY MACROS

Some macros used to insert assembly code in C are defined once you include `<xc.h>`. The macros are: `Nop()`, `ClrWdt()`, `Sleep()` and `Idle()`. The latter two insert the `PWRSABV` instruction with an argument of #0 and #1, respectively.

Chapter 17. Library Routines

Many library functions or routines (and any associated variables) will be automatically linked into a program once they have been referenced in your source code. The use of a function from one library file will not include any other functions from that library. Only used library functions will be linked into the program output and consume memory.

Library and precompiled object files are stored in the compiler's installation directory structure.

Your program will require declarations for any functions or symbols used from libraries. These are contained in the standard C header (.h) files. Header files are not library files and the two files types should not be confused. Library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros.

The `include` directories, under the compiler's installation directory, are where the compiler stores the standard C library system header files. The installation will automatically locate its bundled include files.

Some libraries require manual inclusion in your project, or require special options to use. See the *16-Bit Language Tools Libraries* (DS51456) for questions about particular libraries.

Libraries which are found automatically include:

- Standard C library
- dsPIC30 support libraries
- Standard IEEE floating point library
- Fixed point library
- Device peripheral library

EXAMPLE 17-1: USING THE MATH LIBRARY

```
#include <math.h>    // declare function prototype for sqrt

void main(void)
{
    double i;

    // sqrt referenced; sqrt will be linked in from library file
    i = sqrt(23.5);
}
```

NOTES:

Chapter 18. Optimizations

18.1 INTRODUCTION

Different MPLAB XC16 C Compiler editions support different levels of optimization. Some editions are free to download and others must be purchased.

The compiler is available in the following editions:

Edition	Cost	Description
Professional (PRO)	Yes	Implemented with the highest optimizations and performance levels.
Standard (STD)	Yes	Implemented with ample optimizations levels and high performance levels.
Free	No	Implemented with the most restrictions on code optimizations.
Evaluation (EVAL)	No	PRO edition enabled for 60 days; afterward reverts to Free edition.

18.2 SETTING OPTIMIZATION LEVELS

Here are the optimizations for different editions of the compiler, by O-level and GCC optimization:

- PRO: -O3, -Os, -mpa (Also allows usage of Free and STD optimizations)
- STD: -O2 (Also allows usage of Free optimizations)
- Free: -O0, -O1

Different optimizations may be set ranging from no optimization to full optimization, depending on your compiler edition. When debugging code, you may prefer not to optimize your code to ensure expected program flow.

For details on compiler options used to set optimizations, see **Section 5.7.6 “Options for Controlling Optimization”**.

18.3 OPTIMIZATION FEATURE SUMMARY

The optimization level available for each edition equates to the features specified in the table below.

TABLE 18-1: OPTIMIZATION EDITION SUPPORTED FEATURES

Free	STD	PRO
<ul style="list-style-type: none">• defer pop• delayed branch• omit frame pointer• guess branch prob• cprop registers• forward propagate• if conversion• if conversion2• ipa pure const• ipa reference• merge constants• split wide types• tree ccp• tree dce• tree dom• tree dse• tree ter• tree sra• tree copyrename• tree fre• tree copy prop• tree sink• tree ch	<p>All Free optimizations plus:</p> <ul style="list-style-type: none">• indirect inlining• thread jumps• crossjumping• optimize sibling calls• cse follow jumps• gcse• expensive optimizations• cse after loop• caller saves• peephole2• schedule insns• schedule insns after reload• regmove• strict aliasing• strict overflow• reorder blocks• reorder functions• tree vrp• tree builtin call dce• tree pre• tree switch conversion• ipa cp• ipa sra	<p>All Free and STD optimizations plus:</p> <ul style="list-style-type: none">• predictive commoning• inline functions• unswitch loops• gcse after reload• tree vectorize• ipa cp clone• Whole-program optimizations

Chapter 19. Preprocessing

19.1 INTRODUCTION

All C source files are preprocessed before compilation. The `-E` option can be used to preprocess and then stop the compilation. See **Section 5.7.2 “Options for Controlling the Kind of Output”**.

Assembler files can also be preprocessed if the file extension is `.S` rather than `.s` (see **Section 5.2.3 “Input File Types”**.)

Items discussed in this section are:

- C Language Comments
- Preprocessing Directives
- Predefined Macro Names
- Pragmas vs. Attributes

19.2 C LANGUAGE COMMENTS

The MPLAB XC16 C Compiler supports standard C comments, as well as C++ style comments. Both types are illustrated in the following table.

Comment Syntax	Description	Example
<code>/* */</code>	Standard C code comment. Used for one or more lines.	<code>/* This is line 1 This is line 2 */</code>
<code>//</code>	C++ code comment. Used for one line only.	<code>// This is line 1 // This is line 2</code>

19.3 PREPROCESSING DIRECTIVES

The compiler accepts several specialized preprocessor directives in addition to the standard directives. All of these are listed in Table 19-1.

TABLE 19-1: PREPROCESSOR DIRECTIVES

Directive	Meaning	Example
<code>#define</code>	Define preprocessor macro	<code>#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))</code>
<code>#elif</code>	Short for <code>#else #if</code>	see <code>#ifdef</code>
<code>#else</code>	Conditionally include source lines	see <code>#if</code>
<code>#endif</code>	Terminate conditional source inclusion	see <code>#if</code>
<code>#error</code>	Generate an error message	<code>#error Size too big</code>
<code>#if</code>	Include source lines if constant expression true	<code>#if SIZE < 10 c = process(10) #else skip(); #endif</code>

TABLE 19-1: PREPROCESSOR DIRECTIVES (CONTINUED)

Directive	Meaning	Example
<code>#ifdef</code>	Include source lines if preprocessor symbol defined	<pre>#ifdef FLAG do_loop(); #eliff SIZE == 5 skip_loop(); #endif</pre>
<code>#ifndef</code>	Include source lines if preprocessor symbol not defined	<pre>#ifndef FLAG jump(); #endif</pre>
<code>#include</code>	Include text file into source	<pre>#include <stdio.h> #include "project.h"</pre>
<code>#line</code>	Specify line number and file name for listing	<code>#line 3 final</code>
<code>#pragma</code>	Compiler specific options	Refer to Section 19.5 “Pragmas vs. Attributes”
<code>#undef</code>	Undefines preprocessor symbol	<code>#undef FLAG</code>
<code>#warning</code>	Generate a warning message	<code>#warning Length not set</code>

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself, so for example

```
#define paste1(a,b) a##b
#define paste(a,b) paste1(a,b)
```

lets you use the `paste` macro to concatenate two expressions that themselves may require further expansion. Remember that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

For implementation-defined behavior of preprocessing directives, see **Section A.14 “Preprocessing Directives”**.

19.4 PREDEFINED MACRO NAMES

The compiler predefines several macros which can be tested by conditional directives in source code. Constants that have been deprecated may be found in **Appendix F. “Deprecated Features”**.

19.4.1 Compiler Version Macro

The compiler will define the constant `__XC16_VERSION__`, giving a numeric value to the version identifier. This can be used to take advantage of new compiler features while remaining backwardly compatible with older versions.

The value is based upon the major and minor version numbers of the current release. For example, release version 1.00 will have a `__XC16_VERSION__` definition of 1000. This macro can be used, in conjunction with standard preprocessor comparison statements, to conditionally include/exclude various code constructs.

The current definition of `__XC16_VERSION__` can be discovered by adding `--version` to the command line, or by inspecting the `README.html` file that came with the release.

19.4.2 Output Types and Device Macros

The following symbols are defined with the `-ansi` command line option.

TABLE 19-2: MACROS DEFINED WITH -ANSI

Symbol - Leading Double Underline	Symbol - Leading & Lagging Double Underline	Description
<code>__XC16</code>	<code>__XC16__</code>	If defined, 16-bit compiler is in use.
<code>__C30</code>	<code>__C30__</code>	
<code>__dsPICC30</code>	<code>__dsPIC30__</code>	
<code>__XC16ELF</code>	<code>__XC16ELF__</code>	If defined, compiler is producing ELF output.
<code>__C30ELF</code>	<code>__C30ELF__</code>	
<code>__dsPIC30ELF</code>	<code>__C30ELF__</code>	
<code>__XC16COFF</code>	<code>__XC16COFF__</code>	If defined, compiler is producing COFF output.
<code>__C30COFF</code>	<code>__C30COFF__</code>	
<code>__dsPIC30COFF</code>	<code>__dsPIC30COFF__</code>	

The following symbols are defined when `-ansi` is not selected.

TABLE 19-3: MACROS DEFINED WITHOUT -ANSI

Symbol	Description
<code>XC16</code>	16-bit compiler is in use.
<code>C30</code>	
<code>dsPIC30</code>	

In addition, the compiler defines a symbol based on the target device set with `-mcpu=`. For example, `-mcpu=30F6014`, which defines the symbol `__dsPIC30F6014__`.

19.4.3 Device Features Macros

The following symbols are defined if device features are enabled.

TABLE 19-4: DEVICE FEATURES MACROS/SYMBOLS

Symbol	Description
<code>__HAS_DSP__</code>	Device has a DSP engine.
<code>__HAS_EEDATA__</code>	Device has EEDATA memory.
<code>__HAS_DMA__</code>	Device has DMA memory.
<code>__HAS_DMA_DMAV2__</code>	Device has DMA V2 memory.
<code>__HAS_CODEGUARD__</code>	Device has CodeGuard™ Security.
<code>__HAS_PMP__</code>	Device has Parallel Master Port.
<code>__HAS_PMPV2__</code>	Device has Parallel Master Port V2.
<code>__HAS_PMP_ENHANCED__</code>	Device has Enhanced Parallel Master Port.
<code>__HAS_EDS__</code>	Device has Extended Data Space.
<code>__HAS_5VOLTS__</code>	Device is a 5-volt device.

19.4.4 Other Macros

The following symbols define other features.

TABLE 19-5: OTHER MACROS/SYMBOLS

Symbol	Description
__FILE__	Current file name as a C string.
__LINE__	Current line number as a C string.
__DATE__	Current date as a C string.

19.5 PRAGMAS VS. ATTRIBUTES

The MPLAB XC16 C Compiler uses non-ANSI attributes instead of pragmas or qualifiers to locate variables and functions in memory. As a comparison, the PIC18 MCU C Compiler - also called MPLAB C18 - uses pragmas for sections (`code`, `romdata`, `udata`, `idata`), interrupts (high-priority and low-priority) and variable locations (`bank`, `section`). The former HI-TECH C Compiler for PIC18 MCUs and the newer MPLAB XC8 compiler use qualifiers or pragmas to perform the same actions.

If you are used to using a PIC18 compiler, this section will show how to use XC16 attributes instead. For more on attributes, see **Section 8.12 “Variable Attributes”** and **Section 13.2.1 “Function Specifiers”**.

TABLE 19-6: C18 PRAGMAS VS. ATTRIBUTES

Pragma (MPLAB C18)	Attribute (MPLAB XC16)
<code>#pragma udata [name]</code>	<code>__attribute__((section ("name")))</code>
<code>#pragma idata [name]</code>	<code>__attribute__((section ("name")))</code>
<code>#pragma romdata [name]</code>	<code>__attribute__((space (auto_psv)))</code>
<code>#pragma code [name]</code>	<code>__attribute__((section ("name"), space (prog)))</code>
<code>#pragma interruptlow</code>	<code>__attribute__((interrupt))</code>
<code>#pragma interrupt</code>	<code>__attribute__((interrupt, shadow))</code>
<code>#pragma varlocate bank</code>	NA*
<code>#pragma varlocate name</code>	NA*

*16-bit devices do not have banks.

TABLE 19-7: PICC18 PRAGMAS AND QUALIFIERS VS. ATTRIBUTES

PICC18	Attribute (MPLAB XC16)
<code>#pragma psect old=new</code>	<code>__attribute__((section ("name")))</code>
<code>const</code>	<code>const</code> or <code>__attribute__((space (auto_psv)))</code>
<code>interrupt low_priority</code>	<code>__attribute__((interrupt))</code>
<code>interrupt</code>	<code>__attribute__((interrupt, shadow))</code>

EXAMPLE 19-1: SPECIFY AN UNINITIALIZED VARIABLE IN A USER SECTION IN DATA MEMORY

PICC18	<code>#pragma psect oldbss=mybss</code> <code>int gi;</code>
C18	<code>#pragma udata mybss</code> <code>int gi;</code>
XC16	<code>int __attribute__((__section__(".mybss"))) gi;</code>

where `oldbss` is the name of the psect (section) in which the variable would normally be placed.

EXAMPLE 19-2: LOCATE THE VARIABLE `MABONGA` AT ADDRESS `0x100` IN DATA MEMORY

PICC18	<code>int Mabonga @ 0x100;</code>
C18	<code>#pragma idata myDataSection=0x100;</code> <code>int Mabonga = 1;</code>
XC16	<code>int __attribute__((address(0x100))) Mabonga = 1;</code>

EXAMPLE 19-3: SPECIFY A VARIABLE TO BE PLACED IN PROGRAM MEMORY

PICC18	<code>const char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};</code>
C18	<code>#pragma romdata const_table const rom char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};</code>
XC16	<code>const or __attribute__((space(auto_psv))) char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};</code>

EXAMPLE 19-4: LOCATE THE FUNCTION PRINTSTRING AT ADDRESS 0x8000 IN PROGRAM MEMORY

PICC18	<code>int PrintString(const char *s)@ 0x8000 {...}</code>
C18	<code>#pragma code myTextSection=0x8000; int PrintString(const char *s){...}</code>
XC16	<code>int __attribute__((address(0x8000))) PrintString (const char *s) {...}</code>

EXAMPLE 19-5: COMPILER AUTOMATICALLY SAVES AND RESTORES THE VARIABLES var1 AND var2

PICC18	No equivalent
C18	<code>#pragma interrupt_isr0 save=var1, var2 void isr0(void) { /* perform interrupt function here */ }</code>
XC16	<code>void __attribute__((__interrupt__(__save__(var1,var2)))) isr0(void) { /* perform interrupt function here */ }</code>

Chapter 20. Linking Programs

20.1 INTRODUCTION

The compiler will automatically invoke the linker unless the compiler has been requested to stop after producing an intermediate file.

The linker will run with options that are obtained from the command-line driver. These options specify the memory of the device and how objects should be placed in the memory. Device-specific linker scripts are used.

The linker operation can be controlled using the driver, see **Section 5.7.9 “Options for Linking”** for more information.

The linker creates a map file which details the memory assigned and some objects within the code. The map file is the best place to look for memory information. See *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for an explanation of the detailed information in this file.

20.2 DEFAULT MEMORY SPACES

The compiler defines several special purpose memory spaces to match architectural features of 16-bit devices. Static and external variables may be allocated in the special purpose memory spaces through use of the `space` attribute, described in **Section 8.12 “Variable Attributes”**.

data

General data space. Variables in general data space can be accessed using ordinary C statements. This is the default allocation.

DD **xmemory** - dsPIC30F, dsPIC33EP/F devices only

X data address space. Variables in X data space can be accessed using ordinary C statements. X data address space has special relevance for DSP-oriented libraries and/or assembly language instructions.

DD **ymemory** - dsPIC30F, dsPIC33EP/F devices only

Y data address space. Variables in Y data space can be accessed using ordinary C statements. Y data address space has special relevance for DSP-oriented libraries and/or assembly language instructions.

prog

General program space, which is normally reserved for executable code. Variables in this program space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, using the program space visibility window, or by qualifying with `__prog__`.

auto_psv

A compiler-managed area in program space, designated for program space visibility window access. Variables in this space can be read (but not written) using ordinary C statements and are subject to a maximum of 32K total space allocated.

psv

Program space, designated for program space visibility window access. Variables in PSV space are not managed by the compiler and can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window. Variables in PSV space can be accessed using a single setting of the PSVPAG register or by qualifying with `__psv__`.

DD **eedata** - EEDATA capable devices only

Data EEPROM space, a region of 16-bit wide non-volatile memory located at high addresses in program memory. Variables in eedata space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window. The `__HAS_EEDATA__` manifest constant is defined for devices that support EEDATA.

DD **dma** - DMA capable devices only

DPSRAM DMA memory. Variables in DMA memory can be accessed using ordinary C statements and by the DMA peripheral. The `__HAS_DMA__` manifest constant is defined for devices that support DMA. If the device supports DMA but does not have special DPSRAM available, the linker will not be able to allocate the space and will output an error.

20.3 REPLACING LIBRARY SYMBOLS

The MPLAB XC16 C Compiler comes with a librarian which allows you to unpack a library file and replace modules with your own modified versions. See the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)*. However, you can easily replace a library module that is linked into your program without having to do this.

If you add to your project a source file which contains the definition for a routine with the same name as a library routine, then the library routine will be replaced by your routine.

When trying to resolve a symbol (a function name, or variable name, for example) the compiler first scans all the source modules for the definition. Only if it cannot resolve the symbol in these files does it then search the library files.

If the symbol is defined in a source file, the compiler will never actually search the libraries for this symbol and no error will result even if the symbol was present in the library files. This may not be true if a symbol is defined twice in source files and an error may result if there is a conflict in the definitions.

Another method is to use the `weak` attribute when declaring a symbol. A weak symbol may be superseded by a global definition. When `weak` is applied to a reference to an external symbol, the symbol is not required for linking.

The `weak` attribute may be applied to functions as well as variables. Code may be written such that the function will be used only if it is linked in from some other module. Deciding whether or not to use the feature becomes a link-time decision, not a compile time decision.

For more information on the `weak` attribute, see **Section 8.12 “Variable Attributes”**.

20.4 LINKER-DEFINED SYMBOLS

The 16-bit linker defines several symbols that may be used in your C code development. Please see the *MPLAB XC16 Assembler, Linker and Utilities User's Guide (DS52106)* for more information.

A useful address symbol, `_PROGRAM_END`, is defined in program memory to mark the highest address used by a `CODE` or `PSV` section. It should be referenced with the address operator (`&`) in a built-in function call that accepts the address of an object in program memory. This symbol can be used by applications as an end point for checksum calculations.

For example:

```
unsigned int end_page, end_offset;
_prog_addressT big_addr;

end_page    = __builtin_tblpage(&_PROGRAM_END);
end_offset  = __builtin_tbloffset(&_PROGRAM_END);

_init_prog_address(big_addr, _PROGRAM_END);
```

20.5 DEFAULT LINKER SCRIPT

The command line always requires a linker script. However, if no linker script is specified in an MPLAB IDE project, the IDE will use the device linker script file (*device.gld*) included with the compiler as the default linker script. This device-specific file contains information such as:

- Memory region definitions
- Program, data and debug sections mapping
- Interrupt and alternate interrupt vector table maps
- SFR equates
- Base addresses for various peripherals

Linker scripts may be found, by default, in:

`<install-dir>\support\DeviceFamily\gld`

where *DeviceFamily* is the 16-bit device family, such as dsPIC30F.

To use a custom linker script in your project, simply add that file to the command line or the project in the “Linker Script” (MPLAB IDE v8) or “Linker Files” (MPLAB X IDE) folder.

Appendix A. Implementation-Defined Behavior

A.1 INTRODUCTION

This section offers implementation-defined behavior of the MPLAB XC16 C Compiler. The ISO standard for C requires that vendors document the specifics of “implementation defined” features of the language.

Items discussed are:

- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating Point
- Arrays and Pointers
- Registers
- Structures, Unions, Enumerations and Bit-Fields
- Qualifiers
- Declarators
- Statements
- Preprocessing Directives
- Library Functions
- Signals
- Streams and Files
- tmpfile
- errno
- Memory
- abort
- exit
- getenv
- system
- strerror

A.2 TRANSLATION

Implementation-Defined Behavior for Translation is covered in section G.3.1 of the ANSI C Standard.

Is each non-empty sequence of white-space characters, other than new line, retained or is it replaced by one space character? (ISO 5.1.1.2)

It is replaced by one space character.

How is a diagnostic message identified? (ISO 5.1.1.3)

Diagnostic messages are identified by prefixing them with the source file name and line number corresponding to the message, separated by colon characters (':').

Are there different classes of message? (ISO 5.1.1.3)

Yes.

If yes, what are they? (ISO 5.1.1.3)

Errors, which inhibit production of an output file, and warnings, which do not inhibit production of an output file.

What is the translator return status code for each class of message? (ISO 5.1.1.3)

The return status code for errors is 1; for warnings it is 0.

Can a level of diagnostic be controlled? (ISO 5.1.1.3)

Yes.

If yes, what form does the control take? (ISO 5.1.1.3)

Compiler command line options may be used to request or inhibit the generation of warning messages.

A.3 ENVIRONMENT

Implementation-Defined Behavior for Environment is covered in section G.3.2 of the ANSI C Standard.

What library facilities are available to a freestanding program? (ISO 5.1.2.1)

All of the facilities of the standard C library are available, provided that a small set of functions is customized for the environment, as described in the "Run Time Libraries" section.

Describe program termination in a freestanding environment. (ISO 5.1.2.1)

If the function `main` returns or the function `exit` is called, a `HALT` instruction is executed in an infinite loop. This behavior is customizable.

Describe the arguments (parameters) passed to the function `main`? (ISO 5.1.2.2.1)

No parameters are passed to `main`.

Which of the following is a valid interactive device: (ISO 5.1.2.3)

Asynchronous terminal No

Paired display and keyboard No

Inter program connection No

Other, please describe? None

A.4 IDENTIFIERS

Implementation-Defined Behavior for Identifiers is covered in section G.3.3 of the ANSI C Standard.

How many characters beyond thirty-one (31) are significant in an identifier without external linkage? (ISO 6.1.2)

All characters are significant.

How many characters beyond six (6) are significant in an identifier with external linkage? (ISO 6.1.2)

All characters are significant.

Is case significant in an identifier with external linkage? (ISO 6.1.2)

Yes.

A.5 CHARACTERS

Implementation-Defined Behavior for Characters is covered in section G.3.4 of the ANSI C Standard.

Detail any source and execution characters which are not explicitly specified by the Standard? (ISO 5.2.1)

None.

List escape sequence value produced for listed sequences. (ISO 5.2.2)

TABLE A-1: ESCAPE SEQUENCE CHARACTERS AND VALUES

Sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

How many bits are in a character in the execution character set? (ISO 5.2.4.2)

8.

What is the mapping of members of the source character sets (in character and string literals) to members of the execution character set? (ISO 6.1.3.4)

The identity function.

What is the equivalent type of a plain `char`? (ISO 6.2.1.1)

Either (user defined). The default is `signed char`. A compiler command-line option may be used to make the default `unsigned char`.

A.6 INTEGERS

Implementation-Defined Behavior for Integers is covered in section G.3.5 of the ANSI C Standard.

The following table describes the amount of storage and range of various types of integers: (ISO 6.1.2.5)

TABLE A-2: INTEGER TYPES

Designation	Size (bits)	Range
char	8	-128 ... 127
signed char	8	-128 ... 127
unsigned char	8	0 ... 255
short	16	-32768 ... 32767
signed short	16	-32768 ... 32767
unsigned short	16	0 ... 65535
int	16	-32768 ... 32767
signed int	16	-32768 ... 32767
unsigned int	16	0 ... 65535
long	32	-2147483648 ... 2147438647
signed long	32	-2147483648 ... 2147438647
unsigned long	32	0 ... 4294867295

What is 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? (ISO 6.2.1.2)

There is a loss of significance. No error is signaled.

What are the results of bitwise operations on signed integers? (ISO 6.3)

Shift operators retain the sign. Other operators act as if the operand(s) are unsigned integers.

What is the sign of the remainder on integer division? (ISO 6.3.5)

+

What is the result of a right shift of a negative-valued signed integral type? (ISO 6.3.7)

The sign is retained.

A.7 FLOATING POINT

Implementation-Defined Behavior for Floating Point is covered in section G.3.6 of the ANSI C Standard.

Is the scaled value of a floating constant that is in the range of the representable value for its type, the nearest representable value, or the larger representable value immediately adjacent to the nearest representable value, or the smallest representable value immediately adjacent to the nearest representable value? (ISO 6.1.3.1)

The nearest representable value.

The following table describes the amount of storage and range of various types of floating point numbers: (ISO 6.1.2.5)

TABLE A-3: FLOATING-POINT TYPES

Designation	Size (bits)	Range
float	32	1.175494e-38 ... 3.40282346e+38
double*	32	1.175494e-38 ... 3.40282346e+38
long double	64	2.22507385e-308 ... 1.79769313e+308

* double is equivalent to long double if -fno-short-double is used.

What is the direction of truncation, when an integral number is converted to a floating-point number, that cannot exactly represent the original value? (ISO 6.2.1.3)

Down.

What is the direction of truncation, or rounding, when a floating-point number is converted to a narrower floating-point number? (ISO 6.2.1.4)

Down.

A.8 ARRAYS AND POINTERS

Implementation-Defined Behavior for Arrays and Pointers is covered in section G.3.7 of the ANSI C Standard.

What is the type of the integer required to hold the maximum size of an array that is, the type of the size of operator, `size_t`? (ISO 6.3.3.4, ISO 7.1.1)

unsigned int.

What is the size of integer required for a pointer to be converted to an integral type? (ISO 6.3.4)

16 bits.

What is the result of casting a pointer to an integer, or vice versa? (ISO 6.3.4)

The mapping is the identity function.

What is the type of the integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t`? (ISO 6.3.6, ISO 7.1.1)

unsigned int.

A.9 REGISTERS

Implementation-Defined Behavior for Registers is covered in section G.3.8 of the ANSI C Standard.

To what extent does the storage class specifier `register` actually effect the storage of objects in registers? (ISO 6.5.1)

If optimization is disabled, an attempt will be made to honor the `register` storage class; otherwise, it is ignored.

A.10 STRUCTURES, UNIONS, ENUMERATIONS AND BIT-FIELDS

Implementation-Defined Behavior for Structures, Unions, Enumerations and Bit-Fields is covered in sections A.6.3.9 and G.3.9 of the ANSI C Standard.

What are the results if a member of a union object is accessed using a member of a different type? (ISO 6.3.2.3)

No conversions are applied.

Describe the padding and alignment of members of structures? (ISO 6.5.2.1)

Chars are byte-aligned. All other objects are word-aligned.

What is the equivalent type for a plain `int` bit-field? (ISO 6.5.2.1)

User defined. By default, `signed int` bit-field. May be made an `unsigned int` `bitfield` using a command line option.

What is the order of allocation of bit-fields within an `int`? (ISO 6.5.2.1)

Bits are allocated from least-significant to most-significant.

Can a bit-field straddle a storage-unit boundary? (ISO 6.5.2.1)

Yes.

Which integer type has been chosen to represent the values of an enumeration type? (ISO 6.5.2.2)

`int`.

A.11 QUALIFIERS

Implementation-Defined Behavior for Qualifiers is covered in section G.3.10 of the ANSI C Standard.

Describe what action constitutes an access to an object that has volatile-qualified type? (ISO 6.5.3)

If an object is named in an expression, it has been accessed.

A.12 DECLARATORS

Implementation-Defined Behavior for Declarators is covered in section G.3.11 of the ANSI C Standard.

What is the maximum number of declarators that may modify an arithmetic, structure, or union type? (ISO 6.5.4)

No limit.

A.13 STATEMENTS

Implementation-Defined Behavior for Statements is covered in section G.3.12 of the ANSI C Standard.

What is the maximum number of case values in a switch statement? (ISO 6.6.4.2)

No limit.

A.14 PREPROCESSING DIRECTIVES

Implementation-Defined Behavior for Preprocessing Directives is covered in section G.3.13 of the ANSI C Standard.

Does the value of a single-character character constant in a constant expression, that controls conditional inclusion, match the value of the same character constant in the execution character set? (ISO 6.8.1)

Yes.

Can such a character constant have a negative value? (ISO 6.8.1)

Yes.

What method is used for locating includable source files? (ISO 6.8.2)

The preprocessor searches the current directory, followed by directories named using command-line options.

How are headers identified, or the places specified? (ISO 6.8.2)

The headers are identified on the `#include` directive, enclosed between `<` and `>` delimiters, or between `"` and `"` delimiters. The places are specified using command-line options.

Are quoted names supported for includable source files? (ISO 6.8.2)

Yes.

What is the mapping between delimited character sequences and external source file names? (ISO 6.8.2)

The identity function.

Describe the behavior of each recognized #pragma directive. (ISO 6.8.6)

TABLE A-4: #PRAGMA BEHAVIOR

Pragma	Behavior
<code>#pragma code section-name</code>	Names the code section.
<code>#pragma code</code>	Resets the name of the code section to its default (viz., <code>.text</code>).
<code>#pragma config</code>	Sets configuration bits or registers.
<code>#pragma idata section-name</code>	Names the initialized data section.
<code>#pragma idata</code>	Resets the name of the initialized data section to its default value (viz., <code>.data</code>).
<code>#pragma udata section-name</code>	Names the uninitialized data section.
<code>#pragma udata</code>	Resets the name of the uninitialized data section to its default value (viz., <code>.bss</code>).
<code>#pragma interrupt function-name</code>	Designates function-name as an interrupt function.

What are the definitions for `__DATE__` and `__TIME__` respectively, when the date and time of translation are not available? (ISO 6.8.8)

Not applicable. The compiler is not supported in environments where these functions are not available.

A.15 LIBRARY FUNCTIONS

Implementation-Defined Behavior for Library Functions is covered in section G.3.14 of the ANSI C Standard.

What is the null pointer constant to which the macro NULL expands? (ISO 7.1.5)

0.

How is the diagnostic printed by the assert function recognized, and what is the termination behavior of this function? (ISO 7.2)

The assert function prints the file name, line number and test expression, separated by the colon character (':'). It then calls the abort function.

What characters are tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions? (ISO 7.3.1)

TABLE A-5: CHARACTERS TESTED BY `is` FUNCTIONS

Function	Characters tested
isalnum	One of the letters or digits: <code>isalpha</code> or <code>isdigit</code> .
isalpha	One of the letters: <code>islower</code> or <code>isupper</code> .
iscntrl	One of the five standard motion control characters, backspace and alert: <code>\f</code> , <code>\n</code> , <code>\r</code> , <code>\t</code> , <code>\v</code> , <code>\b</code> , <code>\a</code> .
islower	One of the letters 'a' through 'z'.
isprint	A graphic character or the space character: <code>isalnum</code> or <code>ispunct</code> or space.
isupper	One of the letters 'A' through 'Z'.
ispunct	One of the characters: <code>!"#\$%&'()*;<=>?[\]^_`{ }~.,/:^</code>

What values are returned by the mathematics functions after a domain errors? (ISO 7.5.1)

NaN.

Do the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow range errors? (ISO 7.5.1)

Yes.

Do you get a domain error or is zero returned when the `fmod` function has a second argument of zero? (ISO 7.5.6.4)

Domain error.

A.16 SIGNALS

What is the set of signals for the signal function? (ISO 7.7.1.1)

TABLE A-6: SIGNAL FUNCTION

Name	Description
SIGABRT	Abnormal termination.
SIGINT	Receipt of an interactive attention signal.
SIGILL	Detection of an invalid function image.
SIGFPE	An erroneous arithmetic operation.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

Describe the parameters and the usage of each signal recognized by the signal function. (ISO 7.7.1.1)

Application defined.

Describe the default handling and the handling at program startup for each signal recognized by the signal function? (ISO 7.7.1.1)

None.

If the equivalent of signal (sig, SIG_DFL) is not executed prior to the call of a signal handler, what blocking of the signal is performed? (ISO 7.7.1.1)

None.

Is the default handling reset if a SIGILL signal is received by a handler specified to the signal function? (ISO 7.7.1.1)

No.

A.17 STREAMS AND FILES

Does the last line of a text stream require a terminating new line character? (ISO 7.9.2)

No.

Do space characters, that are written out to a text stream immediately before a new line character, appear when the stream is read back in? (ISO 7.9.2)

Yes.

How many null characters may be appended to data written to a binary stream? (ISO 7.9.2)

None.

Is the file position indicator of an append mode stream initially positioned at the start or end of the file? (ISO 7.9.3)

Start.

Does a write on a text stream cause the associated file to be truncated beyond that point? (ISO 7.9.3)

Application defined.

Describe the characteristics of file buffering. (ISO 7.9.3)

Fully buffered.

Can zero-length file actually exist? (ISO 7.9.3)

Yes.

What are the rules for composing a valid file name? (ISO 7.9.3)

Application defined.

Can the same file be open multiple times? (ISO 7.9.3)

Application defined.

What is the effect of the remove function on an open file? (ISO 7.9.4.1)

Application defined.

What is the effect if a file with the new name exists prior to a call to the rename function? (ISO 7.9.4.2)

Application defined.

What is the form of the output for %p conversion in the fprintf function? (ISO 7.9.6.1)

A hexadecimal representation.

What form does the input for %p conversion in the fscanf function take? (ISO 7.9.6.2)

A hexadecimal representation.

A.18 TMPFILE

Is an open temporary file removed if the program terminates abnormally? (ISO 7.9.4.3)
Yes.

A.19 ERRNO

What value is the macro `errno` set to by the `fgetpos` or `ftell` function on failure? (ISO 7.9.9.1, (ISO 7.9.9.4)

Application defined.

What is the format of the messages generated by the `perror` function? (ISO 7.9.10.4)

The argument to `perror`, followed by a colon, followed by a text description of the value of `errno`.

A.20 MEMORY

What is the behavior of the `calloc`, `malloc` or `realloc` function if the size requested is zero? (ISO 7.10.3)

A block of zero length is allocated.

A.21 ABORT

What happens to open and temporary files when the `abort` function is called? (ISO 7.10.4.1)

Nothing.

A.22 EXIT

What is the status returned by the `exit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`? (ISO 7.10.4.3)

The value of the argument.

A.23 GETENV

What limitations are there on environment names? (ISO 7.10.4.4)

Application defined.

Describe the method used to alter the environment list obtained by a call to the `getenv` function. (ISO 7.10.4.4)

Application defined.

A.24 SYSTEM

Describe the format of the string that is passed to the `system` function. (ISO 7.10.4.5)

Application defined.

What mode of execution is performed by the `system` function? (ISO 7.10.4.5)

Application defined.

A.25 STRERROR

Describe the format of the error message output by the `strerror` function. (ISO 7.11.6.2)

A plain character string.

List the contents of the error message strings returned by a call to the `strerror` function. (ISO 7.11.6.2)

TABLE A-7: ERROR MESSAGE STRINGS

Errno	Message
0	No error
EDOM	Domain error
ERANGE	Range error
EFPOS	File positioning error
EFOPEN	File open error
nnn	Error #nnn

Appendix B. Embedded Compiler Compatibility Mode

B.1 INTRODUCTION

All three MPLAB XC C compilers can be placed into a compatibility mode. In this mode, they are syntactically compatible with the non-standard C language extensions used by other non-Microchip embedded compiler vendors. This compatibility allows C source code written for other compilers to be compiled with minimum modification when using the MPLAB XC compilers.

Since very different device architectures may be targeted by other compilers, the semantics of the non-standard extensions may be different to that in the MPLAB XC compilers. This document indicates when the original C code may need to be reviewed.

The compatibility features offered by the MPLAB C compilers are discussed in the following topics:

- Compiling in Compatibility Mode
- Syntax Compatibility
- Data Type
- Operator
- Extended Keywords
- All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.
- Pragmas

B.2 COMPILING IN COMPATIBILITY MODE

An option is used to enable vendor-specific syntax compatibility. When using MPLAB XC8, this option is `--ext=vendor`; when using MPLAB XC16 or MPLAB XC32, the option is `-mext=vendor`. The argument *vendor* is a key that is used to represent the syntax. See Table B-1 for a list of all keys usable with the MPLAB XC compilers.

TABLE B-1: VENDOR KEYS

Vendor key	Syntax	XC8 Support	XC16 Support	XC32 Support
cci	Common Compiler Interface	Yes	Yes	Yes
iar	IAR C/C++ Compiler™ for ARM	Yes	Yes	Yes

The Common Compiler Interface is a language standard that is common to all Microchip MPLAB XC compilers. The non-standard extensions associated with this syntax are already described in **Chapter 2. “Common C Interface”** and are not repeated here.

B.3 SYNTAX COMPATIBILITY

The goal of this syntax compatibility feature is to ease the migration process when porting source code from other C compilers to the native MPLAB XC compiler syntax.

Many non-standard extensions are not required when compiling for Microchip devices and, for these, there are no equivalent extensions offered by MPLAB XC compilers. These extensions are then simply ignored by the MPLAB XC compilers, although a warning message is usually produced to ensure you are aware of the different compiler behavior. You should confirm that your project will still operate correctly with these features disabled.

Other non-standard extensions are not compatible with Microchip devices. Errors will be generated by the MPLAB XC compiler if these extensions are not removed from the source code. You should review the ramifications of removing the extension and decide whether changes are required to other source code in your project.

Table B-2 indicates the various levels of compatibility used in the tables that are presented throughout this guide.

TABLE B-2: LEVEL OF SUPPORT INDICATORS

Level	Explanation
support	The syntax is accepted in the specified compatibility mode, and its meaning will mimic its meaning when it is used with the original compiler.
support (no args)	In the case of pragmas, the base pragma is supported in the specified compatibility mode, but the arguments are ignored.
native support	The syntax is equivalent to that which is already accepted by the MPLAB XC compiler, and the semantics are compatible. You can use this feature without a vendor compatibility mode having been enabled.
ignore	The syntax is accepted in the specified compatibility mode, but the implied action is not required or performed. The extension is ignored and a warning will be issued by the compiler.
error	The syntax is not accepted in the specified compatibility mode. An error will be issued and compilation will be terminated.

Note that even if a C feature is supported by an MPLAB XC compiler, addresses, register names, assembly instructions, or any other device-specific argument is unlikely to be valid when compiling for a Microchip device. Always review code which uses these items in conjunction with the data sheet of your target Microchip device.

Embedded Compiler Compatibility Mode

B.4 DATA TYPE

Some compilers allow use of the boolean type, `bool`, as well as associated values `true` and `false`, as specified by the C99 ANSI Standard. This type and these values may be used by all MPLAB XC compilers when in compatibility mode¹, as shown in Table B-3.

As indicated by the ANSI Standard, the `<stdbool.h>` header must be included for this feature to work as expected when it is used with MPLAB XC compilers.

TABLE B-3: SUPPORT FOR C99 BOOL TYPE

IAR Compatibility Mode			
Type	XC8	XC16	XC32
<code>bool</code>	support	support	support

Do not confuse the boolean type, `bool`, and the integer type, `bit`, implemented by MPLAB XC8.

B.5 OPERATOR

The `@` operator may be used with other compilers to indicate the desired memory location of an object. As Table B-4 indicates, support for this syntax in MPLAB C is limited to MPLAB XC8 only.

Any address specified with another device is unlikely to be correct on a new architecture. Review the address in conjunction with the data sheet for your target Microchip device.

Using `@` in a compatibility mode with MPLAB XC8 will work correctly, but will generate a warning. To prevent this warning from appearing again, use the reviewed address with the MPLAB C `__at()` specifier instead.

For MPLAB XC16/32, consider using the `address` attribute.

TABLE B-4: SUPPORT FOR NON-STANDARD OPERATOR

IAR Compatibility Mode			
Operator	XC8	XC16	XC32
<code>@</code>	native support	error	error

1. Not all C99 features have been adopted by all Microchip MPLAB XC compilers.

B.6 EXTENDED KEYWORDS

Non-standard extensions often specify how objects are defined or accessed. Keywords are usually used to indicate the feature. The non-standard C keywords corresponding to other compilers are listed in Table B-5, as well as the level of compatibility offered by MPLAB XC compilers. The table notes offer more information about some extensions.

TABLE B-5: SUPPORT FOR NON-STANDARD KEYWORDS

IAR Compatibility Mode			
Keyword	XC8	XC16	XC32
<code>__section_begin</code>	ignore	support	support
<code>__section_end</code>	ignore	support	support
<code>__section_size</code>	ignore	support	support
<code>__segment_begin</code>	ignore	support	support
<code>__segment_end</code>	ignore	support	support
<code>__segment_size</code>	ignore	support	support
<code>__sfb</code>	ignore	support	support
<code>__sfe</code>	ignore	support	support
<code>__sfs</code>	ignore	support	support
<code>__asm</code> or <code>asm</code> ⁽¹⁾	support ⁽²⁾	native support	native support
<code>__arm</code>	ignore	ignore	ignore
<code>__big_endian</code>	error	error	error
<code>__fiq</code>	support	error	error
<code>__intrinsic</code>	ignore	ignore	ignore
<code>__interwork</code>	ignore	ignore	ignore
<code>__irq</code>	support	error	error
<code>__little_endian</code> ⁽³⁾	ignore	ignore	ignore
<code>__nested</code>	ignore	ignore	ignore
<code>__no_init</code>	support	support	support
<code>__noreturn</code>	ignore	support	support
<code>__ramfunc</code>	ignore	ignore	support ⁽⁴⁾
<code>__packed</code>	ignore ⁽⁵⁾	support	support
<code>__root</code>	ignore	support	support
<code>__swi</code>	ignore	ignore	ignore
<code>__task</code>	ignore	support	support
<code>__weak</code>	ignore	support	support
<code>__thumb</code>	ignore	ignore	ignore
<code>__farfunc</code>	ignore	ignore	ignore
<code>__huge</code>	ignore	ignore	ignore
<code>__nearfunc</code>	ignore	ignore	ignore
<code>__inline</code>	support	native support	native support

Note 1: All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.

2: The keyword, `asm`, is supported natively by MPLAB XC8, but this compiler only supports the `__asm` keyword in IAR compatibility mode.

3: This is the default (and only) endianness used by all MPLAB XC compilers.

4: When used with MPLAB XC32, this must be used with the `__longcall__` macro for full compatibility.

5: Although this keyword is ignored, by default, all structures are packed when using

Embedded Compiler Compatibility Mode

MPLAB XC8, so there is no loss of functionality.

B.7 INTRINSIC FUNCTIONS

Intrinsic functions can be used to perform common tasks in the source code. The MPLAB XC compilers' support for the intrinsic functions offered by other compilers is shown in Table B-6.

TABLE B-6: SUPPORT FOR NON-STANDARD INTRINSIC FUNCTIONS

IAR Compatibility Mode			
Function	XC8	XC16	XC32
<code>__disable_fiq¹</code>	support	ignore	ignore
<code>__disable_interrupt</code>	support	support	support
<code>__disable_irq¹</code>	support	ignore	ignore
<code>__enable_fiq¹</code>	support	ignore	ignore
<code>__enable_interrupt</code>	support	support	support
<code>__enable_irq¹</code>	support	ignore	ignore
<code>__get_interrupt_state</code>	ignore	support	support
<code>__set_interrupt_state</code>	ignore	support	support

Note 1: These intrinsic functions map to macros which disable or enable the global interrupt enable bit on 8-bit PIC® devices.

The header file `<xc.h>` must be included for supported functions to operate correctly.

B.8 PRAGMAS

Pragmas may be used by a compiler to control code generation. Any compiler will ignore an unknown pragma, but many pragmas implemented by another compiler have also been implemented by the MPLAB XC compilers in compatibility mode. Table B-7 shows the pragmas and the level of support when using each of the MPLAB XC compilers.

Many of these pragmas take arguments. Even if a pragma is supported by an MPLAB XC compiler, this support may not apply to all of the pragma's arguments. This is indicated in the following table.

TABLE B-7: SUPPORT FOR NON-STANDARD PRAGMAS

IAR Compatibility Mode			
Pragma	XC8	XC16	XC32
<code>bitfields</code>	ignore	ignore	ignore
<code>data_alignment</code>	ignore	support	support
<code>diag_default</code>	ignore	ignore	ignore
<code>diag_error</code>	ignore	ignore	ignore
<code>diag_remark</code>	ignore	ignore	ignore
<code>diag_suppress</code>	ignore	ignore	ignore
<code>diag_warning</code>	ignore	ignore	ignore
<code>include_alias</code>	ignore	ignore	ignore
<code>inline</code>	support (no args)	support (no args)	support (no args)
<code>language</code>	ignore	ignore	ignore
<code>location</code>	ignore	support	support
<code>message</code>	support	native support	native support

TABLE B-7: SUPPORT FOR NON-STANDARD PRAGMAS (CONTINUED)

IAR Compatibility Mode			
Pragma	XC8	XC16	XC32
object_attribute	ignore	ignore	ignore
optimize	ignore	native support	native support
pack	ignore	native support	native support
__printf_args	support	support	support
required	ignore	support	support
rtmodel	ignore	ignore	ignore
__scanf_args	ignore	support	support
section	ignore	support	support
segment	ignore	support	support
swi_number	ignore	ignore	ignore
type_attribute	ignore	ignore	ignore
weak	ignore	native support	native support

Appendix C. Diagnostics

C.1 INTRODUCTION

This appendix lists the most common diagnostic messages generated by the MPLAB XC16 C Compiler.

The compiler can produce two kinds of diagnostic messages: Errors and Warnings. Each kind has a different purpose.

- Error messages report problems that make it impossible to compile your program. The compiler reports errors with the source file name, and the line number where the problem is apparent.
- Warning messages report other unusual conditions in your code that may indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text `warning:` to distinguish them from error messages.

Warnings may indicate danger points that should be checked to ensure that your program performs as directed. A warning may signal that obsolete features or non-standard features of the compiler are being used. Many warnings are issued only if you ask for them with one of the `-W` options (for instance, `-Wall` requests a variety of useful warnings).

In rare instances, the compiler may issue an internal error message report. This signifies that the compiler itself has detected a fault that should be reported to Microchip Support. Details on contacting support are located in the "Preface".

C.2 ERRORS

Symbols

`\x` used with no following HEX digits

The escape sequence `\x` should be followed by hex digits.

`'&'` constraint used with no register class

The asm statement is invalid.

`'%'` constraint used with last operand

The asm statement is invalid.

`#elif` after `#else`

In a preprocessor conditional, the `#else` clause must appear after any `#elif` clauses.

`#elif` without `#if`

In a preprocessor conditional, the `#if` must be used before using the `#elif`.

`#else` after `#else`

In a preprocessor conditional, the `#else` clause must appear only once.

`#else` without `#if`

In a preprocessor conditional, the `#if` must be used before using the `#else`.

#endif without #if

In a preprocessor conditional, the #if must be used before using the #endif.

#error 'message'

This error appears in response to a #error directive.

#if with no expression

An expression that evaluates to a constant arithmetic value was expected.

#include expects "FILENAME" or <FILENAME>

The file name for the #include is missing or incomplete. It must be enclosed by quotes or angle brackets.

is not followed by a macro parameter

The stringsize operator, '#' must be followed by a macro argument name.

#keyword expects "FILENAME" or <FILENAME>

The specified #keyword expects a quoted or bracketed file name as an argument.

is not followed by a macro parameter

The '#' operator should be followed by a macro argument name.

cannot appear at either end of a macro expansion

The concatenation operator, '##' may not appear at the start or the end of a macro expansion.

A

a parameter list with an ellipsis can't match an empty parameter name list declaration

The declaration and definition of a function must be consistent.

"symbol" after #line is not a positive integer

#line is expecting a source line number which must be positive.

aggregate value used where a complex was expected

Do not use aggregate values where complex values are expected.

aggregate value used where a float was expected

Do not use aggregate values where floating-point values are expected.

aggregate value used where an integer was expected

Do not use aggregate values where integer values are expected.

alias arg not a string

The argument to the alias attribute must be a string that names the target for which the current identifier is an alias.

alignment may not be specified for 'identifier'

The aligned attribute may only be used with a variable.

'__alignof' applied to a bit-field

The '__alignof' operator may not be applied to a bit-field.

alternate interrupt vector is not a constant

The interrupt vector number must be an integer constant.

alternate interrupt vector number *n* is not valid

A valid interrupt vector number is required.

ambiguous abbreviation argument

The specified command-line abbreviation is ambiguous.

an argument type that has a default promotion can't match an empty parameter name list declaration.

The declaration and definition of a function must be consistent.

args to be formatted is not ...

The first-to-check index argument of the format attribute specifies a parameter that is not declared '...'.

argument '*identifier*' doesn't match prototype

Function argument types should match the function's prototype.

argument of 'asm' is not a constant string

The argument of 'asm' must be a constant string.

argument to '-B' is missing

The directory name is missing.

argument to '-l' is missing

The library name is missing.

argument to '-specs' is missing

The name of the specs file is missing.

argument to '-specs=' is missing

The name of the specs file is missing.

argument to '-x' is missing

The language name is missing.

argument to '-Xlinker' is missing

The argument to be passed to the linker is missing.

arithmetic on pointer to an incomplete type

Arithmetic on a pointer to an incomplete type is not allowed.

array index in non-array initializer

Do not use array indices in non-array initializers.

array size missing in '*identifier*'

An array size is missing.

array subscript is not an integer

Array subscripts must be integers.

'asm' operand constraint incompatible with operand size

The asm statement is invalid.

'asm' operand requires impossible reload

The asm statement is invalid.

asm template is not a string constant

Asm templates must be string constants.

assertion without predicate

#assert or #unassert must be followed by a predicate, which must be a single identifier.

'attribute' attribute applies only to functions

The attribute '*attribute*' may only be applied to functions.

B

bit-field '*identifier*' has invalid type

Bit-fields must be of enumerated or integral type.

bit-field '*identifier*' width not an integer constant

Bit-field widths must be integer constants.

both long and short specified for '*identifier*'

A variable cannot be of type long and of type short.

both signed and unsigned specified for '*identifier*'

A variable cannot be both signed and unsigned.

braced-group within expression allowed only inside a function

It is illegal to have a braced-group within expression outside a function.

break statement not within loop or switch

Break statements must only be used within a loop or switch.

__builtin_longjmp second argument must be 1

__builtin_longjmp requires its second argument to be 1.

C

called object is not a function

Only functions may be called in C.

cannot convert to a pointer type

The expression cannot be converted to a pointer type.

cannot put object with volatile field into register

It is not legal to put an object with a volatile field into a register.

cannot reload integer constant operand in '*asm*'

The asm statement is invalid.

cannot specify both near and far attributes

The attributes near and far are mutually exclusive, only one may be used for a function or variable.

cannot take address of bit-field '*identifier*'

It is not legal to attempt to take address of a bit-field.

can't open '*file*' for writing

The system cannot open the specified '*file*'. Possible causes are not enough disk space to open the file, the directory does not exist, or there is no write permission in the destination directory.

can't set '*attribute*' attribute after definition

The '*attribute*' attribute must be used when the symbol is defined.

case label does not reduce to an integer constant

Case labels must be compile-time integer constants.

case label not within a switch statement

Case labels must be within a switch statement.

cast specifies array type

It is not permissible for a cast to specify an array type.

cast specifies function type

It is not permissible for a cast to specify a function type.

cast to union type from type not present in union

When casting to a union type, do so from type present in the union.

char-array initialized from wide string

Char-arrays should not be initialized from wide strings. Use ordinary strings.

file: *compiler* compiler not installed on this system

Only the C compiler is distributed; other high-level languages are not supported.

complex invalid for '*identifier*'

The complex qualifier may only be applied to integral and floating types.

conflicting types for '*identifier*'

Multiple, inconsistent declarations exist for identifier.

continue statement not within loop

Continue statements must only be used within a loop.

conversion to non-scalar type requested

Type conversion must be to a scalar (not aggregate) type.

D**data type of '*name*' isn't suitable for a register**

The data type does not fit into the requested register.

declaration for parameter '*identifier*' but no such parameter

Only parameters in the parameter list may be declared.

declaration of '*identifier*' as array of functions

It is not legal to have an array of functions.

declaration of '*identifier*' as array of voids

It is not legal to have an array of voids.

'*identifier*' declared as function returning a function

Functions may not return functions.

'*identifier*' declared as function returning an array

Functions may not return arrays.

decrement of pointer to unknown structure

Do not decrement a pointer to an unknown structure.

'default' label not within a switch statement

Default case labels must be within a switch statement.

'*symbol*' defined both normally and as an alias

A '*symbol*' can not be used as an alias for another symbol if it has already been defined.

'defined' cannot be used as a macro name

The macro name cannot be called 'defined'.

dereferencing pointer to incomplete type

A dereferenced pointer must be a pointer to an incomplete type.

division by zero in #if

Division by zero is not computable.

duplicate case value

Case values must be unique.

duplicate label '*identifier*'

Labels must be unique within their scope.

duplicate macro parameter '*symbol*'

'symbol' has been used more than once in the parameter list.

duplicate member '*identifier*'

Structures may not have duplicate members.

duplicate (or overlapping) case value

Case ranges must not have a duplicate or overlapping value. The error message 'this is the first entry overlapping that value' will provide the location of the first occurrence of the duplicate or overlapping value. Case ranges are an extension of the ANSI standard for the compiler.

E**elements of array '*identifier*' have incomplete type**

Array elements should have complete types.

empty character constant

Empty character constants are not legal.

empty file name in '#*keyword*'

The file name specified as an argument of the specified #*keyword* is empty.

empty index range in initializer

Do not use empty index ranges in initializers

empty scalar initializer

Scalar initializers must not be empty.

enumerator value for '*identifier*' not integer constant

Enumerator values must be integer constants.

error closing '*file*'

The system cannot close the specified '*file*'. Possible causes are not enough disk space to write to the file or the file is too big.

error writing to '*file*'

The system cannot write to the specified '*file*'. Possible causes are not enough disk space to write to the file or the file is too big.

excess elements in char array initializer

There are more elements in the list than the initializer value states.

excess elements in struct initializer

Do not use excess elements in structure initializers.

expression statement has incomplete type

The type of the expression is incomplete.

extra brace group at end of initializer

Do not place extra brace groups at the end of initializers.

extraneous argument to 'option' option

There are too many arguments to the specified command-line option.

F**'identifier' fails to be a typedef or built in type**

A data type must be a typedef or built-in type.

field 'identifier' declared as a function

Fields may not be declared as functions.

field 'identifier' has incomplete type

Fields must have complete types.

first argument to __builtin_choose_expr not a constant

The first argument must be a constant expression that can be determined at compile time.

flexible array member in otherwise empty struct

A flexible array member must be the last element of a structure with more than one named member.

flexible array member in union

A flexible array member cannot be used in a union.

flexible array member not at end of struct

A flexible array member must be the last element of a structure.

'for' loop initial declaration used outside C99 mode

A 'for' loop initial declaration is not valid outside C99 mode.

format string arg follows the args to be formatted

The arguments to the format attribute are inconsistent. The format string argument index must be less than the index of the first argument to check.

format string arg not a string type

The format string index argument of the format attribute specifies a parameter which is not a string type.

format string has invalid operand number

The operand number argument of the format attribute must be a compile-time constant.

function definition declared 'register'

Function definitions may not be declared 'register'.

function definition declared 'typedef'

Function definitions may not be declared 'typedef'.

function does not return string type

The format_arg attribute may only be used with a function which return value is a string type.

function 'identifier' is initialized like a variable

It is not legal to initialize a function like a variable.

function return type cannot be function

The return type of a function cannot be a function.

G

global register variable follows a function definition

Global register variables should precede function definitions.

global register variable has initial value

Do not specify an initial value for a global register variable.

global register variable '*identifier*' used in nested function

Do not use a global register variable in a nested function.

H

'*identifier*' has an incomplete type

It is not legal to have an incomplete type for the specified '*identifier*'.

'*identifier*' has both 'extern' and initializer

A variable declared 'extern' cannot be initialized.

hexadecimal floating constants require an exponent

Hexadecimal floating constants must have exponents.

I

implicit declaration of function '*identifier*'

The function identifier is used without a preceding prototype declaration or function definition.

impossible register constraint in 'asm'

The asm statement is invalid.

incompatible type for argument *n* of '*identifier*'

When calling functions in C, ensure that actual argument types match the formal parameter types.

incompatible type for argument *n* of indirect function call

When calling functions in C, ensure that actual argument types match the formal parameter types.

incompatible types in *operation*

The types used in *operation* must be compatible.

incomplete '*name*' option

The option to the command-line parameter *name* is incomplete.

inconsistent operand constraints in an 'asm'

The asm statement is invalid.

increment of pointer to unknown structure

Do not increment a pointer to an unknown structure.

initializer element is not computable at load time

Initializer elements must be computable at load time.

initializer element is not constant

Initializer elements must be constant.

initializer fails to determine size of '*identifier*'

An array initializer fails to determine its size.

initializer for static variable is not constant

Static variable initializers must be constant.

initializer for static variable uses complicated arithmetic

Static variable initializers should not use complicated arithmetic.

input operand constraint contains '*constraint*'

The specified constraint is not valid for an input operand.

int-array initialized from non-wide string

Int-arrays should not be initialized from non-wide strings.

interrupt functions must not take parameters

An interrupt function cannot receive parameters. *void* must be used to state explicitly that the argument list is empty.

interrupt functions must return void

An interrupt function must have a return type of *void*. No other return type is allowed.

interrupt modifier '*name*' unknown

The compiler was expecting '*irq*', '*altirq*' or '*save*' as an interrupt attribute modifier.

interrupt modifier syntax error

There is a syntax error with the interrupt attribute modifier.

interrupt pragma must have file scope

#pragma interrupt must be at file scope.

interrupt save modifier syntax error

There is a syntax error with the '*save*' modifier of the interrupt attribute.

interrupt vector is not a constant

The interrupt vector number must be an integer constant.

interrupt vector number *n* is not valid

A valid interrupt vector number is required.

invalid #ident directive

#ident should be followed by a quoted string literal.

invalid arg to '*__builtin_frame_address*'

The argument should be the level of the caller of the function (where 0 yields the frame address of the current function, 1 yields the frame address of the caller of the current function, and so on) and is an integer literal.

invalid arg to '*__builtin_return_address*'

The level argument must be an integer literal.

invalid argument for '*name*'

The compiler was expecting '*data*' or '*prog*' as the space attribute parameter.

invalid character '*character*' in #if

This message appears when an unprintable character, such as a control character, appears after #if.

invalid initial value for member '*name*'

Bit-field '*name*' can only be initialized by an integer.

invalid initializer

Do not use invalid initializers.

Invalid location qualifier: '*symbol*'

Expecting '*sfr*' or '*gpr*', which are ignored on dsPIC DSC devices, as location qualifiers.

invalid operands to binary '*operator*'

The operands to the specified binary operator are invalid.

Invalid option '*option*'

The specified command-line option is invalid.

Invalid option '*symbol*' to interrupt pragma

Expecting shadow and/or save as options to interrupt pragma.

Invalid option to interrupt pragma

Garbage at the end of the pragma.

Invalid or missing function name from interrupt pragma

The interrupt pragma requires the name of the function being called.

Invalid or missing section name

The section name must start with a letter or underscore ('_') and be followed by a sequence of letters, underscores and/or numbers. The names '*access*', '*shared*' and '*overlay*' have special meaning.

invalid preprocessing directive #'*directive*'

Not a valid preprocessing directive. Check the spelling.

invalid preprologue argument

The preprologue option is expecting an assembly statement or statements for its argument enclosed in double quotes.

invalid register name for '*name*'

File scope variable '*name*' declared as a register variable with an illegal register name.

invalid register name '*name*' for register variable

The specified *name* is not the name of a register.

invalid save variable in interrupt pragma

Expecting a symbol or symbols to save.

invalid storage class for function '*identifier*'

Functions may not have the 'register' storage class.

invalid suffix '*suffix*' on integer constant

Integer constants may be suffixed by the letters 'u', 'U', 'l' and 'L' only.

invalid suffix on floating constant

A floating constant suffix may be 'f', 'F', 'l' or 'L' only. If there are two 'L's, they must be adjacent and the same case.

invalid type argument of '*operator*'

The type of the argument to *operator* is invalid.

invalid type modifier within pointer declarator

Only `const` or `volatile` may be used as type modifiers within a pointer declarator.

invalid use of array with unspecified bounds

Arrays with unspecified bounds must be used in valid ways.

invalid use of incomplete typedef '*typedef*'

The specified *typedef* is being used in an invalid way; this is not allowed.

invalid use of undefined type '*type identifier*'

The specified *type* is being used in an invalid way; this is not allowed.

invalid use of void expression

Void expressions must not be used.

"*name*" is not a valid filename

#line requires a valid file name.

'*filename*' is too large

The specified file is too large to process the file. Its probably larger than 4 GB, and the preprocessor refuses to deal with such large files. It is required that files be less than 4 GB in size.

ISO C forbids data definition with no type or storage class

A type specifier or storage class specifier is required for a data definition in ISO C.

ISO C requires a named argument before '*...*'

ISO C requires a named argument before '*...*'.

L**label *label* referenced outside of any function**

Labels may only be referenced inside functions.

label '*label*' used but not defined

The specified *label* is used but is not defined.

language '*name*' not recognized

Permissible languages include: c assembler none.

***filename*: linker input file unused because linking not done**

The specified *filename* was specified on the command line, and it was taken to be a linker input file (since it was not recognized as anything else). However, the link step was not run. Therefore, this file was ignored.

long long long is too long for GCC

The compiler supports integers no longer than `long long`.

long or short specified with char for '*identifier*'

The long and short qualifiers cannot be used with the char type.

long or short specified with floating type for '*identifier*'

The long and short qualifiers cannot be used with the float type.

long, short, signed or unsigned invalid for '*identifier*'

The long, short and signed qualifiers may only be used with integral types.

M**macro names must be identifiers**

Macro names must start with a letter or underscore followed by more letters, numbers or underscores.

macro parameters must be comma-separated

Commas are required between parameters in a list of parameters.

macro '*name*' passed *n* arguments, but takes just *n*

Too many arguments were passed to macro '*name*'.

macro '*name*' requires *n* arguments, but only *n* given

Not enough arguments were passed to macro '*name*'.

matching constraint not valid in output operand

The asm statement is invalid.

'*symbol*' may not appear in macro parameter list

'*symbol*' is not allowed as a parameter.

Missing '=' for 'save' in interrupt pragma

The save parameter requires an equal sign before the variable(s) are listed. For example, `#pragma interrupt isr0 save=var1,var2`

missing '(' after predicate

`#assert` or `#unassert` expects parentheses around the answer. For example:

`ns#assert PREDICATE (ANSWER)`

missing '(' in expression

Parentheses are not matching, expecting an opening parenthesis.

missing ')' after "defined"

Expecting a closing parenthesis.

missing ')' in expression

Parentheses are not matching, expecting a closing parenthesis.

missing ')' in macro parameter list

The macro is expecting parameters to be within parentheses and separated by commas.

missing ')' to complete answer

`#assert` or `#unassert` expects parentheses around the answer.

missing argument to '*option*' option

The specified command-line option requires an argument.

missing binary operator before token '*token*'

Expecting an operator before the '*token*'.

missing terminating '*character*' character

Missing terminating character such as a single quote '*character*', double quote "*character*" or right angle bracket *character* >.

missing terminating > character

Expecting terminating > in `#include` directive.

more than *n* operands in 'asm'

The asm statement is invalid.

multiple default labels in one switch

Only a single default label may be specified for each switch.

multiple parameters named '*identifier*'

Parameter names must be unique.

multiple storage classes in declaration of '*identifier*'

Each declaration should have a single storage class.

N

negative width in bit-field '*identifier*'

Bit-field widths may not be negative.

nested function '*name*' *declared* '*extern*'

A nested function cannot be declared '*extern*'.

nested redefinition of '*identifier*'

Nested redefinitions are illegal.

no data type for mode '*mode*'

The argument mode specified for the mode attribute is a recognized GCC machine mode, but it is not one that is implemented in the compiler.

no include path in which to find '*name*'

Cannot find include file '*name*'.

no macro name given in #'*directive*' directive

A macro name must follow the #define, #undef, #ifdef or #ifndef directives.

nonconstant array index in initializer

Only constant array indices may be used in initializers.

non-prototype definition here

If a function prototype follows a definition without a prototype, and the number of arguments is inconsistent between the two, this message identifies the line number of the non-prototype definition.

number of arguments doesn't match prototype

The number of function arguments must match the function's prototype.

O

operand constraint contains incorrectly positioned '+' or '='.

The asm statement is invalid.

operand constraints for '*asm*' differ in number of alternatives

The asm statement is invalid.

operator "*defined*" requires an identifier

"defined" is expecting an identifier.

operator '*symbol*' has no right operand

Preprocessor operator '*symbol*' requires an operand on the right side.

output number *n* not directly addressable

The asm statement is invalid.

output operand constraint lacks '='

The asm statement is invalid.

output operand is constant in '*asm*'

The asm statement is invalid.

overflow in enumeration values

Enumeration values must be in the range of 'int'.

P

parameter '*identifier*' declared void

Parameters may not be declared void.

parameter '*identifier*' has incomplete type

Parameters must have complete types.

parameter '*identifier*' has just a forward declaration

Parameters must have complete types; forward declarations are insufficient.

parameter '*identifier*' is initialized

It is not legal to initialize parameters.

parameter name missing

The macro was expecting a parameter name. Check for two commas without a name between.

parameter name missing from parameter list

Parameter names must be included in the parameter list.

parameter name omitted

Parameter names may not be omitted.

param types given both in param list and separately

Parameter types should be given either in the parameter list or separately, but not both.

parse error

The source line cannot be parsed; it contains errors.

pointer value used where a complex value was expected

Do not use pointer values where complex values are expected.

pointer value used where a floating point value was expected

Do not use pointer values where floating-point values are expected.

pointers are not permitted as case values

A case value must be an integer-valued constant or constant expression.

predicate must be an identifier

`#assert` or `#unassert` require a single identifier as the predicate.

predicate's answer is empty

The `#assert` or `#unassert` has a predicate and parentheses but no answer inside the parentheses, which is required.

previous declaration of '*identifier*'

This message identifies the location of a previous declaration of identifier that conflicts with the current declaration.

***identifier* previously declared here**

This message identifies the location of a previous declaration of identifier that conflicts with the current declaration.

***identifier* previously defined here**

This message identifies the location of a previous definition of identifier that conflicts with the current definition.

prototype declaration

Identifies the line number where a function prototype is declared. Used in conjunction with other error messages.

R

redeclaration of '*identifier*'

The *identifier* is multiply declared.

redeclaration of '*enum identifier*'

Enums may not be redeclared.

'*identifier*' redeclared as different kind of symbol

Multiple, inconsistent declarations exist for *identifier*.

redefinition of '*identifier*'

The *identifier* is multiply defined.

redefinition of '*struct identifier*'

Structs may not be redefined.

redefinition of '*union identifier*'

Unions may not be redefined.

register name given for non-register variable '*name*'

Attempt to map a register to a variable which is not marked as register.

register name not specified for '*name*'

File scope variable '*name*' declared as a register variable without providing a register.

register specified for '*name*' isn't suitable for data type

Alignment or other restrictions prevent using requested register.

request for member '*identifier*' in something not a structure or union

Only structure or unions have members. It is not legal to reference a member of anything else, since nothing else has members.

requested alignment is not a constant

The argument to the aligned attribute must be a compile-time constant.

requested alignment is not a power of 2

The argument to the aligned attribute must be a power of two.

requested alignment is too large

The alignment size requested is larger than the linker allows. The size must be 4096 or less and a power of 2.

return type is an incomplete type

Return types must be complete.

S

save variable '*name*' index not constant

The subscript of the array '*name*' is not a constant integer.

save variable '*name*' is not word aligned

The object being saved must be word aligned

save variable '*name*' size is not even

The object being saved must be evenly sized.

save variable '*name*' size is not known

The object being saved must have a known size.

section attribute cannot be specified for local variables

Local variables are always allocated in registers or on the stack. It is therefore not legal to attempt to place local variables in a named section.

section attribute not allowed for *identifier*

The section attribute may only be used with a function or variable.

section of *identifier* conflicts with previous declaration

If multiple declarations of the same *identifier* specify the section attribute, then the value of the attribute must be consistent.

sfr address '*address*' is not valid

The address must be less than 0x2000 to be valid.

sfr address is not a constant

The sfr address must be a constant.

'size of' applied to a bit-field

'sizeof' must not be applied to a bit-field.

size of array '*identifier*' has non-integer type

Array size specifiers must be of integer type.

size of array '*identifier*' is negative

Array sizes may not be negative.

size of array '*identifier*' is too large

The specified array is too large.

size of variable '*variable*' is too large

The maximum size of the variable can be 32768 bytes.

storage class specified for parameter '*identifier*'

A storage class may not be specified for a parameter.

storage size of '*identifier*' isn't constant

Storage size must be compile-time constants.

storage size of '*identifier*' isn't known

The size of *identifier* is incompletely specified.

stray '*character*' in program

Do not place stray '*character*' characters in the source program.

strftime formats cannot format arguments

While using the attribute format when the archetype parameter is strftime, the third parameter to the attribute, which specifies the first parameter to match against the format string, should be 0. strftime style functions do not have input values to match against a format string.

structure has no member named '*identifier*'

A structure member named '*identifier*' is referenced; but the referenced structure contains no such member. This is not allowed.

subscripted value is neither array nor pointer

Only arrays or pointers may be subscripted.

switch quantity not an integer

Switch quantities must be integers

symbol '*symbol*' not defined

The symbol '*symbol*' needs to be declared before it may be used in the pragma.

syntax error

A syntax error exists on the specified line.

syntax error ':' without preceding '?'

A ':' must be preceded by '?' in the '?:' operator.

T**the only valid combination is 'long double'**

The long qualifier is the only qualifier that may be used with the double type.

this built-in requires a frame pointer

`__builtin_return_address` requires a frame pointer. Do not use the `-fomit-frame-pointer` option.

this is a previous declaration

If a label is duplicated, this message identifies the line number of a preceding declaration.

too few arguments to function

When calling a function in C, do not specify fewer arguments than the function requires. Nor should you specify too many.

too few arguments to function '*identifier*'

When calling a function in C, do not specify fewer arguments than the function requires. Nor should you specify too many.

too many alternatives in 'asm'

The asm statement is invalid.

too many arguments to function

When calling a function in C, do not specify more arguments than the function requires. Nor should you specify too few.

too many arguments to function '*identifier*'

When calling a function in C, do not specify more arguments than the function requires. Nor should you specify too few.

too many decimal points in number

Expecting only one decimal point.

top-level declaration of '*identifier*' specifies 'auto'

Auto variables can only be declared inside functions.

two or more data types in declaration of '*identifier*'

Each identifier may have only a single data type.

two types specified in one empty declaration

No more than one type should be specified.

type of formal parameter *n* is incomplete

Specify a complete type for the indicated parameter.

type mismatch in conditional expression

Types in conditional expressions must not be mismatched.

typedef '*identifier*' is initialized

It is not legal to initialize typedef's. Use `__typeof` instead.

U

'*identifier*' undeclared (first use in this function)

The specified identifier must be declared.

'*identifier*' undeclared here (not in a function)

The specified identifier must be declared.

union has no member named '*identifier*'

A union member named '*identifier*' is referenced, but the referenced union contains no such member. This is not allowed.

unknown field '*identifier*' specified in initializer

Do not use unknown fields in initializers.

unknown machine mode '*mode*'

The argument *mode* specified for the mode attribute is not a recognized machine mode.

unknown register name '*name*' in 'asm'

The asm statement is invalid.

unrecognized format specifier

The argument to the format attribute is invalid.

unrecognized option '*-option*'

The specified command-line option is not recognized.

unrecognized option '*option*'

'*option*' is not a known option.

'*identifier*' used prior to declaration

The identifier is used prior to its declaration.

unterminated #'*name*'

#endif is expected to terminate a #if, #ifdef or #ifndef conditional.

unterminated argument list invoking macro '*name*'

Evaluation of a function macro has encountered the end of file before completing the macro expansion.

unterminated comment

The end of file was reached while scanning for a comment terminator.

V

'va_start' used in function with fixed args

'va_start' should be used only in functions with variable argument lists.

variable '*identifier*' has initializer but incomplete type

It is not legal to initialize variables with incomplete types.

variable or field '*identifier*' declared void

Neither variables nor fields may be declared void.

variable-sized object may not be initialized

It is not legal to initialize a variable-sized object.

virtual memory exhausted

Not enough memory left to write error message.

void expression between '(' and ')'

Expecting a constant expression but found a void expression between the parentheses.

'void' in parameter list must be the entire list

If 'void' appears as a parameter in a parameter list, then there must be no other parameters.

void value not ignored as it ought to be

The value of a void function should not be used in an expression.

W**warning: -pipe ignored because -save-temps specified**

The -pipe option cannot be used with the -save-temps option.

warning: -pipe ignored because -time specified

The -pipe option cannot be used with the -time option.

warning: '-x spec' after last input file has no effect

The '-x' command line option affects only those files named after its on the command line; if there are no such files, then this option has no effect.

weak declaration of 'name' must be public

Weak symbols must be externally visible.

weak declaration of 'name' must precede definition

'name' was defined and then declared weak.

wrong number of arguments specified for *attribute* attribute

There are too few or too many arguments given for the attribute named '*attribute*'.

wrong type argument to bit-complement

Do not use the wrong type of argument to this operator.

wrong type argument to decrement

Do not use the wrong type of argument to this operator.

wrong type argument to increment

Do not use the wrong type of argument to this operator.

wrong type argument to unary exclamation mark

Do not use the wrong type of argument to this operator.

wrong type argument to unary minus

Do not use the wrong type of argument to this operator.

wrong type argument to unary plus

Do not use the wrong type of argument to this operator.

Z**zero width for bit-field '*identifier*'**

Bit-fields may not have zero width.

C.3 WARNINGS

Symbols

'/*' within comment

A comment mark was found within a comment.

'\$' character(s) in identifier or number

Dollar signs in identifier names are an extension to the standard.

#'directive' is a GCC extension

`#warning`, `#include_next`, `#ident`, `#import`, `#assert` and `#unassert` directives are GCC extensions and are not of ISO C89.

#import is obsolete, use an #ifndef wrapper in the header file

The `#import` directive is obsolete. `#import` was used to include a file if it hadn't already been included. Use the `#ifndef` directive instead.

#include_next in primary source file

`#include_next` starts searching the list of header file directories after the directory in which the current file was found. In this case, there were no previous header files so it is starting in the primary source file.

#pragma pack (pop) encountered without matching #pragma pack (push, <n>)

The `pack(pop)` pragma must be paired with a `pack(push)` pragma, which must precede it in the source file.

#pragma pack (pop, identifier) encountered without matching #pragma pack (push, identifier, <n>)

The `pack(pop)` pragma must be paired with a `pack(push)` pragma, which must precede it in the source file.

#warning: message

The directive `#warning` causes the preprocessor to issue a warning and continue preprocessing. The tokens following `#warning` are used as the warning message.

A

absolute address specification ignored

Ignoring the absolute address specification for the code section in the `#pragma` statement because it is not supported in the compiler. Addresses must be specified in the linker script and code sections can be defined with the keyword `__attribute__`.

address of register variable 'name' requested

The register specifier prevents taking the address of a variable.

alignment must be a small power of two, not n

The alignment parameter of the `pack` pragma must be a small power of two.

anonymous enum declared inside parameter list

An anonymous enum is declared inside a function parameter list. It is usually better programming practice to declare enums outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous struct declared inside parameter list

An anonymous struct is declared inside a function parameter list. It is usually better programming practice to declare structs outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous union declared inside parameter list

An anonymous union is declared inside a function parameter list. It is usually better programming practice to declare unions outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous variadic macros were introduced in C99

Macros which accept a variable number of arguments is a C99 feature.

argument '*identifier*' might be clobbered by 'longjmp' or 'vfork'

An argument might be changed by a call to longjmp. These warnings are possible only in optimizing compilation.

array '*identifier*' assumed to have one element

The length of the specified array was not explicitly stated. In the absence of information to the contrary, the compiler assumes that it has one element.

array subscript has type 'char'

An array subscript has type '*char*'.

array type has incomplete element type

Array types should not have incomplete element types.

asm operand *n* probably doesn't match constraints

The specified extended asm operand probably doesn't match its constraints.

assignment of read-only member '*name*'

The member '*name*' was declared as const and cannot be modified by assignment.

assignment of read-only variable '*name*'

'*name*' was declared as const and cannot be modified by assignment.

'*identifier*' attribute directive ignored

The named attribute is not a known or supported attribute, and is therefore ignored.

'*identifier*' attribute does not apply to types

The named attribute may not be used with types. It is ignored.

'*identifier*' attribute ignored

The named attribute is not meaningful in the given context, and is therefore ignored.

'*attribute*' attribute only applies to function types

The specified attribute can only be applied to the return types of functions and not to other declarations.

B

backslash and newline separated by space

While processing for escape sequences, a backslash and newline were found separated by a space.

backslash-newline at end of file

While processing for escape sequences, a backslash and newline were found at the end of the file.

bit-field '*identifier*' type invalid in ISO C

The type used on the specified identifier is not valid in ISO C.

braces around scalar initializer

A redundant set of braces around an initializer is supplied.

built-in function '*identifier*' declared as non-function

The specified function has the same name as a built-in function, yet is declared as something other than a function.

C

C++ style comments are not allowed in ISO C89

Use C style comments `/*` and `*/` instead of C++ style comments `/*`.

call-clobbered register used for global register variable

Choose a register that is normally saved and restored by function calls (W8-W13), so that library routines will not clobber it.

cannot inline function '*main*'

The function '*main*' is declared with the *inline* attribute. This is not supported, since *main* must be called from the C start-up code, which is compiled separately.

can't inline call to '*identifier*' called from here

The compiler was unable to inline the call to the specified function.

case value '*n*' not in enumerated type

The controlling expression of a switch statement is an enumeration type, yet a case expression has the value *n*, which does not correspond to any of the enumeration values.

case value '*value*' not in enumerated type '*name*'

'value' is an extra switch case that is not an element of the enumerated type *'name'*.

cast does not match function type

The return type of a function is cast to a type that does not match the function's type.

cast from pointer to integer of different size

A pointer is cast to an integer that is not 16 bits wide.

cast increases required alignment of target type

When compiling with the `-Wcast-align` command-line option, the compiler verifies that casts do not increase the required alignment of the target type. For example, this warning message will be given if a pointer to char is cast as a pointer to int, since the aligned for char (byte alignment) is less than the alignment requirement for int (word alignment).

character constant too long

Character constants must not be too long.

comma at end of enumerator list

Unnecessary comma at the end of the enumerator list.

comma operator in operand of #if

Not expecting a comma operator in the `#if` directive.

comparing floating point with == or != is unsafe

Floating-point values can be approximations to infinitely precise real numbers. Instead of testing for equality, use relational operators to see whether the two values have ranges that overlap.

comparison between pointer and integer

A pointer type is being compared to an integer type.

comparison between signed and unsigned

One of the operands of a comparison is signed, while the other is unsigned. The signed operand will be treated as an unsigned value, which may not be correct.

comparison is always n

A comparison involves only constant expressions, so the compiler can evaluate the run time result of the comparison. The result is always n .

comparison is always n due to width of bit-field

A comparison involving a bit-field always evaluates to n because of the width of the bit-field.

comparison is always false due to limited range of data type

A comparison will always evaluate to false at run time, due to the range of the data types.

comparison is always true due to limited range of data type

A comparison will always evaluate to true at run time, due to the range of the data types.

comparison of promoted ~unsigned with constant

One of the operands of a comparison is a promoted ~unsigned, while the other is a constant.

comparison of promoted ~unsigned with unsigned

One of the operands of a comparison is a promoted ~unsigned, while the other is unsigned.

comparison of unsigned expression ≥ 0 is always true

A comparison expression compares an unsigned value with zero. Since unsigned values cannot be less than zero, the comparison will always evaluate to true at run time.

comparison of unsigned expression < 0 is always false

A comparison expression compares an unsigned value with zero. Since unsigned values cannot be less than zero, the comparison will always evaluate to false at run time.

comparisons like $X \leq Y \leq Z$ do not have their mathematical meaning

A C expression does not necessarily mean the same thing as the corresponding mathematical expression. In particular, the C expression $X \leq Y \leq Z$ is not equivalent to the mathematical expression $X \leq Y \leq Z$.

conflicting types for built-in function '*identifier*'

The specified function has the same name as a built-in function but is declared with conflicting types.

const declaration for '*identifier*' follows non-const

The specified identifier was declared const after it was previously declared as non-const.

control reaches end of non-void function

All exit paths from non-void function should return an appropriate value. The compiler detected a case where a non-void function terminates, without an explicit return value. Therefore, the return value might be unpredictable.

conversion lacks type at end of format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that a format field in the format string lacked a type specifier.

concatenation of string literals with `__FUNCTION__` is deprecated

`__FUNCTION__` will be handled the same way as `__func__` (which is defined by the ISO standard C99). `__func__` is a variable, not a string literal, so it does not concatenate with other string literals.

conflicting types for '*identifier*'

The specified identifier has multiple, inconsistent declarations.

D

data definition has no type or storage class

A data definition was detected that lacked a type and storage class.

data qualifier '*qualifier*' ignored

Data qualifiers, which include 'access', 'shared' and 'overlay', are not used in the compiler, but are there for compatibility with the MPLAB C Compiler for PIC18 MCUs.

declaration of '*identifier*' has 'extern' and is initialized

Externs should not be initialized.

declaration of '*identifier*' shadows a parameter

The specified *identifier* declaration shadows a parameter, making the parameter inaccessible.

declaration of '*identifier*' shadows a symbol from the parameter list

The specified identifier declaration shadows a symbol from the parameter list, making the symbol inaccessible.

declaration of '*identifier*' shadows global declaration

The specified *identifier* declaration shadows a global declaration, making the global inaccessible.

'*identifier*' declared inline after being called

The specified function was declared inline after it was called.

'*identifier*' declared inline after its definition

The specified function was declared inline after it was defined.

'*identifier*' declared 'static' but never defined

The specified function was declared static, but was never defined.

decrement of read-only member '*name*'

The member '*name*' was declared as const and cannot be modified by decrementing.

decrement of read-only variable '*name*'

'*name*' was declared as const and cannot be modified by decrementing.

'*identifier*' defined but not used

The specified function was defined, but was never used.

deprecated use of label at end of compound statement

A label should not be at the end of a statement. It should be followed by a statement.

dereferencing 'void **' pointer

It is not correct to dereference a 'void **' pointer. Cast it to a pointer of the appropriate type before dereferencing the pointer.

division by zero

Compile-time division by zero has been detected.

duplicate 'const'

The 'const' qualifier should be applied to a declaration only once.

duplicate 'restrict'

The 'restrict' qualifier should be applied to a declaration only once.

duplicate 'volatile'

The 'volatile' qualifier should be applied to a declaration only once.

E**embedded '\0' in format**

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the format string contains an embedded '\0' (zero), which can cause early termination of format string processing.

empty body in an else-statement

An else statement is empty.

empty body in an if-statement

An if statement is empty.

empty declaration

The declaration contains no names to declare.

empty range specified

The range of values in a case range is empty, that is, the value of the low expression is greater than the value of the high expression. Recall that the syntax for case ranges is `case low... high:`.

'enum identifier' declared inside parameter list

The specified enum is declared inside a function parameter list. It is usually better programming practice to declare enums outside parameter lists, since they can never become complete types when defined inside parameter lists.

enum defined inside parms

An enum is defined inside a function parameter list.

enumeration value 'identifier' not handled in switch

The controlling expression of a switch statement is an enumeration type, yet not all enumeration values have case expressions.

enumeration values exceed range of largest integer

Enumeration values are represented as integers. The compiler detected that an enumeration range cannot be represented in any of the compiler integer formats, including the largest such format.

excess elements in array initializer

There are more elements in the initializer list than the array was declared with.

excess elements in scalar initializer");

There should be only one initializer for a scalar variable.

excess elements in struct initializer

There are more elements in the initializer list than the structure was declared with.

excess elements in union initializer

There are more elements in the initializer list than the union was declared with.

extra semicolon in struct or union specified

The structure type or union type contains an extra semicolon.

extra tokens at end of #‘directive’ directive

The compiler detected extra text on the source line containing the #‘directive’ directive.

F**-ffunction-sections may affect debugging on some targets**

You may have problems with debugging if you specify both the -g option and the -ffunction-sections option.

first argument of ‘identifier’ should be ‘int’

Expecting declaration of first argument of specified identifier to be of type int.

floating constant exceeds range of ‘double’

A floating-point constant is too large or too small (in magnitude) to be represented as a ‘double’.

floating constant exceeds range of ‘float’

A floating-point constant is too large or too small (in magnitude) to be represented as a ‘float’.

floating constant exceeds range of ‘long double’

A floating-point constant is too large or too small (in magnitude) to be represented as a ‘long double’.

floating point overflow in expression

When folding a floating-point constant expression, the compiler found that the expression overflowed, that is, it could not be represented as float.

‘type1’ format, ‘type2’ arg (arg ‘num’)

The format is of type ‘type1’, but the argument being passed is of type ‘type2’.
The argument in question is the ‘num’ argument.

format argument is not a pointer (arg *n*)

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the specified argument number *n* was not a pointer, san the format specifier indicated it should be.

format argument is not a pointer to a pointer (arg *n*)

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the specified argument number *n* was not a pointer san the format specifier indicated it should be.

fprefetch-loop-arrays not supported for this target

The option to generate instructions to prefetch memory is not supported for this target.

function call has aggregate value

The return value of a function is an aggregate.

function declaration isn’t a prototype

When compiling with the -Wstrict-prototypes command-line option, the compiler ensures that function prototypes are specified for all functions. In this case, a function definition was encountered without a preceding function prototype.

function declared ‘noreturn’ has a ‘return’ statement

A function was declared with the noreturn attribute-indicating that the function does not return-yet the function contains a return statement. This is inconsistent.

function might be possible candidate for attribute 'noreturn'

The compiler detected that the function does not return. If the function had been declared with the 'noreturn' attribute, then the compiler might have been able to generate better code.

function returns address of local variable

Functions should not return the addresses of local variables, since, when the function returns, the local variables are de-allocated.

function returns an aggregate

The return value of a function is an aggregate.

function '*name*' redeclared as inline**previous declaration of function '*name*' with attribute noline**

Function '*name*' was declared a second time with the keyword 'inline', which now allows the function to be considered for inlining.

function '*name*' redeclared with attribute noline**previous declaration of function '*name*' was inline**

Function '*name*' was declared a second time with the noline attribute, which now causes it to be ineligible for inlining.

function '*identifier*' was previously declared within a block

The specified function has a previous explicit declaration within a block, yet it has an implicit declaration on the current line.

G

GCC does not yet properly implement '['*']' array declarators

Variable length arrays are not currently supported by the compiler.

H

hex escape sequence out of range

The hex sequence must be less than 100 in hex (256 in decimal).

I

ignoring asm-specifier for non-static local variable '*identifier*'

The asm-specifier is ignored when it is used with an ordinary, non-register local variable.

ignoring invalid multibyte character

When parsing a multibyte character, the compiler determined that it was invalid. The invalid character is ignored.

ignoring option '*option*' due to invalid debug level specification

A debug option was used with a debug level that is not a valid debug level.

ignoring #pragma *identifier*

The specified pragma is not supported by the compiler, and is ignored.

imaginary constants are a GCC extention

ISO C does not allow imaginary numeric constants.

implicit declaration of function '*identifier*'

The specified function has no previous explicit declaration (definition or function prototype), so the compiler makes assumptions about its return type and parameters.

increment of read-only member '*name*'

The member '*name*' was declared as const and cannot be modified by incrementing.

increment of read-only variable '*name*'

'*name*' was declared as const and cannot be modified by incrementing.

initialization of a flexible array member

A flexible array member is intended to be dynamically allocated not statically.

'*identifier*' initialized and declared 'extern'

Externs should not be initialized.

initializer element is not constant

Initializer elements should be constant.

inline function '*name*' given attribute *noinline*

The function '*name*' has been declared as inline, but the *noinline* attribute prevents the function from being considered for inlining.

inlining failed in call to '*identifier*' called from here

The compiler was unable to inline the call to the specified function.

integer constant is so large that it is unsigned

An integer constant value appears in the source code without an explicit unsigned modifier, yet the number cannot be represented as a signed int; therefore, the compiler automatically treats it as an unsigned int.

integer constant is too large for '*type*' type

An integer constant should not exceed $2^{32} - 1$ for an unsigned long int, $2^{63} - 1$ for a long long int or $2^{64} - 1$ for an unsigned long long int.

integer overflow in expression

When folding an integer constant expression, the compiler found that the expression overflowed; that is, it could not be represented as an int.

invalid application of '*sizeof*' to a function type

It is not recommended to apply the *sizeof* operator to a function type.

invalid application of '*sizeof*' to a void type

The *sizeof* operator should not be applied to a void type.

invalid digit '*digit*' in octal constant

All digits must be within the radix being used. For instance, only the digits 0 thru 7 may be used for the octal radix.

invalid second arg to `__builtin_prefetch`; using zero

Second argument must be 0 or 1.

invalid storage class for function '*name*'

'auto' storage class should not be used on a function defined at the top level. 'static' storage class should not be used if the function is not defined at the top level.

invalid third arg to `__builtin_prefetch`; using zero

Third argument must be 0, 1, 2, or 3.

'*identifier*' is an unrecognized format function type

The specified *identifier*, used with the format attribute, is not one of the recognized format function types `printf`, `scanf`, or `strftime`.

'identifier' is narrower than values of its type

A bit-field member of a structure has for its type an enumeration, but the width of the field is insufficient to represent all enumeration values.

'storage class' is not at beginning of declaration

The specified storage class is not at the beginning of the declaration. Storage classes are required to come first in declarations.

ISO C does not allow extra ';' outside of a function

An extra ';' was found outside a function. This is not allowed by ISO C.

ISO C does not support '++' and '--' on complex types

The increment operator and the decrement operator are not supported on complex types in ISO C.

ISO C does not support '~' for complex conjugation

The bitwise negation operator cannot be use for complex conjugation in ISO C.

ISO C does not support complex integer types

Complex integer types, such as `__complex__ short int`, are not supported in ISO C.

ISO C does not support plain 'complex' meaning 'double complex'

Using `__complex__` without another modifier is equivalent to 'complex double' which is not supported in ISO C.

ISO C does not support the 'char' 'kind of format' format

ISO C does not support the specification character 'char' for the specified 'kind of format'.

ISO C doesn't support unnamed structs/unions

All structures and/or unions must be named in ISO C.

ISO C forbids an empty source file

The file contains no functions or data. This is not allowed in ISO C.

ISO C forbids empty initializer braces

ISO C expects initializer values inside the braces.

ISO C forbids nested functions

A function has been defined inside another function.

ISO C forbids omitting the middle term of a ?: expression

The conditional expression requires the middle term or expression between the '?' and the ':'.
The conditional expression requires the middle term or expression between the '?' and the ':'.

ISO C forbids qualified void function return type

A qualifier may not be used with a void function return type.

ISO C forbids range expressions in switch statements

Specifying a range of consecutive values in a single case label is not allowed in ISO C.

ISO C forbids subscripting 'register' array

Subscripting a 'register' array is not allowed in ISO C.

ISO C forbids taking the address of a label

Taking the address of a label is not allowed in ISO C.

ISO C forbids zero-size array 'name'

The array size of 'name' must be larger than zero.

ISO C restricts enumerator values to range of 'int'

The range of enumerator values must not exceed the range of the int type.

ISO C89 forbids compound literals

Compound literals are not valid in ISO C89.

ISO C89 forbids mixed declarations and code

Declarations should be done first before any code is written. It should not be mixed in with the code.

ISO C90 does not support '[' array declarators

Variable length arrays are not supported in ISO C90.

ISO C90 does not support complex types

Complex types, such as `__complex__ float x`, are not supported in ISO C90.

ISO C90 does not support flexible array members

A flexible array member is a new feature in C99. ISO C90 does not support it.

ISO C90 does not support 'long long'

The `long long` type is not supported in ISO C90.

ISO C90 does not support 'static' or type qualifiers in parameter array declarators

When using an array as a parameter to a function, ISO C90 does not allow the array declarator to use 'static' or type qualifiers.

ISO C90 does not support the 'char' 'function' format

ISO C does not support the specification character 'char' for the specified function format.

ISO C90 does not support the 'modifier' 'function' length modifier

The specified modifier is not supported as a length modifier for the given function.

ISO C90 forbids variable-size array 'name'

In ISO C90, the number of elements in the array must be specified by an integer constant expression.

L

label 'identifier' defined but not used

The specified label was defined, but not referenced.

large integer implicitly truncated to unsigned type

An integer constant value appears in the source code without an explicit unsigned modifier, yet the number cannot be represented as a signed int; therefore, the compiler automatically treats it as an unsigned int.

left-hand operand of comma expression has no effect

One of the operands of a comparison is a promoted ~unsigned, while the other is unsigned.

left shift count >= width of type

Shift counts should be less than the number of bits in the type being shifted. Otherwise, the shift is meaningless, and the result is undefined.

left shift count is negative

Shift counts should be positive. A negative left shift count does not mean shift right; it is meaningless.

library function '*identifier*' declared as non-function

The specified function has the same name as a library function, yet is declared as something other than a function.

line number out of range

The limit for the line number for a #line directive in C89 is 32767 and in C99 is 2147483647.

'*identifier*' locally external but globally static

The specified *identifier* is locally external but globally static. This is suspect.

location qualifier '*qualifier*' ignored

Location qualifiers, which include 'grp' and 'sfr', are not used in the compiler, but are there for compatibility with MPLAB C Compiler for PIC18 MCUs.

'long' switch expression not converted to 'int' in ISO C

ISO C does not convert 'long' switch expressions to 'int'.

M

'main' is usually a function

The identifier main is usually used for the name of the main entry point of an application. The compiler detected that it was being used in some other way, for example, as the name of a variable.

'*operation*' makes integer from pointer without a cast

A pointer has been implicitly converted to an integer.

'*operation*' makes pointer from integer without a cast

An integer has been implicitly converted to a pointer.

malformed '#pragma pack-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma pack(pop[,id])-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma pack(push[,id],<n>)-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma weak-ignored'

The syntax of the weak pragma is incorrect.

'*identifier*' might be used uninitialized in this function

The compiler detected a control path through a function which might use the specified identifier before it has been initialized.

missing braces around initializer

A required set of braces around an initializer is missing.

missing initializer

An initializer is missing.

modification by 'asm' of read-only variable '*identifier*'

A const variable is the left-hand-side of an assignment in an 'asm' statement.

multi-character *character* constant

A character constant contains more than one character.

N

negative integer implicitly converted to unsigned type

A negative integer constant value appears in the source code, but the number cannot be represented as a signed int; therefore, the compiler automatically treats it as an unsigned int.

nested extern declaration of '*identifier*'

There are nested extern definitions of the specified *identifier*.

no newline at end of file

The last line of the source file is not terminated with a newline character.

no previous declaration for '*identifier*'

When compiling with the `-Wmissing-declarations` command-line option, the compiler ensures that functions are declared before they are defined. In this case, a function definition was encountered without a preceding function declaration.

no previous prototype for '*identifier*'

When compiling with the `-Wmissing-prototypes` command-line option, the compiler ensures that function prototypes are specified for all functions. In this case, a function definition was encountered without a preceding function prototype.

no semicolon at end of struct or union

A semicolon is missing at the end of the structure or union declaration.

non-ISO-standard escape sequence, '*seq*'

'*seq*' is '\e' or '\E' and is an extension to the ISO standard. The sequence can be used in a string or character constant and stands for the ASCII character <ESC>.

non-static declaration for '*identifier*' follows static

The specified identifier was declared non-static after it was previously declared as static.

'noreturn' function does return

A function declared with the *noreturn* attribute returns. This is inconsistent.

'noreturn' function returns non-void value

A function declared with the *noreturn* attribute returns a non-void value. This is inconsistent.

null format string

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the format string was missing.

O

octal escape sequence out of range

The octal sequence must be less than 400 in octal (256 in decimal).

output constraint '*constraint*' for operand *n* is not at the beginning

Output constraints in extended asm should be at the beginning.

overflow in constant expression

The constant expression has exceeded the range of representable values for its type.

overflow in implicit constant conversion

An implicit constant conversion resulted in a number that cannot be represented as a signed int; therefore, the compiler automatically treats it as an unsigned int.

P

parameter has incomplete type

A function parameter has an incomplete type.

parameter names (without types) in function declaration

The function declaration lists the names of the parameters but not their types.

parameter points to incomplete type

A function parameter points to an incomplete type.

parameter '*identifier*' points to incomplete type

The specified function parameter points to an incomplete type.

passing arg '*number*' of '*name*' as complex rather than floating due to prototype

The prototype declares argument '*number*' as a complex, but a float value is used so the compiler converts to a complex to agree with the prototype.

passing arg '*number*' of '*name*' as complex rather than integer due to prototype

The prototype declares argument '*number*' as a complex, but an integer value is used so the compiler converts to a complex to agree with the prototype.

passing arg '*number*' of '*name*' as floating rather than complex due to prototype

The prototype declares argument '*number*' as a float, but a complex value is used so the compiler converts to a float to agree with the prototype.

passing arg '*number*' of '*name*' as 'float' rather than 'double' due to prototype

The prototype declares argument '*number*' as a float, but a double value is used so the compiler converts to a float to agree with the prototype.

passing arg '*number*' of '*name*' as floating rather than integer due to prototype

The prototype declares argument '*number*' as a float, but an integer value is used so the compiler converts to a float to agree with the prototype.

passing arg '*number*' of '*name*' as integer rather than complex due to prototype

The prototype declares argument '*number*' as an integer, but a complex value is used so the compiler converts to an integer to agree with the prototype.

passing arg '*number*' of '*name*' as integer rather than floating due to prototype

The prototype declares argument '*number*' as an integer, but a float value is used so the compiler converts to an integer to agree with the prototype.

pointer of type '*void **' used in arithmetic

A pointer of type '*void **' has no size and should not be used in arithmetic.

pointer to a function used in arithmetic

A pointer to a function should not be used in arithmetic.

previous declaration of '*identifier*'

This warning message appears in conjunction with another warning message. The previous message identifies the location of the suspect code. This message identifies the first declaration or definition of the *identifier*.

previous implicit declaration of '*identifier*'

This warning message appears in conjunction with the warning message "type mismatch with previous implicit declaration". It locates the implicit declaration of the identifier that conflicts with the explicit declaration.

R

“*name*” reasserted

The answer for “*name*” has been duplicated.

“*name*” redefined

“*name*” was previously defined and is being redefined now.

redefinition of ‘*identifier*’

The specified identifier has multiple, incompatible definitions.

redundant redeclaration of ‘*identifier*’ in same scope

The specified identifier was re-declared in the same scope. This is redundant.

register used for two global register variables

Two global register variables have been defined to use the same register.

repeated ‘*flag*’ flag in format

When checking the argument list of a call to *strftime*, the compiler found that there was a flag in the format string that is repeated.

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the flags { ,+,#,0,- } was repeated in the format string.

return-type defaults to ‘int’

In the absence of an explicit function return-type declaration, the compiler assumes that the function returns an int.

return type of ‘*name*’ is not ‘int’

The compiler is expecting the return type of ‘*name*’ to be ‘int’.

‘return’ with a value, in function returning void

The function was declared as void but returned a value.

‘return’ with no value, in function returning non-void

A function declared to return a non-void value contains a return statement with no value. This is inconsistent.

right shift count \geq width of type

Shift counts should be less than the number of bits in the type being shifted. Otherwise, the shift is meaningless, and the result is undefined.

right shift count is negative

Shift counts should be positive. A negative right shift count does not mean shift left; it is meaningless.

S

second argument of ‘*identifier*’ should be ‘char **’

Expecting second argument of specified identifier to be of type ‘char **’.

second parameter of ‘va_start’ not last named argument

The second parameter of ‘va_start’ must be the last named argument.

shadowing built-in function ‘*identifier*’

The specified function has the same name as a built-in function, and consequently shadows the built-in function.

shadowing library function ‘*identifier*’

The specified function has the same name as a library function, and consequently shadows the library function.

shift count \geq width of type

Shift counts should be less than the number of bits in the type being shifted. Otherwise, the shift is meaningless, and the result is undefined.

shift count is negative

Shift counts should be positive. A negative left shift count does not mean shift right, nor does a negative right shift count mean shift left; they are meaningless.

size of 'name' is larger than n bytes

Using `-Wlarger-than-len` will produce the above warning when the size of 'name' is larger than the len bytes defined.

size of 'identifier' is n bytes

The size of the specified identifier (which is n bytes) is larger than the size specified with the `-Wlarger-than-len` command-line option.

size of return value of 'name' is larger than n bytes

Using `-Wlarger-than-len` will produce the above warning when the size of the return value of 'name' is larger than the len bytes defined.

size of return value of 'identifier' is n bytes

The size of the return value of the specified function is n bytes, which is larger than the size specified with the `-Wlarger-than-len` command-line option.

spurious trailing '%' in format

When checking the argument list of a call to `printf`, `scanf`, etc., the compiler found that there was a spurious trailing '%' character in the format string.

statement with no effect

A statement has no effect.

static declaration for 'identifier' follows non-static

The specified identifier was declared static after it was previously declared as non-static.

string length ' n ' is greater than the length ' n ' ISO C n compilers are required to support

The maximum string length for ISO C89 is 509. The maximum string length for ISO C99 is 4095.

'struct identifier' declared inside parameter list

The specified struct is declared inside a function parameter list. It is usually better programming practice to declare structs outside parameter lists, since they can never become complete types when defined inside parameter lists.

struct has no members

The structure is empty, it has no members.

structure defined inside parms

A union is defined inside a function parameter list.

style of line directive is a GCC extension

Use the format `#line linenum` for traditional C.

subscript has type 'char'

An array subscript has type 'char'.

suggest explicit braces to avoid ambiguous 'else'

A nested if statement has an ambiguous else clause. It is recommended that braces be used to remove the ambiguity.

suggest hiding *#directive* from traditional C with an indented #

The specified directive is not traditional C and may be 'hidden' by indenting the #. A directive is ignored unless its # is in column 1.

suggest not using *#elif* in traditional C

#elif should not be used in traditional K&R C.

suggest parentheses around assignment used as truth value

When assignments are used as truth values, they should be surrounded by parentheses, to make the intention clear to readers of the source program.

suggest parentheses around + or - inside shift

suggest parentheses around && within ||

suggest parentheses around arithmetic in operand of |

suggest parentheses around comparison in operand of |

suggest parentheses around arithmetic in operand of ^

suggest parentheses around comparison in operand of ^

suggest parentheses around + or - in operand of &

suggest parentheses around comparison in operand of &

While operator precedence is well defined in C, sometimes a reader of an expression might be required to expend a few additional microseconds in comprehending the evaluation order of operands in an expression if the reader has to rely solely upon the precedence rules, without the aid of explicit parentheses. A case in point is the use of the '+' or '-' operator inside a shift. Many readers will be spared unnecessary effort if parentheses are used to clearly express the intent of the programmer, even though the intent is unambiguous to the programmer and to the compiler.

T

'*identifier*' takes only zero or two arguments

Expecting zero or two arguments only.

the meaning of '\a' is different in traditional C

When the `-wtraditional` option is used, the escape sequence '\a' is not recognized as a meta-sequence: its value is just 'a'. In non-traditional compilation, '\a' represents the ASCII BEL character.

the meaning of '\x' is different in traditional C

When the `-wtraditional` option is used, the escape sequence '\x' is not recognized as a meta-sequence: its value is just 'x'. In non-traditional compilation, '\x' introduces a hexadecimal escape sequence.

third argument of '*identifier*' should probably be 'char **'

Expecting third argument of specified identifier to be of type 'char **'.

this function may return with or without a value

All exit paths from non-void function should return an appropriate value. The compiler detected a case where a non-void function terminates, sometimes with and sometimes without an explicit return value. Therefore, the return value might be unpredictable.

this target machine does not have delayed branches

The `-fdelayed-branch` option is not supported.

too few arguments for format

When checking the argument list of a call to `printf`, `scanf`, etc., the compiler found that the number of actual arguments was fewer than that required by the format string.

too many arguments for format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the number of actual arguments was more than that required by the format string.

traditional C ignores #‘directive’ with the # indented

Traditionally, a directive is ignored unless its # is in column 1.

traditional C rejects initialization of unions

Unions cannot be initialized in traditional C.

traditional C rejects the ‘ul’ suffix

Suffix ‘u’ is not valid in traditional C.

traditional C rejects the unary plus operator

The unary plus operator is not valid in traditional C.

trigraph ??char converted to char

Trigraphs, which are a three-character sequence, can be used to represent symbols that may be missing from the keyboard. Trigraph sequences convert as follows:

??([??)=	??<={	??>=}	??=#	??/=	??'\	??'^	??!=	??-=	??~
------	------	-------	-------	------	------	------	------	------	------	-----

trigraph ??char ignored

Trigraph sequence is being ignored. *char* can be (,), <, >, =, /, ', !, or -

type defaults to ‘int’ in declaration of ‘identifier’

In the absence of an explicit type declaration for the specified *identifier*, the compiler assumes that its type is int.

type mismatch with previous external decl

previous external decl of ‘identifier’

The type of the specified identifier does not match the previous declaration.

type mismatch with previous implicit declaration

An explicit declaration conflicts with a previous implicit declaration.

type of ‘identifier’ defaults to ‘int’

In the absence of an explicit type declaration, the compiler assumes that *identifier*’s type is int.

type qualifiers ignored on function return type

The type qualifier being used with the function return type is ignored.

U

undefining ‘defined’

‘defined’ cannot be used as a macro name and should not be undefined.

undefining ‘name’

The #undef directive was used on a previously defined macro name ‘name’.

union cannot be made transparent

The `transparent_union` attribute was applied to a union, but the specified variable does not satisfy the requirements of that attribute.

‘union identifier’ declared inside parameter list

The specified union is declared inside a function parameter list. It is usually better programming practice to declare unions outside parameter lists, since they can never become complete types when defined inside parameter lists.

union defined inside parms

A union is defined inside a function parameter list.

union has no members

The union is empty, it has no members.

unknown conversion type character '*character*' in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the conversion characters in the format string was invalid (unrecognized).

unknown conversion type character 0x*number* in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the conversion characters in the format string was invalid (unrecognized).

unknown escape sequence '*sequence*'

'sequence' is not a valid escape code. An escape code must start with a '\ ' and use one of the following characters: n, t, b, r, f, b, \, ', ", a, or ?, or it must be a numeric sequence in octal or hex. In octal, the numeric sequence must be less than 400 octal. In hex, the numeric sequence must start with an 'x' and be less than 100 hex.

unnamed struct/union that defines no instances

struct/union is empty and has no name.

unreachable code at beginning of *identifier*

There is unreachable code at beginning of the specified function.

unrecognized gcc debugging option: *char*

The 'char' is not a valid letter for the -d*letters* debugging option.

unused parameter '*identifier*'

The specified function parameter is not used in the function.

unused variable '*name*'

The specified variable was declared but not used.

use of '*' and 'flag' together in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that both the flags '*' and 'flag' appear in the format string.

use of C99 long long integer constants

Integer constants are not allowed to be declared long long in ISO C89.

use of '*length*' length modifier with '*type*' type character

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the specified length was incorrectly used with the specified *type*.

'*name*' used but never defined

The specified function was used but never defined.

'*name*' used with '*spec*' '*function*' format

'name' is not valid with the conversion specification 'spec' in the format of the specified function.

useless keyword or type name in empty declaration

An empty declaration contains a useless keyword or type name.

V

__VA_ARGS__ can only appear in the expansion of a C99 variadic macro

The predefined macro `__VA_ARGS__` should be used in the substitution part of a macro definition using ellipses.

value computed is not used

A value computed is not used.

variable ‘name’ declared ‘inline’

The keyword ‘inline’ should be used with functions only.

variable ‘%s’ might be clobbered by ‘longjmp’ or ‘vfork’

A non-volatile automatic variable might be changed by a call to `longjmp`. These warnings are possible only in optimizing compilation.

volatile register variables don’t work as you might wish

Passing a variable as an argument could transfer the variable to a different register (w0-w7) than the one specified (if not w0-w7) for argument transmission. Or the compiler may issue an instruction that is not suitable for the specified register and may need to temporarily move the value to another place. These are only issues if the specified register is modified asynchronously (i.e., though an ISR).

W

-Wformat-extra-args ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-extra-args`.

-Wformat-nonliteral ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-nonliteral`.

-Wformat-security ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-security`.

-Wformat-y2k ignored without -Wformat

`-Wformat` must be specified to use.

-Wid-clash-LEN is no longer supported

The option `-Wid-clash-LEN` is no longer supported.

-Wmissing-format-attribute ignored without -Wformat

`-Wformat` must be specified to use `-Wmissing-format-attribute`.

-Wuninitialized is not supported without -O

Optimization must be on to use the `-Wuninitialized` option.

‘identifier’ was declared ‘extern’ and later ‘static’

The specified identifier was previously declared ‘extern’ and is now being declared as static.

‘identifier’ was declared implicitly ‘extern’ and later ‘static’

The specified identifier was previously declared implicitly ‘extern’ and is now being declared as static.

‘identifier’ was previously implicitly declared to return ‘int’

There is a mismatch against the previous implicit declaration.

'*identifier*' was used with no declaration before its definition

When compiling with the `-Wmissing-declarations` command-line option, the compiler ensures that functions are declared before they are defined. In this case, a function definition was encountered without a preceding function declaration.

'*identifier*' was used with no prototype before its definition

When compiling with the `-Wmissing-prototypes` command-line option, the compiler ensures that function prototypes are specified for all functions. In this case, a function call was encountered without a preceding function prototype for the called function.

writing into constant object (arg *n*)

When checking the argument list of a call to `printf`, `scanf`, etc., the compiler found that the specified argument number *n* was a const object that the format specifier indicated should be written into.

Z

zero-length *identifier* format string

When checking the argument list of a call to `printf`, `scanf`, etc., the compiler found that the format string was empty (`""`).

Appendix D. GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

D.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

D.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgments”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

D.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

D.4 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

D.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- c) State on the Title page the name of the publisher of the Modified Version, as the publisher.
- d) Preserve all the copyright notices of the Document.
- e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- h) Include an unaltered copy of this License.
- i) Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- k) For any section Entitled "Acknowledgments" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
- l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- m) Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- n) Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

D.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgments”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

D.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

D.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

D.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgments", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

D.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

D.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Soft-

ware Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

D.12 RELICENSING

“Massive Multi-author Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multi-author Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

NOTES:

Appendix E. ASCII Character Set

TABLE E-1: ASCII CHARACTER SET

Least Significant Character	Most Significant Character								
	Hex	0	1	2	3	4	5	6	7
	0	NUL	DLE	Space	0	@	P	'	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

NOTES:

Appendix F. Deprecated Features

F.1 INTRODUCTION

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects which depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependency on deprecated features. Support for these features may be removed entirely in future versions of the language tools.

Deprecated features covered are:

- Predefined Constants
- Variables in Specified Registers
- Changing Non-Auto Variable Allocation

F.2 PREDEFINED CONSTANTS

The following preprocessing symbols are defined by the compiler.

Symbol	Defined with -ansi command-line option?
dsPIC30	No
__dsPIC30	Yes
__dsPIC30__	Yes

The ELF-specific version of the compiler defines the following preprocessing symbols.

Symbol	Defined with -ansi command-line option?
dsPIC30ELF	No
__dsPIC30ELF	Yes
__dsPIC30ELF__	Yes

The COFF-specific version of the compiler defines the following preprocessing symbols.

Symbol	Defined with -ansi command-line option?
dsPIC30COFF	No
__dsPIC30COFF	Yes
__dsPIC30COFF__	Yes

For the most current information, see **Section 19.4 “Predefined Macro Names”**.

F.3 VARIABLES IN SPECIFIED REGISTERS

The compiler allows you to put a few global variables into specified hardware registers.

Note: Using too many registers, in particular register W0, may impair the ability of the 16-bit compiler to compile. It is not recommended that registers be placed into fixed registers.

You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses. Stores into local register variables may be deleted when they appear to be unused. References to local register variables may be deleted, moved or simplified.

These local variables are sometimes convenient for use with the extended inline assembly (see **Chapter 16. "Mixing C and Assembly Code"**), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the inline assembly statement).

F.3.1 Defining Global Register Variables

You can define a global register variable like this:

```
register int *foo asm ("w8");
```

Here `w8` is the name of the register which should be used. Choose a register that is normally saved and restored by function calls (W8-W13), so that library routines will not clobber it.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted, moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them especially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e., in a source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. This problem can be avoided by recompiling `qsort` with the same global register variable definition.

If you want to recompile `qsort` or other source files that do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler command-line option `-ffixed-reg`. You need not actually add a global register declaration to their source code.

A function that can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function that is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value that belongs to its caller.

The library function `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`.

All global register variable declarations must precede all function definitions. If such a declaration appears after function definitions, the register may be used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

F.3.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("w8");
```

Here `w8` is the name of the register that should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. Using this feature may leave the compiler too few available registers to compile certain functions.

This option does not ensure that the compiler will generate code that has this variable in the register you specify at all times. You may not code an explicit reference to this register in an `asm` statement and assume it will always refer to this variable.

Assignments to local register variables may be deleted when they appear to be unused. References to local register variables may be deleted, moved or simplified.

F.4 CHANGING NON-AUTO VARIABLE ALLOCATION

Another way to locate data is by placing the variable into a user-defined section, and specifying the starting address of that section in a custom linker script. This is done as follows:

1. Modify the data declaration in the C source to specify a user-defined section.
2. Add the user-defined section to a custom linker script file to specify the starting address of the section.

For example, to locate the variable `Mabonga` at address `0x1000` in data memory, first declare the variable as follows in the C source:

```
int __attribute__((__section__(".myDataSection"))) Mabonga = 1;
```

The section attribute specifies that the variable should be placed in a section named `.myDataSection`, rather than the default `.data` section. It does not specify where the user-defined section is to be located. Again, that must be done in a custom linker script, as follows. Using the device-specific linker script as a base, add the following section definition:

```
.myDataSection 0x1000 :
{
    *(.myDataSection);
} >data
```

This specifies that the output file should contain a section named `.myDataSection` starting at location `0x1000` and containing all input sections named `.myDataSection`. Since, in this example, there is a single variable `Mabonga` in that section, then the variable will be located at address `0x1000` in data memory.

Appendix G. Built-in Functions

G.1 INTRODUCTION

This appendix lists the built-in functions that are specific to MPLAB XC16 C Compiler. Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using inline assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and do not involve function calls or library routines.

There are a number of reasons why providing built-in functions is preferable to requiring programmers to use inline assembly. They include the following:

1. Providing built-in functions for specific purposes simplifies coding.
2. Certain optimizations are disabled when inline assembly is used. This is not the case for built-in functions.
3. For machine instructions that use dedicated registers, coding inline assembly while avoiding register allocation errors can require considerable care. The built-in functions make this process simpler as you do not need to be concerned with the particular register requirements for each individual machine instruction.

Built-In Function List

__builtin_addab	__builtin_mulsb
__builtin_add	__builtin_mulsu
__builtin_btg	__builtin_mulus
__builtin_clr	__builtin_muluu
__builtin_clr_prefetch	__builtin_nop
__builtin_write_CRYOTP	__builtin_psvpage
__builtin_disable_interrupts	__builtin_psvoffset
__builtin_disi	__builtin_readsfr
__builtin_divf	__builtin_return_address
__builtin_divmodsd	__builtin_sac
__builtin_divmodud	__builtin_sacr
__builtin_divsd	__builtin_section_begin
__builtin_divud	__builtin_section_end
__builtin_dmapage	__builtin_section_size
__builtin_dmaoffset	__builtin_set_isr_state
__builtin_ed	__builtin_sftac
__builtin_edac	__builtin_subab
__builtin_edspage	__builtin_tbladdress
__builtin_edsoffset	__builtin_tblpage
__builtin_enable_interrupts	__builtin_tbloffset
__builtin_fbcl	__builtin_tblrhd
__builtin_get_isr_state	__builtin_tblrld
__builtin_lac	__builtin_tblwth
__builtin_mac	__builtin_tblwtl
__builtin_modsd	__builtin_write_NVM
__builtin_modud	__builtin_write_NVM_secure
__builtin_movsac	__builtin_write_PWMSFR
__builtin_mpy	__builtin_write_RTCWEN
__builtin_mpyyn	__builtin_write_OSCCONL
__builtin_msc	__builtin_write_OSCCONH

G.2 BUILT-IN FUNCTION DESCRIPTIONS

This section describes the programmer interface to the compiler built-in functions. Since the functions are “built in”, there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler’s namespace (they all have the prefix `__builtin_`), so they will not conflict with function or variable names in the programmer’s namespace.

`__builtin_addab`

Description:	Add accumulators A and B with the result written back to the specified accumulator. For example: <pre>volatile register int result asm("A"); volatile register int B asm("A"); result = __builtin_addab(result,B);</pre> will generate: <pre>add A</pre>
Prototype:	<pre>int __builtin_addab(int Accum_a, int Accum_b);</pre>
Argument:	<i>Accum_a</i> First accumulator to add. <i>Accum_b</i> Second accumulator to add.
Return Value:	Returns the addition result to an accumulator.
Assembler Operator/ Machine Instruction:	<code>add</code>
Error Messages	An error message will be displayed if the result is not an accumulator register.

`__builtin_add`

Description:	Add <i>value</i> to the accumulator specified by <i>result</i> with a shift specified by literal shift. For example: <pre>volatile register int result asm("A"); int value; result = __builtin_add(result,value,0);</pre> If <i>value</i> is held in <i>w0</i> , the following will be generated: <pre>add w0, #0, A</pre>
Prototype:	<pre>int __builtin_add(int Accum,int value, const int shift);</pre>
Argument:	<i>Accum</i> Accumulator to add. <i>value</i> Integer number to add to accumulator value. <i>shift</i> Amount to shift resultant accumulator value.
Return Value:	Returns the shifted addition result to an accumulator.
Assembler Operator/ Machine Instruction:	<code>add</code>
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• argument 0 is not an accumulator• the shift value is not a literal within range

__builtin_btg

Description: This function will generate a btg machine instruction.
Some examples include:

```
int i;    /* near by default */
int l __attribute__((far));

struct foo {
    int bit1:1;
} barbits;

int bar;

void some_bittoggles() {
    register int j asm("w9");
    int k;

    k = i;

    __builtin_btg(&i,1);
    __builtin_btg(&j,3);
    __builtin_btg(&k,4);
    __builtin_btg(&l,11);

    return j+k;
}
```

Note that taking the address of a variable in a register will produce warning by the compiler and cause the register to be saved onto the stack (so that its address may be taken); this form is not recommended. This caution only applies to variables explicitly placed in registers by the programmer.

Prototype: void __builtin_btg(unsigned int *, unsigned int 0xn);

Argument: * A pointer to the data item for which a bit should be toggled.
0xn A literal value in the range of 0 to 15.

Return Value: Returns a btg machine instruction.

**Assembler Operator/
Machine Instruction:** btg

Error Messages An error message will be displayed if the parameter values are not within range

__builtin_clr

Description: Clear the specified accumulator. For example:
volatile register int result asm("A");
result = __builtin_clr();
will generate:
clr A

Prototype: int __builtin_clr(void);

Argument: None

Return Value: Returns the cleared value result to an accumulator.

**Assembler Operator/
Machine Instruction:** clr

Error Messages An error message will be displayed if the result is not an accumulator register.

__builtin_clr_prefetch

Description:

Clear an accumulator and prefetch data ready for a future MAC operation.
xptr may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required.
yptr may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required.
xval and *yval* nominate the address of a C variable where the prefetched value will be stored.
xincr and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.
If *AWB* is non null, the other accumulator will be written back into the referenced variable.
For example:

```
volatile register int result asm("A");
volatile register int B asm("B");
int x_memory_buffer[256]
__attribute__((space(xmemory)));
int y_memory_buffer[256]
__attribute__((space(ymemory)));
int *xmemory;
int *ymemory;
int awb;
int xVal, yVal;

xmemory = x_memory_buffer;
ymemory = y_memory_buffer;
result = __builtin_clr(&xmemory, &xVal, 2,
                      &ymemory, &yVal, 2, &awb, B);
```

might generate:

```
clr A, [w8]+=2, w4, [w10]+=2, w5, w13
```

The compiler may need to spill w13 to ensure that it is available for the write-back. It may be recommended to users that the register be claimed for this purpose.

After this instruction:

- **result** will be cleared
- **xVal** will contain `x_memory_buffer[0]`
- **yVal** will contain `y_memory_buffer[0]`
- **xmemory** and **ymemory** will be incremented by 2, ready for the next mac operation

Prototype:

```
int __builtin_clr_prefetch(
    int **xptr, int *xval, int xincr,
    int **yptr, int *yval, int yincr, int *AWB,
    int AWB_accum);
```

Argument:

<i>xptr</i>	Integer pointer to x prefetch.
<i>xval</i>	Integer value of x prefetch.
<i>xincr</i>	Integer increment value of x prefetch.
<i>yptr</i>	Integer pointer to y prefetch.
<i>yval</i>	Integer value of y prefetch.
<i>yincr</i>	Integer increment value of y prefetch.
<i>AWB</i>	Accumulator write back location.
<i>AWB_accum</i>	Accumulator to write back.

Return Value:

Returns the cleared value result to an accumulator.

Assembler Operator/ Machine Instruction:

`clr`

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
- *AWB_accum* is not an accumulator and *AWB* is not null

__builtin_write_CRYOTP

Description:	Initiates a write to the Crypto OTP by issuing the correct unlock sequence and setting the CRYWR bit. Interrupts may need to be disabled for proper operation. This <code>builtin</code> function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	<code>void __builtin_write_CRYOTP(void);</code>
Argument:	None.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>mov #0x55, Wn</code> <code>mov Wn, _CRYKEY</code> <code>mov #0xAA, Wn</code> <code>mov Wn, _CRYKEY</code> <code>bset _CRYCON, #0</code> <code>nop</code> <code>nop</code>
Error Messages	None.

__builtin_disable_interrupts

Description:	Disable the specified interrupts.
Prototype:	<code>void __builtin_disable_interrupts(unsigned int interrupt);</code>
Argument:	<code>interrupt</code> Integer value specifying the interrupt to disable.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>disable_interrupts</code>
Error Messages	None.

__builtin_disi

Description:	Disables all interrupts for a specified number of instruction cycles. See Section 14.7 “Enabling/Disabling Interrupts” . Will emit the specified DISI instruction at the point it appears in the source program: <code>disi #<disi_count></code>
Prototype:	<code>void __builtin_disi(int disi_count);</code>
Argument:	<code>disi_count</code> instruction cycle count. Must be a literal integer between 0 and 16383.
Return Value:	N/A
Assembler Operator/ Machine Instruction:	<code>disi.f</code>

__builtin_divf

Description:	Computes the quotient num / den . A math error exception occurs if den is zero. Function arguments are unsigned, as is the function result.
Prototype:	<pre>unsigned int __builtin_divf(unsigned int num, unsigned int den);</pre>
Argument:	<pre>num numerator den denominator</pre>
Return Value:	Returns the unsigned integer value of the quotient num / den .
Assembler Operator/ Machine Instruction:	<pre>div.f</pre>

__builtin_divmodsd

Description:	Issues the 16-bit architecture's native signed divide support with the same restrictions given in the " <i>dsPIC30F/33F Programmer's Reference Manual</i> " (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.
Prototype:	<pre>signed int __builtin_divmodsd(signed long dividend, signed int divisor, signed int *remainder);</pre>
Argument:	<pre>dividend number to be divided divisor number to divide by remainder pointer to remainder</pre>
Return Value:	Quotient and remainder.
Assembler Operator/ Machine Instruction:	<pre>divmodsd</pre>
Error Messages	None.

__builtin_divmodud

Description:	Issues the 16-bit architecture's native unsigned divide support with the same restrictions given in the “ <i>dsPIC30F/33F Programmer's Reference Manual</i> ” (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.						
Prototype:	<pre>unsigned int __builtin_divmodud(unsigned long dividend, unsigned int divisor, unsigned int *remainder);</pre>						
Argument:	<table><tr><td><i>dividend</i></td><td>number to be divided</td></tr><tr><td><i>divisor</i></td><td>number to divide by</td></tr><tr><td><i>remainder</i></td><td>pointer to remainder</td></tr></table>	<i>dividend</i>	number to be divided	<i>divisor</i>	number to divide by	<i>remainder</i>	pointer to remainder
<i>dividend</i>	number to be divided						
<i>divisor</i>	number to divide by						
<i>remainder</i>	pointer to remainder						
Return Value:	Quotient and remainder.						
Assembler Operator/ Machine Instruction:	divmodud						
Error Messages	None.						

__builtin_divsd

Description:	Computes the quotient <i>num</i> / <i>den</i> . A math error exception occurs if <i>den</i> is zero. Function arguments are signed, as is the function result. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions.				
Prototype:	<pre>int __builtin_divsd(const long num, const int den);</pre>				
Argument:	<table><tr><td><i>num</i></td><td>numerator</td></tr><tr><td><i>den</i></td><td>denominator</td></tr></table>	<i>num</i>	numerator	<i>den</i>	denominator
<i>num</i>	numerator				
<i>den</i>	denominator				
Return Value:	Returns the signed integer value of the quotient <i>num</i> / <i>den</i> .				
Assembler Operator/ Machine Instruction:	div.sd				

__builtin_divud

Description:	Computes the quotient <i>num</i> / <i>den</i> . A math error exception occurs if <i>den</i> is zero. Function arguments are unsigned, as is the function result. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions.				
Prototype:	<pre>unsigned int __builtin_divud(const unsigned long num, const unsigned int den);</pre>				
Argument:	<table><tr><td><i>num</i></td><td>numerator</td></tr><tr><td><i>den</i></td><td>denominator</td></tr></table>	<i>num</i>	numerator	<i>den</i>	denominator
<i>num</i>	numerator				
<i>den</i>	denominator				
Return Value:	Returns the unsigned integer value of the quotient <i>num</i> / <i>den</i> .				
Assembler Operator/ Machine Instruction:	div.ud				

__builtin_dmapage

Description:	Obtains the page number of a symbol within DMA memory. For example: <pre>unsigned int result; char buffer[256] __attribute__((space(dma))); result = __builtin_dmapage(&buffer);</pre> Might generate: <pre>mov #dmapage(buffer), w0</pre>
Prototype:	<pre>unsigned int __builtin_dmapage(const void *p);</pre>
Argument:	<i>*p</i> pointer to DMA address value
Return Value:	Returns the page number of a variable located in DMA memory.
Assembler Operator/ Machine Instruction:	dmapage
Error Messages	An error message will be displayed if the parameter is not the address of a global symbol.

__builtin_dmaoffset

Description:	Obtains the offset of a symbol within DMA memory. For example: <pre>unsigned int result; char buffer[256] __attribute__((space(dma))); result = __builtin_dmaoffset(&buffer);</pre> Might generate: <pre>mov #dmaoffset(buffer), w0</pre>
Prototype:	<pre>unsigned int __builtin_dmaoffset(const void *p);</pre>
Argument:	<i>*p</i> pointer to DMA address value
Return Value:	Returns the offset to a variable located in DMA memory.
Assembler Operator/ Machine Instruction:	dmaoffset
Error Messages	An error message will be displayed if the parameter is not the address of a global symbol.

__builtin_ed

Description:	Squares <i>sqr</i> , returning it as the result. Also prefetches data for future square operation by computing <i>**xptr - **yptr</i> and storing the result in <i>*distance</i> . <i>xincr</i> and <i>yincr</i> may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value. For example: <pre>volatile register int result asm("A"); int *xmemory, *ymemory; int distance; result = __builtin_ed(distance, &xmemory, 2, &ymemory, 2, &distance);</pre> might generate: <pre>ed w4*w4, A, [w8]+=2, [W10]+=2, w4</pre>
Prototype:	<pre>int __builtin_ed(int sqr, int **xptr, int xincr, int **yptr, int yincr, int *distance);</pre>

MPLAB® XC16 C Compiler User's Guide

__builtin_ed

Argument:	<i>sqr</i>	Integer squared value.
	<i>xptr</i>	Integer pointer to pointer to x prefetch.
	<i>xincr</i>	Integer increment value of x prefetch.
	<i>yptr</i>	Integer pointer to pointer to y prefetch.
	<i>yincr</i>	Integer increment value of y prefetch.
	<i>distance</i>	Integer pointer to distance.
Return Value:	Returns the squared result to an accumulator.	
Assembler Operator/ Machine Instruction:	ed	
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• <i>xptr</i> is null• <i>yptr</i> is null• <i>distance</i> is null	

__builtin_edac

Description:	Squares <i>sqr</i> and sums with the nominated accumulator register, returning it as the result. Also prefetches data for future square operation by computing <i>**xptr - **yptr</i> and storing the result in <i>*distance</i> . <i>xincr</i> and <i>yincr</i> may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value. For example: <pre>volatile register int result asm("A"); int *xmemory, *ymemory; int distance; result = __builtin_ed(result, distance, &xmemory, 2, &ymemory, 2, &distance);</pre> might generate: <pre>edac w4*w4, A, [w8]+=2, [W10]+=2, w4</pre>	
Prototype:	<pre>int __builtin_edac(int Accum, int sqr, int **xptr, int xincr, int **yptr, int yincr, int *distance);</pre>	
Argument:	<i>Accum</i>	Accumulator to sum.
	<i>sqr</i>	Integer squared value.
	<i>xptr</i>	Integer pointer to pointer to x prefetch.
	<i>xincr</i>	Integer increment value of x prefetch.
	<i>yptr</i>	Integer pointer to pointer to y prefetch.
	<i>yincr</i>	Integer increment value of y prefetch.
	<i>distance</i>	Integer pointer to distance.
Return Value:	Returns the squared result to specified accumulator.	
Assembler Operator/ Machine Instruction:	edac	
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• <i>Accum</i> is not an accumulator register• <i>xptr</i> is null• <i>yptr</i> is null• <i>distance</i> is null	

__builtin_edspage

Description:	Returns the eds page number of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an Extended Data Space (EDS); otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_edspage(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the eds page number of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>edspage</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_edspage()</code> is not the address of an object in extended data space”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned page = __builtin_edspage(&obj);</code></p>

__builtin_edsoffset

Description:	Returns the eds page offset of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an Extended Data Space (EDS); otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_edsoffset(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the eds page number offset of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>edsoffset</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_edsoffset()</code> is not the address of an object in extended data space”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned page = __builtin_edsoffset(&obj);</code></p>

__builtin_enable_interrupts

Description:	Enable the specified interrupts.
Prototype:	<code>void __builtin_enable_interrupts(unsigned int interrupt);</code>
Argument:	<i>interrupt</i> Integer value specifying the interrupt to enable.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>enable_interrupts</code>
Error Messages	None.

__builtin_fbcl

Description:	Finds the first bit change from left in <i>value</i> . This is useful for dynamic scaling of fixed-point data. For example: <pre>int result, value; result = __builtin_fbcl(value);</pre> might generate: <pre>fbcl w4, w5</pre>
Prototype:	<pre>int __builtin_fbcl(int value);</pre>
Argument:	<i>value</i> Integer number to check for change.
Return Value:	Returns a literal value sign extended to represent the number of bits to shift left.
Assembler Operator/ Machine Instruction:	<i>fbcl</i>
Error Messages	None.

__builtin_get_isr_state

Description:	Determine the current CPU interrupt state.
Prototype:	<pre>unsigned int __builtin_get_isr_state(void);</pre>
Argument:	None.
Return Value:	Returns an integer value specifying the current CPU interrupt state.
Assembler Operator/ Machine Instruction:	<i>get_isr_state</i>
Error Messages	None.

__builtin_lac

Description:	Shifts <i>value</i> by <i>shift</i> (a literal between -8 and 7) and returns the value to be stored into the accumulator register. For example: <pre>volatile register int result asm("A"); int value; result = __builtin_lac(value,3);</pre> Might generate: <pre>lac w4, #3, A</pre>
Prototype:	<pre>int __builtin_lac(int value, int shift);</pre>
Argument:	<i>value</i> Integer number to be shifted. <i>shift</i> Literal amount to shift.
Return Value:	Returns the shifted addition result to an accumulator.
Assembler Operator/ Machine Instruction:	<i>lac</i>
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• the shift value is not a literal within range

__builtin_mac

Description:	<p>Computes $a \times b$ and sums with accumulator; also prefetches data ready for a future MAC operation.</p> <p><i>xptr</i> may be null to signify no X prefetch to be performed, in which case the values of <i>xincr</i> and <i>xval</i> are ignored, but required.</p> <p><i>yptr</i> may be null to signify no Y prefetch to be performed, in which case the values of <i>yincr</i> and <i>yval</i> are ignored, but required.</p> <p><i>xval</i> and <i>yval</i> nominate the address of a C variable where the prefetched value will be stored.</p> <p><i>xincr</i> and <i>yincr</i> may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.</p> <p>If <i>AWB</i> is non null, the other accumulator will be written back into the referenced variable.</p> <p>For example:</p> <pre>volatile register int result asm("A"); volatile register int B asm("B"); int *xmemory; int *ymemory; int xVal, yVal; result = __builtin_mac(result, xVal, yVal, &xmemory, &xVal, 2, &ymemory, &yVal, 2, 0, B);</pre> <p>might generate:</p> <pre>mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5</pre>																						
Prototype:	<pre>int __builtin_mac(int Accum, int a, int b, int **xptr, int *xval, int xincr, int **yptr, int *yval, int yincr, int *AWB, int AWB_accum);</pre>																						
Argument:	<table> <tr> <td><i>Accum</i></td><td>Accumulator to sum.</td></tr> <tr> <td><i>a</i></td><td>Integer multiplicand.</td></tr> <tr> <td><i>b</i></td><td>Integer multiplier.</td></tr> <tr> <td><i>xptr</i></td><td>Integer pointer to pointer to x prefetch.</td></tr> <tr> <td><i>xval</i></td><td>Integer pointer to value of x prefetch.</td></tr> <tr> <td><i>xincr</i></td><td>Integer increment value of x prefetch.</td></tr> <tr> <td><i>yptr</i></td><td>Integer pointer to pointer to y prefetch.</td></tr> <tr> <td><i>yval</i></td><td>Integer pointer to value of y prefetch.</td></tr> <tr> <td><i>yincr</i></td><td>Integer increment value of y prefetch.</td></tr> <tr> <td><i>AWB</i></td><td>Accumulator write-back location.</td></tr> <tr> <td><i>AWB_accum</i></td><td>Accumulator to write-back.</td></tr> </table>	<i>Accum</i>	Accumulator to sum.	<i>a</i>	Integer multiplicand.	<i>b</i>	Integer multiplier.	<i>xptr</i>	Integer pointer to pointer to x prefetch.	<i>xval</i>	Integer pointer to value of x prefetch.	<i>xincr</i>	Integer increment value of x prefetch.	<i>yptr</i>	Integer pointer to pointer to y prefetch.	<i>yval</i>	Integer pointer to value of y prefetch.	<i>yincr</i>	Integer increment value of y prefetch.	<i>AWB</i>	Accumulator write-back location.	<i>AWB_accum</i>	Accumulator to write-back.
<i>Accum</i>	Accumulator to sum.																						
<i>a</i>	Integer multiplicand.																						
<i>b</i>	Integer multiplier.																						
<i>xptr</i>	Integer pointer to pointer to x prefetch.																						
<i>xval</i>	Integer pointer to value of x prefetch.																						
<i>xincr</i>	Integer increment value of x prefetch.																						
<i>yptr</i>	Integer pointer to pointer to y prefetch.																						
<i>yval</i>	Integer pointer to value of y prefetch.																						
<i>yincr</i>	Integer increment value of y prefetch.																						
<i>AWB</i>	Accumulator write-back location.																						
<i>AWB_accum</i>	Accumulator to write-back.																						
Return Value:	Returns the cleared value result to an accumulator.																						
Assembler Operator/ Machine Instruction:	<i>mac</i>																						
Error Messages	<p>An error message will be displayed if:</p> <ul style="list-style-type: none"> the result is not an accumulator register <i>Accum</i> is not an accumulator register <i>xval</i> is a null value but <i>xptr</i> is not null <i>yval</i> is a null value but <i>yptr</i> is not null <i>AWB_accum</i> is not an accumulator register and <i>AWB</i> is not null 																						

__builtin_modsd

Description:	Issues the 16-bit architecture's native signed divide support with the same restrictions given in the " <i>dsPIC30F/33F Programmer's Reference Manual</i> " (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture only the remainder.
Prototype:	<pre>signed int __builtin_modsd(signed long dividend, signed int divisor);</pre>
Argument:	<pre>dividend</pre> number to be divided <pre>divisor</pre> number to divide by
Return Value:	Remainder.
Assembler Operator/ Machine Instruction:	<code>modsd</code>
Error Messages	None.

__builtin_modud

Description:	Issues the 16-bit architecture's native unsigned divide support with the same restrictions given in the " <i>dsPIC30F/33F Programmer's Reference Manual</i> " (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture only the remainder.
Prototype:	<pre>unsigned int __builtin_modud(unsigned long dividend, unsigned int divisor);</pre>
Argument:	<pre>dividend</pre> number to be divided <pre>divisor</pre> number to divide by
Return Value:	Remainder.
Assembler Operator/ Machine Instruction:	<code>modud</code>
Error Messages	None.

__builtin_movsac

Description:

Computes nothing, but prefetches data ready for a future MAC operation. *xptr* may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required. *yptr* may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required. *xval* and *yval* nominate the address of a C variable where the prefetched value will be stored. *xincr* and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value. If *AWB* is non null, the other accumulator will be written back into the referenced variable. For example:

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_movsac(&xmemory, &xVal, 2,
                        &ymemory, &yVal, 2, 0, 0);
```

might generate:

```
movsac A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```
int __builtin_movsac(
    int **xptr, int *xval, int xincr,
    int **yptr, int *yval, int yincr, int *AWB,
    int AWB_accum);
```

Argument:

<i>xptr</i>	Integer pointer to pointer to x prefetch.
<i>xval</i>	Integer pointer to value of x prefetch.
<i>xincr</i>	Integer increment value of x prefetch.
<i>yptr</i>	Integer pointer to pointer to y prefetch.
<i>yval</i>	Integer pointer to value of y prefetch.
<i>yincr</i>	Integer increment value of y prefetch.
<i>AWB</i>	Accumulator write back location.
<i>AWB_accum</i>	Accumulator to write back.

Return Value:

Returns prefetch data.

Assembler Operator/ Machine Instruction:

movsac

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
- *AWB_accum* is not an accumulator register and *AWB* is not null

__builtin_mpy

Description:	<p>Computes $a \times b$; also prefetches data ready for a future MAC operation.</p> <p><i>xptr</i> may be null to signify no X prefetch to be performed, in which case the values of <i>xincr</i> and <i>xval</i> are ignored, but required.</p> <p><i>yptr</i> may be null to signify no Y prefetch to be performed, in which case the values of <i>yincr</i> and <i>yval</i> are ignored, but required.</p> <p><i>xval</i> and <i>yval</i> nominate the address of a C variable where the prefetched value will be stored.</p> <p><i>xincr</i> and <i>yincr</i> may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.</p> <p>For example:</p> <pre>volatile register int result asm("A"); int *xmemory; int *ymemory; int xVal, yVal; result = __builtin_mpy(xVal, yVal, &xmemory, &xVal, 2, &ymemory, &yVal, 2);</pre> <p>might generate:</p> <pre>mpy w4*w5, A, [w8]+=2, w4, [w10]+=2, w5</pre>																		
Prototype:	<pre>int __builtin_mpy(int a, int b, int **xptr, int *xval, int xincr, int **yptr, int *yval, int yincr);</pre>																		
Argument:	<table><tr><td><i>a</i></td><td>Integer multiplicand.</td></tr><tr><td><i>b</i></td><td>Integer multiplier.</td></tr><tr><td><i>xptr</i></td><td>Integer pointer to pointer to x prefetch.</td></tr><tr><td><i>xval</i></td><td>Integer pointer to value of x prefetch.</td></tr><tr><td><i>xincr</i></td><td>Integer increment value of x prefetch.</td></tr><tr><td><i>yptr</i></td><td>Integer pointer to pointer to y prefetch.</td></tr><tr><td><i>yval</i></td><td>Integer pointer to value of y prefetch.</td></tr><tr><td><i>yincr</i></td><td>Integer increment value of y prefetch.</td></tr><tr><td><i>AWB</i></td><td>Integer pointer to accumulator selection.</td></tr></table>	<i>a</i>	Integer multiplicand.	<i>b</i>	Integer multiplier.	<i>xptr</i>	Integer pointer to pointer to x prefetch.	<i>xval</i>	Integer pointer to value of x prefetch.	<i>xincr</i>	Integer increment value of x prefetch.	<i>yptr</i>	Integer pointer to pointer to y prefetch.	<i>yval</i>	Integer pointer to value of y prefetch.	<i>yincr</i>	Integer increment value of y prefetch.	<i>AWB</i>	Integer pointer to accumulator selection.
<i>a</i>	Integer multiplicand.																		
<i>b</i>	Integer multiplier.																		
<i>xptr</i>	Integer pointer to pointer to x prefetch.																		
<i>xval</i>	Integer pointer to value of x prefetch.																		
<i>xincr</i>	Integer increment value of x prefetch.																		
<i>yptr</i>	Integer pointer to pointer to y prefetch.																		
<i>yval</i>	Integer pointer to value of y prefetch.																		
<i>yincr</i>	Integer increment value of y prefetch.																		
<i>AWB</i>	Integer pointer to accumulator selection.																		
Return Value:	Returns the value of $a \times b$.																		
Assembler Operator/ Machine Instruction:	<i>mpy</i>																		
Error Messages	<p>An error message will be displayed if:</p> <ul style="list-style-type: none">• the result is not an accumulator register• <i>xval</i> is a null value but <i>xptr</i> is not null• <i>yval</i> is a null value but <i>yptr</i> is not null																		

__builtin_mpy

Description:

Computes $-a \times b$; also prefetches data ready for a future MAC operation.
xptr may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required.
yptr may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required.
xval and *yval* nominate the address of a C variable where the prefetched value will be stored.
xincr and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.
 For example:

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                      &xmemory, &xVal, 2,
                      &ymemory, &yVal, 2);
```

might generate:

```
mpy.n w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```
int __builtin_mpy(int a, int b,
                 int **xptr, int *xval, int xincr,
                 int **yptr, int *yval, int yincr);
```

Argument:

<i>a</i>	Integer multiplicand.
<i>b</i>	Integer multiplier.
<i>xptr</i>	Integer pointer to pointer to x prefetch.
<i>xval</i>	Integer pointer to value of x prefetch.
<i>xincr</i>	Integer increment value of x prefetch.
<i>yptr</i>	Integer pointer to pointer to y prefetch.
<i>yval</i>	Integer pointer to value of y prefetch.
<i>yincr</i>	Integer increment value of y prefetch.
<i>AWB</i>	Integer pointer to accumulator selection.

Return Value:

Returns the value of $-a \times b$.

Assembler Operator/ Machine Instruction:

mpyn

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null

__builtin_msc

Description: Computes $a \times b$ and subtracts from accumulator; also prefetches data ready for a future MAC operation.

xptr may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required.

yptr may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required.

xval and *yval* nominate the address of a C variable where the prefetched value will be stored.

xincr and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

If *AWB* is not null, the other accumulator will be written back into the referenced variable.

For example:

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_msc(result, xVal, yVal,
                      &xmemory, &xVal, 2,
                      &ymemory, &yVal, 2, 0, 0);
```

might generate:

```
msc w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```
int __builtin_msc(int Accum, int a, int b,
int **xptr, int *xval, int xincr,
int **yptr, int *yval, int yincr, int *AWB,
int AWB_accum);
```

Argument:

<i>Accum</i>	IAccumulator to sum.
<i>a</i>	Integer multiplicand.
<i>b</i>	Integer multiplier.
<i>xptr</i>	Integer pointer to pointer to x prefetch.
<i>xval</i>	Integer pointer to value of x prefetch.
<i>xincr</i>	Integer increment value of x prefetch.
<i>yptr</i>	Integer pointer to pointer to y prefetch.
<i>yval</i>	Integer pointer to value of y prefetch.
<i>yincr</i>	Integer increment value of y prefetch.
<i>AWB</i>	Accumulator write back location.
<i>AWB_accum</i>	Accumulator to write back.

Return Value: Returns the value of accumulator minus the result of $a \times b$.

**Assembler Operator/
Machine Instruction:** msc

Error Messages An error message will be displayed if:

- the result is not an accumulator register
- *Accum* is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
- *AWB_accum* is not an accumulator register and *AWB* is not null

__builtin_mulss

Description:	Computes the product $p0 \times p1$. Function arguments are signed integers, and the function result is a signed long integer. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions. For example: <pre>volatile register int a asm("A"); signed long result; const signed int p0, p1; const unsigned int p2, p3; result = __builtin_mulss(p0,p1); a = __builtin_mulss(p0,p1);</pre>
Prototype:	<code>signed long __builtin_mulss(const signed int p0, const signed int p1);</code>
Argument:	<code>p0</code> multiplicand <code>p1</code> multiplier
Return Value:	Returns the signed long integer value of the product $p0 \times p1$. The value can either be returned into a variable of type signed long or directly into an accumulator register.
Assembler Operator/ Machine Instruction:	<code>mul.ss</code>

__builtin_mulsu

Description:	Computes the product $p0 \times p1$. Function arguments are integers with mixed signs, and the function result is a signed long integer. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction, including immediate mode for operand <code>p1</code> . For example: <pre>volatile register int a asm("A"); signed long result; const signed int p0, p1; const unsigned int p2, p3; result = __builtin_mulsu(p0,p2); a = __builtin_mulsu(p0,p2);</pre>
Prototype:	<code>signed long __builtin_mulsu(const signed int p0, const unsigned int p1);</code>
Argument:	<code>p0</code> multiplicand <code>p1</code> multiplier
Return Value:	Returns the signed long integer value of the product $p0 \times p1$. The value can either be returned into a variable of type signed long or directly into an accumulator register.
Assembler Operator/ Machine Instruction:	<code>mul.su</code>

__builtin_mulus

Description:	Computes the product $p0 \times p1$. Function arguments are integers with mixed signs, and the function result is a signed long integer. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction. For example: <pre>volatile register int a asm("A"); signed long result; const signed int p0, p1; const unsigned int p2, p3; result = __builtin_mulus(p2,p0); a = __builtin_mulus(p2,p0);</pre>
Prototype:	<code>signed long __builtin_mulus(const unsigned int p0, const signed int p1);</code>
Argument:	<code>p0</code> multiplicand <code>p1</code> multiplier
Return Value:	Returns the signed long integer value of the product $p0 \times p1$. The value can either be returned into a variable of type signed long or directly into an accumulator register.
Assembler Operator/ Machine Instruction:	<code>mul.us</code>

__builtin_muluu

Description:	Computes the product $p0 \times p1$. Function arguments are unsigned integers, and the function result is an unsigned long integer. The command-line option <code>-Wconversions</code> can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction, including immediate mode for operand <code>p1</code> . For example: <pre>volatile register int a asm("A"); unsigned long result; const signed int p0, p1; const unsigned int p2, p3; result = __builtin_muluu(p2,p3); a = __builtin_muluu(p2,p3);</pre>
Prototype:	<code>unsigned long __builtin_muluu(const unsigned int p0, const unsigned int p1);</code>
Argument:	<code>p0</code> multiplicand <code>p1</code> multiplier
Return Value:	Returns the signed long integer value of the product $p0 \times p1$. The value can either be returned into a variable of type unsigned long or directly into an accumulator register.
Assembler Operator/ Machine Instruction:	<code>mul.uu</code>

__builtin_nop

Description:	Generates a <code>nop</code> instruction.
Prototype:	<code>void __builtin_nop(void);</code>
Argument:	None.
Return Value:	Returns a no operation (<code>nop</code>).
Assembler Operator/ Machine Instruction:	<code>nop</code>

__builtin_psvpage

Description:	Returns the psv page number of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_psvpage(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the psv page number of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>psvpage</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_psvpage()</code> is not the address of an object in code, psv, or eed-ata section”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned page = __builtin_psvpage(&obj);</code></p>

__builtin_psvoffset

Description:	Returns the psv page offset of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_psvoffset(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the psv page number offset of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>psvoffset</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_psvoffset()</code> is not the address of an object in code, psv, or eedata section”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned page = __builtin_psvoffset(&obj);</code></p>

__builtin_readsfr

Description:	Reads the SFR.
Prototype:	<code>unsigned int __builtin_readsfr(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the SFR.
Assembler Operator/ Machine Instruction:	<code>readsfr</code>
Error Messages	The following error message is produced when this function is used incorrectly:

__builtin_return_address

Description:	Returns the return address of the current function, or of one of its callers. For the <i>level</i> argument, a value of 0 yields the return address of the current function, a value of 1 yields the return address of the caller of the current function, and so forth. When level exceeds the current stack depth, 0 will be returned. This function should only be used with a non-zero argument for debugging purposes.
Prototype:	<code>int __builtin_return_address (const int level);</code>
Argument:	<i>level</i> Number of frames to scan up the call stack.
Return Value:	Returns the return address of the current function, or of one of its callers.
Assembler Operator/ Machine Instruction:	<code>return_address</code>

__builtin_sac

Description:	Shifts value by <i>shift</i> (a literal between -8 and 7) and returns the value. For example: <pre>volatile register int value asm("A"); int result; result = __builtin_sac(value,3);</pre> Might generate: <pre>sac A, #3, w0</pre>
Prototype:	<code>int __builtin_sac(int value, int shift);</code>
Argument:	<i>value</i> Integer number to be shifted. <i>shift</i> Literal amount to shift.
Return Value:	Returns the shifted result to an accumulator.
Assembler Operator/ Machine Instruction:	<code>sac</code>
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• the shift value is not a literal within range

__builtin_sacr

Description:	Shifts value by <i>shift</i> (a literal between -8 and 7) and returns the value which is rounded using the rounding mode determined by the CORCONbits.RND control bit. For example: <pre>volatile register int value asm("A"); int result; result = __builtin_sacr(value,3);</pre> Might generate: <pre>sac.r A, #3, w0</pre>
Prototype:	<code>int __builtin_sacr(int value, int shift);</code>
Argument:	<i>value</i> Integer number to be shifted. <i>shift</i> Literal amount to shift.
Return Value:	Returns the shifted result to CORCON register.
Assembler Operator/ Machine Instruction:	<code>sacr</code>
Error Messages	An error message will be displayed if: <ul style="list-style-type: none">• the result is not an accumulator register• the shift value is not a literal within range

__builtin_section_begin

Description:	Get run-time information about a section beginning address.
Prototype:	<code>unsigned long __builtin_section_begin("section_name");</code>
Argument:	<code>section_name</code> name of the section
Return Value:	Returns the beginning address of the named section.
Assembler Operator/ Machine Instruction:	<code>section_begin</code>
Error Messages	An error message will be displayed if the named section cannot be found.

__builtin_section_end

Description:	Get run-time information about a section ending address.
Prototype:	<code>unsigned long __builtin_section_end("section_name");</code>
Argument:	<code>section_name</code> name of the section
Return Value:	Returns the ending address of the named section.
Assembler Operator/ Machine Instruction:	<code>section_end</code>
Error Messages	An error message will be displayed if the named section cannot be found.

__builtin_section_size

Description:	Get run-time information about a section's size.
Prototype:	<code>unsigned long __builtin_section_size("section_name");</code>
Argument:	<code>section_name</code> name of the section
Return Value:	Returns the size of the named section.
Assembler Operator/ Machine Instruction:	<code>section_size</code>
Error Messages	An error message will be displayed if the named section cannot be found.

__builtin_set_isr_state

Description:	Set the current CPU interrupt state.
Prototype:	<code>void __builtin_get_isr_state(unsigned int state);</code>
Argument:	<code>state</code> Integer value specifying the current CPU interrupt state.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>set_isr_state</code>
Error Messages	None.

__builtin_sftac

Description: Shifts accumulator by *shift*. The valid shift range is -16 to 16.

For example:

```
volatile register int result asm("A");
int i;
```

```
result = __builtin_sftac(result,i);
```

Might generate:

```
sftac A, w0
```

Prototype: `int __builtin_sftac(int Accum, int shift);`

Argument: *Accum* *Accumulator to shift.*

shift Amount to shift.

Return Value: Returns the shifted result to an accumulator.

**Assembler Operator/
Machine Instruction:** `sftac`

Error Messages An error message will be displayed if:

- the result is not an accumulator register
- *Accum* is not an accumulator register
- the shift value is not a literal within range

__builtin_subab

Description: Subtracts accumulators A and B with the result written back to the specified accumulator.

For example:

```
volatile register int result asm("A");
volatile register int B asm("B");
result = __builtin_subab(result,B);
```

will generate:

```
sub A
```

Prototype: `int __builtin_subab(int Accum_a, int Accum_b);`

Argument: *Accum_a* *Accumulator from which to subtract.*

Accum_b Accumulator to subtract.

Return Value: Returns the subtraction result to an accumulator.

**Assembler Operator/
Machine Instruction:** `sub`

Error Messages An error message will be displayed if the result is not an accumulator register.

__builtin_tbladdress

Description:	Returns a value that represents the address of an object in program memory. The argument <i>p</i> must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned long __builtin_tbladdress(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns an unsigned long value that represents the address of an object in program memory.
Assembler Operator/ Machine Instruction:	<code>tbladdress</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_tbladdress()</code> is not the address of an object in code, psv, or eedata section”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned long page = __builtin_tbladdress(&obj);</code></p>

__builtin_tblpage

Description:	Returns the table page number of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_tblpage(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the table page number of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>tblpage</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly: “Argument to <code>__builtin_tblpage()</code> is not the address of an object in code, psv, or eedata section”.</p> <p>The argument must be an explicit object address. For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid: <code>unsigned page = __builtin_tblpage(&obj);</code></p>

__builtin_tbloffset

Description:	Returns the table page offset of the object whose address is given as a parameter. The argument <i>p</i> must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the <i>space</i> attribute in Section 2.3.1 “Specifying Attributes of Variables” .
Prototype:	<code>unsigned int __builtin_tbloffset(const void *p);</code>
Argument:	<i>p</i> object address
Return Value:	Returns the table page number offset of the object whose address is given as a parameter.
Assembler Operator/ Machine Instruction:	<code>tbloffset</code>
Error Messages	<p>The following error message is produced when this function is used incorrectly:</p> <p>“Argument to <code>__builtin_tbloffset()</code> is not the address of an object in code, psv, or eedata section”.</p> <p>The argument must be an explicit object address.</p> <p>For example, if <i>obj</i> is object in an executable or read-only section, the following syntax is valid:</p> <pre>unsigned page = __builtin_tbloffset(&obj);</pre>

__builtin_tblrhdh

Description:	Issues the <code>tblrdh.w</code> instruction to read a word from Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of <code>__builtin_tbloffset()</code> and <code>__builtin_tblpage()</code> . Please refer to the data sheet or <i>dsPIC Family Reference Manual</i> for complete details regarding reading and writing program Flash.
Prototype:	<code>unsigned int __builtin_tblrhdh(unsigned int offset);</code>
Argument:	<i>offset</i> desired memory offset
Return Value:	Contents of the memory address in Flash or EEDATA memory.
Assembler Operator/ Machine Instruction:	<code>tblrdh</code>
Error Messages	None.

__builtin_tblrldl

Description:	Issues the <code>tblrdl.w</code> instruction to read a word from Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of <code>__builtin_tbloffset()</code> and <code>__builtin_tblpage()</code> . Please refer to the data sheet or “ <i>dsPIC30F Family Reference Manual</i> ” (DS70046) for complete details regarding reading and writing program Flash.
Prototype:	<code>unsigned int __builtin_tblrldl(unsigned int offset);</code>
Argument:	<i>offset</i> desired memory offset
Return Value:	Contents of the memory address in Flash or EEDATA memory.
Assembler Operator/ Machine Instruction:	<code>tblrdl</code>
Error Messages	None.

__builtin_tblwth

Description:	Issues the <code>tblwth.w</code> instruction to write a word to Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of <code>__builtin_tbloffset()</code> and <code>__builtin_tblpage()</code> . Please refer to the data sheet or “ <i>dsPIC30F Family Reference Manual</i> ” (DS70046) for complete details regarding reading and writing program Flash.
Prototype:	<pre>void __builtin_tblwth(unsigned int offset unsigned int data);</pre>
Argument:	<pre>offset desired memory offset data data to be written</pre>
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>tblwth</code>
Error Messages	None.

__builtin_tblwtl

Description:	Issues the <code>tblrdl.w</code> instruction to write a word to Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of <code>__builtin_tbloffset()</code> and <code>__builtin_tblpage()</code> . Please refer to the data sheet or “ <i>dsPIC30F Family Reference Manual</i> ” (DS70046) for complete details regarding reading and writing program Flash.
Prototype:	<pre>void __builtin_tblwtl(unsigned int offset unsigned int data);</pre>
Argument:	<pre>offset desired memory offset data data to be written</pre>
Return Value:	None.
Assembler Operator/ Machine Instruction:	<code>tblwtl</code>
Error Messages	None.

__builtin_write_NVM

Description:	Enables the Flash for writing by issuing the correct unlock sequence and enabling the Write bit of the NVMCON register. Interrupts may need to be disable for proper operation. This <code>builtin</code> function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	<pre>void __builtin_write_NVM(void);</pre>
Argument:	None.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<pre>mov #0x55, Wn mov Wn, _NVMKEY mov #0xAA, Wn mov Wn, _NVMKEY bset _NVMCON, #15 nop nop</pre>
Error Messages	None.

__builtin_write_NVM_secure

Description:	Enables the Flash for writing by issuing an unlock sequence specified by two keys and enabling the Write bit of the NVMCON register. Interrupts may need to be disabled for proper operation. This <code>builtin</code> function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	<pre>void __builtin_write_NVM_secure(unsigned int key1, unsigned int key2);</pre>
Argument:	<i>key1</i> first key in the NVM unlock sequence <i>key2</i> second key in the NVM unlock sequence
Return Value:	None.
Assembler Operator/ Machine Instruction:	Depending on the location of the keys: <pre>mov W0, Wn mov Wn, _NVMKEY mov W1, Wn mov Wn, _NVMKEY bset _NVMCON, #15 nop nop</pre>
Error Messages	None.

__builtin_write_PWMSFR

Description:	Writes the PWM unlock sequence to the SFR pointed to by <i>PWM_KEY</i> and then writes <i>value</i> to the SFR pointed to by <i>PWM_sfr</i> .
Prototype:	<pre>void __builtin_write_PWMSFR(volatile unsigned int *PWM_sfr, unsigned int value, volatile unsigned int *PWM_KEY);</pre>
Argument:	<i>PWM_sfr</i> register to be written <i>value</i> value to write <i>PWM_KEY</i> hardware unlock key location
Return Value:	None.
Assembler Operator/ Machine Instruction:	<pre>mov #<it>PWM_KEY</it>, w3 mov #<it>value</it>, w2 mov #0x4321, w1 mov #0xABCD, w0 mov w1, [w3] mov w0, [w3] mov w2, [w3]</pre>
Error Messages	None.
Examples	<p>Example 1:</p> <pre>__builtin_write_PWMSFR(&PWM1CON1, 0x123, &PWM1KEY);</pre> <p>Example 2:</p> <pre>__builtin_write_PWMSFR(&P1FLTACON, 0x123, &PWMKEY);</pre> <p>The choice of <i>PWM_KEY</i> may depend upon architecture.</p>

__builtin_write_RTCWEN

Description:	Used to write to the RTCC Timer by implementing the unlock sequence by writing the correct unlock values to <code>NVMKEY</code> and then setting the <code>RTCWREN</code> bit of <code>RCFGCAL</code> SFR. Interrupts may need to be disable for proper operation. This <code>builtin</code> function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	<code>void __builtin_write_RTCWEN(void);</code>
Argument:	None.
Return Value:	None.
Assembler Operator/ Machine Instruction:	<pre>mov #0x55, Wn mov Wn, _NVMKEY mov #0xAA, Wn mov Wn, _NVMKEY bset _RCFGCAL, #13 nop nop</pre>
Error Messages	None.

__builtin_write_OSCCONL

Description:	Unlocks and writes its argument to <code>OSCCONL</code> . Interrupts may need to be disable for proper operation. This <code>builtin</code> function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	<code>void __builtin_write_OSCCONL(unsigned char value);</code>
Argument:	<code>value</code> character to be written
Return Value:	None.
Assembler Operator/ Machine Instruction*:	<pre>mov #0x46, w0 mov #0x57, w1 mov #_OSCCON, w2 mov.b w0, [w2] mov.b w1, [w2] mov.b value, [w2]</pre>
Error Messages	None.

* The exact sequence may be different.

__builtin_write_OSCCONH

Description: Unlocks and writes its argument to OSCCONH.
Interrupts may need to be disabled for proper operation.
This `builtin` function can be used as a part of a complex sequence discussed in the data sheet or family reference manual.
See this documentation for more information.

Prototype: `void __builtin_write_OSCCONH(unsigned char value);`

Argument: `value` character to be written

Return Value: None.

Assembler Operator/ `mov #0x78, w0`
Machine Instruction*: `mov #0x9A, w1`
`mov #_OSCCON+1, w2`
`mov.b w0, [w2]`
`mov.b w1, [w2]`
`mov.b value, [w2]`

Error Messages None.

* The exact sequence may be different.

Appendix H. Document Revision History

Revision A (April 2012)

Initial release of this document.

Revision B (July 2012)

- **Chapter 2. "Common C Interface"** was added.
- Figure "Software Development Tools Data Flow" was updated.
- Table 5-16 "Linking Options" now includes the -fill option.
- Added the -pack_upper_byte qualifier information in **Section 8.11.4 "__pack_upper_byte Type Qualifier"** and **Section 10.8 "Packing Data Stored in Flash"**.
- Added DBRPAG/PSVPAG preservation bullet under **Section 13.8 "Function Call Conventions"**
- Fixed code syntax in **Section 14.4 "Specifying the Interrupt Vector"**.
- Fixed Eval Edition description under **Chapter 18. "Optimizations"**.
- Added "volatile" to SFR registers in **Appendix G. "Built-in Functions"**.
- Added built-in functions __builtin_write_CRYOTP and __builtin_write_NVM_secure in **Appendix G. "Built-in Functions"**.

Revision C (Sept 2013)

- Renamed MPLAB Assembler/Linker for PIC24 MCUs and dsPIC DSCs (and variants) to MPLAB XC16 Assembler/Linker.
- Changed executable output from .out to .elf.
- Updated MDB information in **Section 1.4 "Compiler and Other Development Tools"**.
- Added **Chapter 4. "XC16 Toolchain and MPLAB X IDE"** and **Chapter 4. "XC16 Toolchain and MPLAB IDE v8"**.
- Added options under **Section 5.7 "Driver Option Descriptions"**:
-menable-fixed and -fsigned-bitfields.
- Added information on using #pragmas under **Section 6.5 "Configuration Bit Access"**.
- Added fixed-point arithmetic support:
 - **Chapter 9. "Fixed-Point Arithmetic Support"**.
 - **Section 8.4 "Floating-Point Data Types"**
 - **Section 12.2 "Register Variables"** (_Sat, _Fract, _Accum)
 - **Section 13.2.2 "Function Attributes"** (round)
 - **Section 13.8 "Function Call Conventions"** (_Fract, _Accum)
- Bitfield updates under **Section 8.6.2 "Bit-fields in Structures"**.
- Added the following attributes to **Section 13.2.2 "Function Attributes"**: naked, keep.
- Added ISR section naming under **Section 14.3 "Writing an Interrupt Service Routine"**. Also, Interrupt Vector information has been removed from this manual and moved to the docs subdirectory of the compiler installation directory, as per **Section 14.4 "Specifying the Interrupt Vector"**.

- Optimization details have been added to **Chapter 18. “Optimizations”**.
- Updates to **Section 19.4.2 “Output Types and Device Macros”**.
- Additions concerning bit-fields in **Section A.10 “Structures, Unions, Enumerations and Bit-Fields”** and `#pragma config` in **Section A.14 “Preprocessing Directives”**.
- Added built-in functions below to **Appendix G. “Built-in Functions”**:
 - `__builtin_disable_interrupts`
 - `__builtin_enable_interrupts`
 - `__builtin_get_isr_state`
 - `__builtin_set_isr_state`
 - `__builtin_section_begin`
 - `__builtin_section_end`
 - `__builtin_section_size`
- Added **Appendix B. “Embedded Compiler Compatibility Mode”**.

Revision D (August 2014)

- Added **Chapter 3. “How To’s”**.
- Removed Chapter 4. XC16 Toolchain and MPLAB IDE v8.



MPLAB[®] XC16 C COMPILER USER'S GUIDE

Support

INTRODUCTION

Please refer to the items discussed here for support issues.

- myMicrochip Personalized Notification Service
- The Microchip Web Site
- Microchip Forums
- Customer Support
- Contact Microchip Technology

myMICROCHIP PERSONALIZED NOTIFICATION SERVICE

myMicrochip: <http://www.microchip.com/pcn>

Microchip's personal notification service helps keep customers current on their Microchip products of interest. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool.

Please visit myMicrochip to begin the registration process and select your preferences to receive personalized notifications. A FAQ and registration details are available on the page, which can be opened by selecting the link above.

When you are selecting your preferences, choosing "Development Systems" will populate the list with available development tools. The main categories of tools are listed below:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM[™] assembler); all MPLAB linkers (including MPLINK[™] object linker); and all MPLAB librarians (including MPLIB[™] object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB REAL ICE[™] in-circuit emulator.
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the PICKit[™] 2, PICKit 3 and MPLAB ICD 3 in-circuit debuggers.
- **MPLAB[®] IDE** – The latest information on Microchip MPLAB IDE, the Windows[®] Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.

MPLAB® XC16 C Compiler User's Guide

- **Programmers** – The latest information on Microchip programmers. These include the device (production) programmers MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger, MPLAB PM3 and development (nonproduction) programmers PICKit 2 and 3.
- **Starter/Demo Boards** – These include MPLAB Starter Kit boards, PICDEM demo boards, and various other evaluation boards.

THE MICROCHIP WEB SITE

Web Site: <http://www.microchip.com>

Microchip provides online support via our web site. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

MICROCHIP FORUMS

Forums: <http://www.microchip.com/forums>

Microchip provides additional online support via our web forums. Currently available forums are:

- Development Tools
- 8-bit PIC MCUs
- 16-bit PIC MCUs
- 32-bit PIC MCUs

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document. See our web site for a complete, up-to-date listing of sales offices.

Technical Support: <http://support.microchip.com>

Documentation errors or comments may be emailed to docerrors@microchip.com.

CONTACT MICROCHIP TECHNOLOGY

You may call or fax Microchip Corporate offices at the numbers below:

Voice: (480) 792-7200

Fax: (480) 792-7277

NOTES:



Glossary

A

Absolute Section

A GCC compiler section with a fixed (absolute) address that cannot be changed by the linker.

Absolute Variable/Function

A variable or function placed at an absolute address using the OCG compiler's @ *address* syntax.

Access Memory

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

Access Entry Points

Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. They support the separate linking of boot and secure application segments.

Address

Value that identifies a location in memory.

Alphabetic Character

Alphabetic characters are those characters that are letters of the arabic alphabet (a, b, ..., z, A, B, ..., Z).

Alphanumeric

Alphanumeric characters are comprised of alphabetic characters and decimal digits (0,1, ..., 9).

ANDed Breakpoints

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

Anonymous Structure

16-bit C Compiler – An unnamed structure.

PIC18 C Compiler – An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union. For example, in the following code, *hi* and *lo* are members of an anonymous structure inside the union *caster*.

```
union castaway {
    int intval;
    struct {
        char lo; //accessible as caster.lo
        char hi; //accessible as caster.hi
    };
} caster;
```

ANSI

American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

Application

A set of software and hardware that may be controlled by a PIC® microcontroller.

Archive/Archiver

An archive/library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver/librarian to combine the object files into one archive/library file. An archive/library can be linked with object modules and other archives/libraries to create executable code.

ASCII

American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembly/Assembler

Assembly is a programming language that describes binary machine code in a symbolic form. An assembler is a language tool that translates assembly language source code into machine code.

Assigned Section

A GCC compiler section which has been assigned to a target memory block in the linker command file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

Asynchronous Stimulus

Data generated to simulate external inputs to a simulator device.

Attribute

GCC Characteristics of variables or functions in a C program which are used to describe machine-specific properties.

Attribute, Section

GCC Characteristics of sections, such as “executable”, “readonly”, or “data” that can be specified as flags in the assembler `.section` directive.

B

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

Bookmarks

Use bookmarks to easily locate specific lines in a file.

Select Toggle Bookmarks on the Editor toolbar to add/remove bookmarks. Click other icons on this toolbar to move to the next or previous bookmark.

Breakpoint

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build

Compile and link all the source files for an application.

C**C/C++**

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C++ is the object-oriented version of C.

Calibration Memory

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

Clean

Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

COFF

Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface

A means of communication between a program and its user based solely on textual input and output.

Compiled Stack

A region of memory managed by the compiler in which variables are statically allocated space. It replaces a software or hardware stack when such mechanisms cannot be efficiently implemented on the target device.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Assembly

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU

See Central Processing Unit.

Cross Reference File

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives

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

Data Memory

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Data Monitor and Control Interface (DMCI)

The Data Monitor and Control Interface, or DMCI, is a tool in MPLAB X IDE. The interface provides dynamic input control of application variables in projects. Application-generated data can be viewed graphically using any of 4 dynamically-assignable graph windows.

Debug/Debugger

See ICE/ICD.

Debugging Information

Compiler and assembler options that, when selected, provide varying degrees of information used to debug application code. See compiler or assembler documentation for details on selecting debug options.

Deprecated Features

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller

A digital signal controller (DSC) is a microcontroller device with digital signal processing capability, i.e., Microchip dsPIC DSC devices.

Digital Signal Processing\Digital Signal Processor

Digital signal processing (DSP) is the computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled). A digital signal processor is a microprocessor that is designed for use in digital signal processing.

Directives

Statements in source code that provide control of the language tool's operation.

Download

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

DWARF

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

Emulation/Emulator

See ICE/ICD.

Endianness

The ordering of bytes in a multi-byte object.

Environment

MPLAB PM3 – A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

Epilogue

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Error/Error File

An error reports a problem that makes it impossible to continue processing your program. When possible, an error identifies the source file name and line number where the problem is apparent. An error file contains error messages and diagnostics generated by a language tool.

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, breakpoints and interrupts.

Executable Code

Software that is ready to be loaded for execution.

Export

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

Expressions

Combinations of constants and/or symbols separated by arithmetic or logical operators.

Extended Microcontroller Mode

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 PIC18 device.

Extended Mode (PIC18 MCUs)

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

External Label

A label that has external linkage.

External Linkage

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

External Symbol

A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution

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

External Input Line

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

External RAM

Off-chip Read/Write memory.

F

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

File Registers

On-chip data memory, including General Purpose Registers (GPRs) and Special Function Registers (SFRs).

Filter

Determine by selection what data is included/excluded in a trace display or data file.

Fixup

The process of replacing object file symbolic references with absolute addresses after relocation by the linker.

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 PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

Free-Standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI '89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

G

GPR

General Purpose Register. The portion of device data memory (RAM) available for general use.

H

Halt

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Heap

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code\Hex File

Hex code is executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then $16^2 = 256$, etc.

High Level Language

A language for writing programs that is further removed from the processor than assembly.

I

ICE/ICD

In-Circuit Emulator/In-Circuit Debugger: A hardware tool that debugs and programs a target device. An emulator has more features than a debugger, such as trace.

In-Circuit Emulation/In-Circuit Debug: The act of emulating or debugging with an in-circuit emulator or debugger.

-ICE/-ICD: A device (MCU or DSC) with on-board in-circuit emulation or debug circuitry. This device is always mounted on a header board and used to debug with an in-circuit emulator or debugger.

ICSP

In-Circuit Serial Programming. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

IDE

Integrated Development Environment, as in MPLAB IDE/MPLAB X IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

Bring data into the MPLAB IDE/MPLAB X 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.

Instruction Set

The collection of machine language instructions that a particular processor understands.

Instructions

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Internal Linkage

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

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications. Also known as ISO.

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Request (IRQ)

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine (ISR)

Language tools – A function that handles an interrupt.

MPLAB IDE/MPLAB X IDE – User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

L

L-value

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency

The time between an event and its response.

Library/Librarian

See Archive/Archiver.

Linker

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives

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

Listing File

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

Little Endian

A data ordering scheme for multibyte data whereby the least significant byte is stored at the lower addresses.

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 can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Loop-Back Test Board

Used to test the functionality of the MPLAB REAL ICE in-circuit emulator.

LVDS

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

With standard I/O signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: <http://www.webopedia.com/TERM/L/LVDS.html>.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro

Macro instruction. An instruction that represents a sequence of instructions in abbreviated form.

Macro Directives

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

Makefile

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB IDE/MPLAB X IDE, i.e., with a `make`.

Make Project

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

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also `uC`.

Memory Model

For C compilers, a representation of the memory available to the application. For the PIC18 C compiler, a description that specifies the size of pointers that point to program memory.

Message

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller

A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

Microcontroller Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

Text instructions that can be translated directly into machine code. Also referred to as opcodes.

Module

The preprocessed output of a source file after preprocessor directives have been executed. Also known as a translation unit.

MPASM™ Assembler

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices and Microchip memory devices.

MPLAB Language Tool for Device

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

MPLAB ICD

Microchip in-circuit debugger that works with MPLAB IDE/MPLAB X IDE. See ICE/ICD.

MPLAB IDE/MPLAB X IDE

Microchip's Integrated Development Environment. MPLAB IDE/MPLAB X IDE comes with an editor, project manager and simulator.

MPLAB PM3

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB IDE/MPLAB X IDE or stand-alone. Replaces PRO MATE II.

MPLAB REAL ICE™ In-Circuit Emulator

Microchip's next-generation in-circuit emulator that works with MPLAB IDE/MPLAB X IDE. See ICE/ICD.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE/MPLAB X IDE in support of PIC MCU and dsPIC DSC devices.

MPLIB™ Object Librarian

Microchip's librarian that can work with MPLAB IDE/MPLAB X IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C18 C Compiler.

MPLINK™ Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE/MPLAB X IDE, though it does not have to be.

MRU

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

N

Native Data Size

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device's data memory: bytes for PIC18 devices and words for 16-bit devices.

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB IDE/MPLAB X IDE project component.

Non-Extended Mode (PIC18 MCUs)

In Non-Extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

Non Real Time

Refers to the processor at a breakpoint or executing single-step instructions or MPLAB IDE/MPLAB X IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

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

O

Object Code/Object File

Object code is the machine code generated by an assembler or compiler. An object file is a file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives

Directives that are used only when creating an object file.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

Off-Chip Memory

Off-chip memory refers to the memory selection option for the PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. See Mnemonics.

Operators

Symbols, like the plus sign '+' and the minus sign '-', that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter

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 PC running a supported Windows operating system.

Persistent Data

Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device Reset.

Phantom Byte

An unimplemented byte in the dsPIC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC hex files.

PIC MCUs

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

PICKit 2 and 3

Microchip's developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

Plug-ins

The MPLAB IDE/MPLAB X IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools may be found under the Tools menu.

Pod

The enclosure for an in-circuit emulator or debugger. Other names are “Puck”, if the enclosure is round, and “Probe”, not be confused with logic probes.

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.

Pragma

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

A production programmer is a programming tool that has resources designed in to program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

Profile

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

Program Counter

The location that contains the address of the instruction that is currently executing.

Program Counter Unit

16-bit assembler – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory

MPLAB IDE/MPLAB X IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

16-bit assembler/compiler – The memory area in a device where instructions are stored.

Project

A project contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options.

Prologue

A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement specified in the runtime model. This code executes before any user code for a given function.

Prototype System

A term referring to a user's target application, or target board.

Psect

The OCG equivalent of a GCC section, short for program section. A block of code or data which is treated as a whole by the linker.

PWM Signals

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

Q

Qualifier

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

R

Radix

The number base, hex, or decimal, used in specifying an address.

RAM

Random Access Memory (Data Memory). Memory in which information can be accessed in any order.

Raw Data

The binary representation of code or data associated with a section.

Read Only Memory

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

Real Time

When an in-circuit emulator or debugger is released from the halt state, 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 an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursive Calls

A function that calls itself, either directly or indirectly.

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.

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion or through execution during interrupt processing.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object whose address has not been assigned to a fixed location in memory.

Relocatable Section

16-bit assembler – 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 symbols in the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory). Memory that cannot be modified.

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.

Run-time Model

Describes the use of target architecture resources.

Runtime Watch

A Watch window where the variables change in as the application is run. See individual tool documentation to determine how to set up a runtime watch. Not all tools support runtime watches.

S

Scenario

For MPLAB SIM simulator, a particular setup for stimulus control.

Section

The GCC equivalent of an OCG psect. A block of code or data which is treated as a whole by the linker.

Section Attribute

A GCC characteristic ascribed to a section (e.g., an `access` section).

Sequenced Breakpoints

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up; the last breakpoint in the sequence occurs first.

Serialized Quick Turn Programming

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

Shell

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

Simulator

A software program that models the operation of devices.

Single Step

This command steps through code, one instruction at a time. After each instruction, MPLAB IDE/MPLAB X 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/MPLAB X 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 opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code

The form in which a computer program is written by the programmer. Source code is written in a formal programming language which can be translated into machine code or executed by an interpreter.

Source File

An ASCII text file containing source code.

Special Function Registers (SFRs)

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

SQTP

See Serialized Quick Turn Programming.

Stack, Hardware

Locations in PIC microcontroller where the return address is stored when a function call is made.

Stack, Software

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is dynamically allocated at runtime by instructions in the program. It allows for reentrant function calls.

Stack, Compiled

A region of memory managed and allocated by the compiler in which variables are statically assigned space. It replaces a software stack when such mechanisms cannot be efficiently implemented on the target device. It precludes reentrancy.

MPLAB Starter Kit for Device

Microchip's starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program your own changes.

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/MPLAB X 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 breakpoint will be set at the instruction after the CALL. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of CALL instructions.

Step Out

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch

A counter for measuring execution cycles.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

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/MPLAB X IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute

Represents an immediate value such as a definition through the assembly `.equ` directive.

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.”

T

Target

Refers to user hardware.

Target Application

Software 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.

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/MPLAB X IDE functions.

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/MPLAB X IDE's trace window.

Trace Memory

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

Trace Macro

A macro that will provide trace information from emulator data. Since this is a software trace, the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

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 breakpoint settings. Any number of trigger output points can be set.

Trigraphs

Three-character sequences, all starting with ??, that are defined by ISO C as replacements for single characters.

U

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.

USB

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

V

Vector

The memory locations that an application will jump to when either a Reset or interrupt occurs.

Volatile

A variable qualifier which prevents the compiler applying optimizations that affect how the variable is accessed in memory.

W

Warning

MPLAB IDE/MPLAB X IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

16-bit assembler/compiler – Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text ‘warning:’ to distinguish them from error messages.

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 breakpoint.

Watchdog Timer (WDT)

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

Workbook

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

NOTES:

Index

IAR compatibility	251–256		
Symbols			
__align qualifier	34	__builtin_tblastress	335
__bank qualifier	33	__builtin_tblastress	336
__builtin_add	313	__builtin_tblastress	335
__builtin_addab	313	__builtin_tblastress	336
__builtin_btg	314	__builtin_tblastress	336
__builtin_clr	314	__builtin_tblastress	337
__builtin_clr_prefect	315	__builtin_tblastress	337
__builtin_disable_interrupts	316	__builtin_write_CRYOTP	316
__builtin_disi	316	__builtin_write_NVM	337
__builtin_divf	317	__builtin_write_NVM_secure	338
__builtin_divmodsd	317	__builtin_write_OSCCONH	340
__builtin_divmodud	318	__builtin_write_OSCCONL	339
__builtin_divsd	318	__builtin_write_PWMSFR	338
__builtin_divud	318	__builtin_write_RTCWEN	339
__builtin_dmaoffset	319	__deprecated qualifier	39
__builtin_dmapage	319	__eeprom qualifier	35
__builtin_ed	319	__far qualifier	30
__builtin_edac	320	__interrupt qualifier	35
__builtin_edsoffset	321	__near qualifier	31
__builtin_edspage	321	__pack qualifier	38
__builtin_enable_interrupts	321	__persisten qualifier	32
__builtin_fbcl	322	__section qualifier	39
__builtin_get_isr_state	322	__XC16_VERSION__	230
__builtin_lac	322	__xdata qualifier	33
__builtin_mac	323	__ydata qualifier	33
__builtin_modsd	324	__Accum	200
__builtin_modud	324	__delay function	61
__builtin_movsac	325	__Fract	200
__builtin_mpy	326	.bss	146, 245
__builtin_mpyn	327	.c	72
__builtin_msc	328	.const	215
__builtin_mulss	329	.data	146, 245
__builtin_mulsu	329	.gld	72
__builtin_mulus	330	.s	72
__builtin_muluu	330	.text	97, 193, 196, 245
__builtin_nop	330	#define	111
__builtin_psvoffset	331	#ident	117
__builtin_psvpage	331	#if	105
__builtin_readsfr	331	#include	112, 113
__builtin_return_address	332	#include directive	88
__builtin_sac	332	#line	114
__builtin_sacr	332	#pragma	102, 233, 245
__builtin_section_begin	333		
__builtin_section_end	333	Numerics	
__builtin_section_size	333	0b binary radix specifier	136
__builtin_set_isr_state	333	16-Bit Specific Options	95
__builtin_sftac	334		
__builtin_subab	334	A	
		-A	111
		abort	193, 249
		absolute functions	29

MPLAB® XC16 C Compiler User's Guide

absolute variables	29	Attribute, Variable	142
activation, see compiler installation & activation		address	143
address Attribute	143, 189	aligned	143
alias Attribute	189	boot	143
aligned Attribute	143	deprecated	144
Alignment	143, 146, 200, 244	eds	144
-ansi	98, 114, 197	far	144, 181, 199
ANSI C Standard	13	fillupper	144
ANSI C standard	18	mode	144
conformance	125	near	145, 181, 199
implementation-defined behavior	125	noload	145
ANSI C, Strict	99	packed	145
ANSI Standard Library Support	14	persistent	146
ANSI-89 extension	128	reverse	146
Archiver	88	section	146
arrays		secure	147
initialization	137	sfr	147
Arrays and Pointers	243	space	147
ASCII Character Set	305	transparent_union	149
ASCII characters	129	unordered	149
extended	137	unsupported	149
asm	143, 220	unused	149
asm C statement	43	weak	149
Assembler	88	auto variables	156, 159
assembly code		memory allocation	159
mixing with C	58	auto_psv Space	95
writing	58	Automatic Variable	102, 103, 160
assembly list files	93	-aux-info	98
Assembly Options	114	B	
-Wa	114	-B	87
assembly source files	87	binary constants	
Assembly, Inline	220	C code	136
Assembly, Mixing with C	217	bit-fields	26, 27, 99, 131, 212, 244
Atomic Operation	210	bitwise complement operator	184
attribute	14, 142, 188, 233	boot Attribute	143, 189
Attribute, Function		Built-In Functions	
address	189	__builtin_add	313
alias	189	__builtin_addab	313
boot	189	__builtin_btg	314
const	191	__builtin_clr	314
deprecated	191	__builtin_clr_prefect	315
far	191	__builtin_disable_interrupts	316
format	191	__builtin_disi	316
format_arg	191	__builtin_divf	317
interrupt	192, 206, 208	__builtin_divmodsd	317
keep	192	__builtin_divmodud	318
naked	192	__builtin_divsd	318
near	192	__builtin_divud	318
no_instrument_function	192	__builtin_dmaoffset	319
noload	192	__builtin_dmapage	319
noreturn	104, 193	__builtin_ed	319
round	193	__builtin_edac	320
save(list)	193	__builtin_edsoffset	321
section	193, 196, 309	__builtin_edspace	321
secure	194	__builtin_enable_interrupts	321
shadow	195, 206	__builtin_fbcl	322
unsupported	195	__builtin_get_isr_state	322
unused	195	__builtin_lac	322
user_init	195	__builtin_mac	323
weak	195	__builtin_modsd	324

builtin_modul	324	char data types	23, 129
builtin_movsac	325	character constants	
builtin_mpy	326	in C	137
builtin_mpyr	327	Characters	241
builtin_msc	328	Code Generation Conventions Options	117
builtin_mulss	329	-fargument-alias	117
builtin_mulsu	329	-fargument-noalias	117
builtin_mulus	330	-fargument-noalias-global	117
builtin_muluu	330	-fcall-saved	117
builtin_nop	330	-fcall-used	117
builtin_psvoffset	331	-ffixed	117
builtin_psvpage	331	-fno-ident	117
builtin_readsfr	331	-fno-short-double	118
builtin_return_address	332	-fno-verbose-asm	118
builtin_sac	332	-fpack-struct	118
builtin_sacr	332	-fpcc-struct-return	118
builtin_section_begin	333	-fshort-enums	118
builtin_section_end	333	-fverbose-asm	118
builtin_section_size	333	Code Size, Reduce	95, 106
builtin_set_isr_state	333	Coding ISRs	206
builtin_sftac	334	COFF	88
builtin_subab	334	Command Line Options	85
builtin_tbladdress	335	Command-Line Compiler	85
builtin_tbloffset	336	Command-Line Options	95
builtin_tblpage	335	Command-Line Simulator	14, 88
builtin_tblrhd	336	Comments	99, 111
builtin_tblrld	336	common compiler interface	19
builtin_tblwth	337	Common Subexpression Elimination	107, 108, 109, 191
builtin_tblwtl	337	compilation	
builtin_write_CRYOTP	316	incremental builds	90
builtin_write_NVM	337	Compiler	88
builtin_write_NVM_secure	338	Command-Line	85
builtin_write_OSCCONH	340	Driver	14, 88
builtin_write_OSCCONL	339	Overview	13
builtin_write_PWMSFR	338	Compiler Description	13
builtin_write_RTCWEN	339	compiler installation & activation	45
C		compiler selection	47
-C	111	compiler-generated code	65
-c	97, 115	Compiling Multiple Files	90
C Dialect Control Options	98	Complex	
-ansi	98	Data Types	135
-aux-info	98	Floating Types	135
-ffreestanding	98	Integer Types	135
-fno-asm	98	complex	135
-fno-builtin	99	const Attribute	191
-fno-signed-bitfields	99	const objects	
-fno-unsigned-bitfields	99	initialization	138
-fsigned-bitfields	99	const qualifier	138, 166
-fsigned-char	99	Constants	
-funsigned-bitfields	99	Predefined	230, 307
-funsigned-char	99	constants	
-menable-fixed	98	C specifiers	136
-traditional	197	character	137
C Stack Usage	160	string, see string literals	137
C standard libraries	92	Contact Microchip Technology	345
C, Mixing with Assembly	217	conversion between types	183
Cast	102, 103, 104	CORCON	122, 215
casting	184	Customer Support	344
CCI	19		
char	99, 128, 144, 185, 200		

MPLAB® XC16 C Compiler User's Guide

D

-D	111, 112, 114
data memory	156
Data Memory Allocation	158
Data Memory Space	95, 97, 179
Data Memory Space, Near	145
Data Type	144
Complex	135
data types	
size of	23
Data, Packed	175
-dD	111
Debugging Information	105
Debugging Options	105
-g	105
-Q	105
-save-temps	106
Declarators	244
Defining Global Register Variables	308
delay routine	61
deprecated Attribute	104, 144, 191
Development Tools	15
device support	65
Device Support Files	119
diagnostic files	93
Diagnostics	257
Directories	75, 112, 114
Directory Search Options	117
-B	87
-specs=	117
disabling interrupts	60
-dM	111
-dN	111
Documentation	13
Conventions	11
Layout	9
double	118, 129, 185, 200
driver	
input files	86
driver option	
CCI	44
EXT	252
PRE	229
driver options	48, 86
dsPIC-Specific Options	
-mauxflash	97
-mconst-in-auxflash	95
-mconst-in-code	95
-mconst-in-data	95
-mcpu	96
-merrata	95
-mfillupper	95
-mlarge-arrays	96
-mlarge-code	96
-mlarge-data	96
-mno-isr-warn	96
-mno-pa	96
-momf=	96
-mpa	96
-mpa=	96

-msmall-code	96
-msmall-data	97
-msmall-scalar	97
-msmart-io	97
-mtext=	97

DWARF	96
-------------	----

E

-E	97, 111, 113, 114, 115
eds Attribute	144
EEDATA	159
EEPROM, data	159
ELF	88, 96
Enabling/Disabling Interrupts	209
Enumerations	244
Environment	240
Environment Variables	
PIC30_C_INCLUDE_PATH	86
PIC30_COMPILER_PATH	86
PIC30_LIBRARY_PATH	87
TMPDIR	87
XC16_C_INCLUDE_PATH	86
XC16_COMPILER_PATH	86
XC16_EXEC_PREFIX	87
XC16_LIBRARY_PATH	87
errno	249
Error Control Options	
-pedantic-errors	99
-Werror	104
-Werror-implicit-function-declaration	99
error messages	67
location	67
Errors	257
Escape Sequences	241
Example	81
Exception Vectors	207
exit	249
extended character set	137
Extensions	113
extern	104, 110, 198
External Symbols	217

F

F constant suffix	137
-falign-functions	107
-falign-labels	107
-falign-loops	107
far Attribute	144, 181, 191, 199, 220
Far Data Space	181
-fargument-alias	117
-fargument-noalias	117
-fargument-noalias-global	117
-fcaller-saves	107
-fcall-saved	117
-fcall-used	117
-fcse-follow-jumps	107
-fcse-skip-blocks	107
-fdata-sections	107
-fdefer-pop. See -fno-defer	
-fexpensive-optimizations	107
-ffixed	117, 308

-ffreestanding	98	Call Conventions	200
-ffunction-sections	107	Calls, Preserving Registers	202
-fgcse	107	Parameters	200
-fgcse-lm	108	Pointers	180, 198
-fgcse-sm	108	function	
File Extensions	87	parameters	159
File Naming Convention	86	pointers	134
file types		size limits	196
input	86	specifiers	187
Files	248	functions	
--fill	115	absolute	29
fillupper Attribute	144	location of	65
-finline-functions	104, 106, 110, 197	size of	65
-finline-limit	110	static	187
-finstrument-functions	192	-funroll-all-loops	106, 109
-fkeep-inline-functions	110, 197	-funroll-loops	106, 109
-fkeep-static-consts	110	-funsigned-bitfields	99
Flags, Positive and Negative	110, 117	-funsigned-char	99
float	118, 129, 144, 185, 200	-fverbose-asm	118
Floating	129	G	
Floating Point	243	-g	105
Floating Types, Complex	135	--gc-sections	115
floating-point constant suffixes	137	general registers	220
-fno	110, 117	getenv	249
-fno-asm	98	Global Register Variables	308
-fno-builtin	99	Guidelines for Writing ISRs	205
-fno-defer-pop	108	H	
-fno-function-cse	110	-H	111
-fno-ident	117	header file	
-fno-inline	110	search path	22
-fno-keep-static-consts	110	Header Files	75, 86, 87, 111, 112, 113, 114
-fno-peephole	108	header files	21, 225
-fno-peephole2	108	--heap	179
-fno-short-double	118	--help	98
-fno-show-column	111	help!	45
-fno-signed-bitfields	99	Hex File	89
-fno-unsigned-bitfields	99	hexadecimal constants	
-fno-verbose-asm	118	C code	136
-fomit-frame-pointer	106, 110	High-Priority Interrupts	210
-foptimize-register-move	108	I	
-foptimize-sibling-calls	110	-I	86, 112, 114
format Attribute	191	-l	112, 114
format_arg Attribute	191	Identifiers	241
-fpack-struct	118	identifiers	
-fpcc-struct-return	118	unique length of	23
Frame Pointer (W14)	77, 110, 117, 160	-idirafter	112
-fregmove	108	-imacros	112, 114
-frename-registers	108	imag	135
-frerun-cse-after-loop	108, 109	Implementation-Defined Behavior	239
-frerun-loop-opt	108	implementation-defined behaviour	125
-fschedule-insns	108	-include	112, 114
-fschedule-insns2	108	incremental builds	90
-fshort-enums	118	Inhibit Warnings	99
-fsigned-bitfields	99	Inline	104, 106, 110, 220
-fsigned-char	99	inline	110, 197
-fstrength-reduce	108, 109	Inline Functions	197
-fstrict-aliasing	106, 109	input files	86
-fsyntax-only	99	installation, see compiler installation & activation	
-fthread-jumps	106, 109		
Function			

MPLAB® XC16 C Compiler User's Guide

int	128, 144, 185, 200	--fill	115
Integer	220	--gc-sections	115
Behavior	242	-L	115
Types, Complex	135	-l	116
integer constants	136	-legacy-libc	115
integer suffixes	136	-nodefaultlibs	116
integral promotion	183	-nostdlib	116
intermediate files	86	-s	116
Internet Address, Microchip	344	-u	116
Interrupt		-Wl	116
Enabling/Disabling	209	-Xlinker	116
Functions	217	LL, Suffix	128
Handling	217	Local Register Variables	308, 309
High Priority	210	long	128, 144, 185, 200
Latency	208	long _Fract	200
Low Priority	210	long double	118, 129, 144, 185, 200
Nesting	208	long long	104, 128, 144, 185
Priority	208	long long int	128
Protection From	212	Loop Optimization	191
Service Routine Context Saving	208	Loop Optimizer	108
Vectors	207	Loop Unrolling	77, 109
Vectors, Writing	207	Low-Priority Interrupts	210
interrupt Attribute	192, 195, 206, 208, 233	M	
interrupts		-M	113
disabling	60	Mabonga	233, 309
-iprefix	112	macro	75, 111, 112, 114, 198
ISR		MacrosData Memory Allocation	158
Coding	206	main function	21, 215
Guidelines for Writing	205	main-line code	204
Syntax for Writing	205	make files	90
Writing	205	map files	93
-isystem	112	-mauxflash	97
-iwithprefix	112	-mconst-in-auxflash	95, 180, 198
-iwithprefixbefore	112	-mconst-in-code	95, 180, 198
K		-mconst-in-data	95, 180, 198
keep Attribute	192	-mcpu	96
L		-MD	113
-L	115	Memory	249
-l	116	memory	
L constant suffix	136	remaining	66
Large Code Model	96	summary	66
Large Data Model	96	memory allocation	155
Latency	208	data memory	156
-legacy-libc	115	function code	196
lib directory	92	non-auto variables	156
Librarian	88	static variables	157
librarian	237	Memory Models	14, 180, 198
libraries		-mconst-in-auxflash	180, 198
creating	48	-mconst-in-code	180, 198
replacing modules in	237	-mconst-in-data	180, 198
user defined	92	-mlarge-code	180, 198
Library	116, 225	-mlarge-data	180, 198
ANSI Standard	14	-msmall-code	180, 198
Functions	246	-msmall-data	180, 198
limits.h header file	128, 129	-msmall-scalar	180, 198
Linker	88, 116	Memory Spaces	157
Linker Script	119, 122	-menable-fixed	98, 130
Linker Scripts	72	-merrata	95
Linking Options	115	messages	
		meaning	67

-MF	113	-fcse-follow-jumps	107
-mfillupper	95	-fcse-skip-blocks	107
-MG	113	-fdata-sections	107
Mixing Assembly Language and C Variables and Functions	217	-fexpensive-optimizations	107
-mlarge-arrays	96	-ffunction-sections	107
-mlarge-code	96, 180, 198	-fgcse	107
-mlarge-data	96, 180, 198	-fgcse-lm	108
-MM	113	-fgcse-sm	108
-MMD	113	-finline-functions	110
-mno-isr-warn	96	-finline-limit	110
-mno-pa	96	-fkeep-inline-functions	110
mode Attribute	144	-fkeep-static-consts	110
modules	88	-fno-defer-pop	108
-momf=	96	-fno-function-cse	110
-MP	113	-fno-inline	110
-mpa	96	-fno-peephole	108
-mpa=	96	-fno-peephole2	108
MPLAB X IDE	69	-fomit-frame-pointer	110
project properties options	48	-foptimize-register-move	108
-MQ	113	-foptimize-sibling-calls	110
-msmall-code	96, 180, 198, 199	-fregmove	108
-msmall-data	97, 180, 181, 198	-frename-registers	108
-msmall-scalar	97, 180, 181, 198	-frerun-cse-after-loop	108
-msmart-io	97	-frerun-loop-opt	108
-MT	113	-fschedule-insns	108
-mtext=	97	-fschedule-insns2	108
myMicrochip Personalized Notification Service	343	-fstrength-reduce	108
N		-fstrict-aliasing	109
naked Attribute	192	-fthread-jumps	109
Near and Far Code	199	-funroll-all-loops	109
Near and Far Data	181, 199	-funroll-loops	109
near Attribute	145, 181, 192, 199, 220	-O	106
Near Data Section	181	-O0	106
Near Data Space	222	-O1	106
Nesting Interrupts	208	-O2	106
no_instrument_function Attribute	192	-O3	106
-nodefaultlibs	116	-Os	106
noload Attribute	145, 192	Optimization, Loop	108, 191
non-volatile RAM	138	Optimization, Peephole	108
noreturn Attribute	104, 193	optimizations	
-nostdinc	112, 114	causing corruption	60
-nostdlib	116	faster code	63
NULL macro	28	Options	
NULL pointers	135	16-Bit Specific	95
O		Assembling	114
-O	105, 106	C Dialect Control	98
-o	89, 97	Code Generation Conventions	117
-O0	106	Debugging	105
-O1	106	Directory Search	117
-O2	106	Linking	115
-O3	106	Optimization Control	106
Object File	76, 88, 107, 113, 115, 116, 225	Output Control	97
Optimization	14	Preprocessor Control	111
Optimization Control Options	106	Warnings and Errors Control	99
-falign-functions	107	-Os	106
-falign-labels	107	Output Control Options	97
-falign-loops	107	-c	97
-fcaller-saves	107	-E	97
		--help	98
		-o	97

MPLAB® XC16 C Compiler User's Guide

-S	97	-MF	113
-v	97	-MG	113
-x	98	-MM	113
output files		-MMD	113
names of	93	-MQ	113
P		-MT	113
-P	114	-nostdinc	114
packed Attribute	118, 145	-P	114
Packing Data Stored in Flash	175	-trigraphs	114
Parameters, Function	200	-U	114
parameters, see function, parameters		-undef	114
PATH	88	preprocessor macros	
-pedantic	99, 104	predefined	42
-pedantic-errors	99	Preserving Registers Across Function Calls	202
Peephole Optimization	108	printf function	61
persistent Attribute	146	Procedural Abstraction	96
persistent data	159, 216	Processor ID	96
PIC30_C_INCLUDE_PATH	86	Program Memory Pointers	180, 198
PIC30_COMPILER_PATH	86	project name	93
PIC30_LIBRARY_PATH	87	Project Properties	73
pic30-ar	69	MPLAB XC16 Assembler Options	73
pic30-as	69	MPLAB XC16 C Compiler Options	75
pic30-gcc	85	MPLAB XC16 Object Linker Options	77
pic30-ld	69	XC16 Toolsuite Global Options	73
pointer	185, 200	Projects	71
comparisons	135	projects	90
definitions	133	PSV Usage	177
qualifiers	133	PSV Window	163, 177, 180, 198
types	133	Q	
Pointers	104	-Q	105
Frame	77, 110, 117	qualifier	
Function	180, 198	__align	34
Stack	117	__bank	33
pointers	133	__deprecate	39
assigning integers	134	__eeprom	35
function	134	__far	30
Predefined Constants	230, 307	__interrupt	35
prefix	112	__near	31
preprocessing	229	__pack	38
Preprocessing Directives	245	__persistent	32
Preprocessor Control Options	111	__section	39
-A	111	__xdata	33
-C	111	__ydata	33
-D	111	auto	159
-dD	111	const	138, 166
-dM	111	volatile	60, 138
-dN	111	Qualifiers	244
-fno-show-column	111	qualifiers	138
-H	111	and auto variables	159
-I	112	and structures	131
-I-	112	R	
-idirafter	112	radix specifiers	
-imacros	112	C code	136
-include	112	RAW Dependency	108
-iprefix	112	Reading, Recommended	12
-isystem	112	read-only variables	138
-iwithprefix	112	real	135
-iwithprefixbefore	112	Reduce Code Size	95, 106
-M	113	Register	
-MD	113		

Behavior	244
Definition Files	119
register	308, 309
registers	
used by functions	186
replacing library modules	237
Reset	207, 208, 209
Return Type	100
Return Value	202
reverse Attribute	146
rotate operator	61
round Attribute	193
runtime startup code	215

S

-S	97, 115
-s	116
safeguarding code	60
save(list) Attribute	193
-save-temps	106
Scalars	180, 198
Scheduling	108
section	76, 107
section Attribute	146, 193, 196, 233, 309
secure Attribute	147, 194
SFR	14, 89, 119
sfr Attribute	147
SFRs	121
shadow Attribute	195, 206, 233
short	128, 185, 200
Signals	247
signed char	128
signed int	128
signed long	128
signed long long	128
signed short	128
Simulator, Command-Line	14, 88
size limits	64
const variables	166
Small Code Model	14, 96
Small Data Model	14, 97
Software Stack	159, 160, 195
Source Code	72
source files	88
space Attribute	147, 233
Special Function Registers	89, 208
special function registers, see SFRs	
Specifying Registers for Local Variables	309
-specs=	117
SPLIM	120, 159
Stack	208
C Usage	160
Overflow	68
Pointer (W15)	117, 120, 159, 160, 215
Pointer Limit Register (SPLIM)	120, 159, 215
Software	159, 160
Standard I/O Functions	14
Startup	
and Initialization	92
Module, Alternate	92, 216
Module, Primary	92, 215

Modules	160
Statements	245
static functions	187
static variables	157
storage duration	156
Streams	248
strerror	250
string literals	137
storage location	137
type of	137
struct types, see structures	
structure	185, 200
structure qualifiers	131
structure, bit fields	131
Structures	244
structures	131
bit-fields in	131
Suffix LL	128
Suffix ULL	128
switch	101
symbol	116
Syntax Check	99
Syntax for Writing ISRs	205
system	249
System Header Files	101, 113

T

-T	119
TBLRD	178
temporary variables	159
TMPDIR	87
tmpfile	249
-traditional	99, 197
Traditional C	105
Translation	240
translation units	88
transparent_union Attribute	149
Trigraphs	101, 114
-trigraphs	114
Type Conversion	104
type conversions	183

U

-U	111, 112, 114
-u	116
U constant suffix	136
ULL, Suffix	128
unnamed bit-fields	132
-undef	114
Underscore	205, 217
Unions	244
unions	
qualifiers	131
unnamed structure members	132
unordered Attribute	149
Unroll Loop	77, 109
unsigned char	128
unsigned int	128
unsigned long	128
unsigned long long	128
unsigned long long int	128

MPLAB® XC16 C Compiler User's Guide

unsigned short	128	-Wimplicit-int	100
unsupported Attribute	149, 195	-Winline	104
unused Attribute	102, 149, 195	-Wlarger-than-	104
Unused Function Parameter	102	-Wlong-long	104
unused memory	66	-Wmain	100
Unused Variable	102	-Wmissing-braces	100
unused variables		-Wmissing-declarations	104
removing	138	-Wmissing-format-attribute	104
USB	365	-Wmissing-noreturn	104
user_init Attribute	195	-Wmissing-prototypes	104
User-Defined Data Section	309	-Wmultichar	100
User-Defined Text Section	196, 309	-Wnested-externs	104
Using Inline Assembly Language	220	-Wno-long-long	104
V		-Wno-multichar	100
-v	97	-Wno-sign-compare	105
Variable Attributes	142	-Wpadded	104
variables		-Wparentheses	100
absolute	29	-Wpointer-arith	104
auto	159	-Wredundant-decls	104
location of	65	-Wreturn-type	100
maximum size of	64	-Wsequence-point	101
static	157	-Wshadow	104
storage duration	156	-Wsign-compare	105
Variables in Specified Registers	308	-Wstrict-prototypes	105
void	185	-Wswitch	101
volatile qualifier	60, 138	-Wsystem-headers	101
W		-Wtraditional	105
-W	99, 102, 103, 105, 257	-Wtrigraphs	101
-w	99	-Wundef	105
W Registers	200, 217	-Wuninitialized	102
W14	160	-Wunknown-pragmas	102
W15	160	-Wunused	102
-Wa	114	-Wunused-function	102
-Waggregate-return	103	-Wunused-label	102
-Wall	99, 102, 103, 105	-Wunused-parameter	102
warning messages	67	-Wunused-value	102
location	67	-Wunused-variable	102
suppressing	68	-Wwrite-strings	105
Warnings	276	Warnings, Inhibit	99
Warnings and Errors Control Options	99	Watchdog Timer	365
-fsyntax-only	99	-Wbad-function-cast	103
-pedantic	99	-Wcast-align	103
-pedantic-errors	99	-Wcast-qual	103
-W	103	-Wchar-subscripts	99
-w	99	-Wcomment	99
-Waggregate-return	103	-Wconversion	104
-Wall	99	-Wdiv-by-zero	99
-Wbad-function-cast	103	weak Attribute	149, 195
-Wcast-align	103	Web Site, Microchip	344
-Wcast-qual	103	-Werror	104
-Wchar-subscripts	99	-Werror-implicit-function-declaration	99
-Wcomment	99	-Wformat	99, 104, 191
-Wconversion	104	-Wimplicit	99
-Wdiv-by-zero	99	-Wimplicit-function-declaration	100
-Werror	104	-Wimplicit-int	100
-Werror-implicit-function-declaration	99	-Winline	104, 197
-Wformat	99	-WI	116
-Wimplicit	99	-Wlarger-than-	104
-Wimplicit-function-declaration	100	-Wlong-long	104
		-Wmain	100

-Wmissing-braces	100
-Wmissing-declarations	104
-Wmissing-format-attribute	104
-Wmissing-noreturn	104
-Wmissing-prototypes	104
-Wmultichar	100
-Wnested-externs	104
-Wno-	99
-Wno-deprecated-declarations	104
-Wno-div-by-zero	99
-Wno-long-long	104
-Wno-multichar	100
-Wno-sign-compare	103, 105
-Wpadded	104
-Wparentheses	100
-Wpointer-arith	104
-Wredundant-decls	104
-Wreturn-type	100
Writing an Interrupt Service Routine	205
Writing the Interrupt Vector	207
-Wsequence-point	101
-Wshadow	104
-Wsign-compare	105
-Wstrict-prototypes	105
-Wswitch	101
-Wsystem-headers	101
-Wtraditional	105
-Wtrigraphs	101
-Wundef	105
-Wuninitialized	102
-Wunknown-pragmas	101, 102
-Wunused	102, 103
-Wunused-function	102
-Wunused-label	102
-Wunused-parameter	102
-Wunused-value	102
-Wunused-variable	102
-Wwrite-strings	105

X

-x	98
XC16_C_INCLUDE_PATH	86
XC16_COMPILER_PATH	86
XC16_EXEC_PREFIX	87
XC16_LIBRARY_PATH	87
XC16_VERSION	230
-Xlinker	116

Worldwide Sales and Service

AMERICAS

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

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

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

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

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

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

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

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

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

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

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

ASIA/PACIFIC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ASIA/PACIFIC

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

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

India - Pune
Tel: 91-20-3019-1500

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

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

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

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

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

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

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

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

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

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

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

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

EUROPE

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

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

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

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

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

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

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

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