



MPLAB® XC16 C Compiler User Guide

Notice to Customers



Important:

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. Please refer to our website (www.microchip.com) to obtain the latest documentation available.

Documents are identified with a "DS" number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is "DSXXXXXA," where "XXXXX" is the document number and "A" is the revision level of the document.

For the most up-to-date information on development tools, see the MPLAB® X IDE online help. Select the Help menu and then Topics, to open a list of available online help files.



Table of Contents

Notice to Customers.....	1
1. Preface.....	7
1.1. Conventions Used in This Guide.....	7
1.2. Recommended Reading.....	8
2. Compiler Overview.....	9
2.1. Device Description.....	9
2.2. Compiler Description and Documentation.....	9
3. Compiler and Other Development Tools.....	11
4. Common C Interface.....	12
4.1. Background – The Desire for Portable Code.....	12
4.2. Using the CCI.....	13
4.3. ANSI Standard Refinement.....	14
4.4. ANSI Standard Extensions.....	20
4.5. Compiler Features.....	31
5. How To's.....	33
5.1. Installing and Activating the Compiler.....	33
5.2. Invoking the Compiler.....	34
5.3. Writing Source Code.....	35
5.4. Getting My Application to Do What I Want.....	42
5.5. Understanding the Compilation Process.....	44
5.6. Fixing Code That Does Not Work.....	49
6. XC16 Toolchain and MPLAB X IDE.....	51
6.1. MPLAB X IDE and Tools Installation.....	51
6.2. MPLAB X IDE Setup.....	51
6.3. MPLAB X IDE Projects.....	52
6.4. Operation Summary.....	53
6.5. References.....	54
6.6. Project Setup.....	54
6.7. Project Example.....	66
7. Compiler Command-Line Driver.....	69
7.1. Invoking the Compiler.....	69
7.2. The Compilation Sequence.....	71
7.3. Runtime Files.....	74
7.4. Compiler Output.....	75
7.5. Compiler Messages.....	76
7.6. Driver Option Descriptions.....	76
7.7. MPLAB X IDE Toolchain Equivalents.....	98
8. Device-Related Features.....	99
8.1. Device Support.....	99
8.2. Device Header Files.....	99

8.3.	Stack.....	100
8.4.	Configuration Bit Access.....	101
8.5.	Using SFRs.....	101
8.6.	Bit-Reversed and Modulo Addressing.....	102
8.7.	Using EDS.....	103
8.8.	Stack Usage Guidance.....	104
9.	Differences Between MPLAB XC16 and ANSI C.....	107
9.1.	Divergence from the ANSI C Standard.....	107
9.2.	Extensions to the ANSI C Standard.....	107
9.3.	Implementation-Defined Behavior.....	107
10.	Supported Data Types and Variables.....	108
10.1.	Identifiers.....	108
10.2.	Integer Data Types.....	108
10.3.	Floating-Point Data Types.....	109
10.4.	Fixed-Point Data Types.....	110
10.5.	Structures and Unions.....	111
10.6.	Pointer Types.....	112
10.7.	Literal Constant Types and Formats.....	114
10.8.	Standard Type Qualifiers.....	116
10.9.	Compiler-Specific Type Qualifiers.....	116
10.10.	Variable Attributes.....	118
11.	Fixed-Point Arithmetic Support.....	126
11.1.	Enabling Fixed-Point Arithmetic Support.....	126
11.2.	Data Types.....	126
11.3.	Rounding.....	127
11.4.	Division By Zero.....	127
11.5.	External Definitions.....	127
11.6.	Mixing C and Assembly Language Code.....	128
12.	Memory Allocation and Access.....	129
12.1.	Address Spaces.....	129
12.2.	Variables In Data Space Memory.....	129
12.3.	Variables in Program Space.....	135
12.4.	Parallel Master Port Access.....	139
12.5.	External Memory Access.....	140
12.6.	Extended Data Space Access.....	144
12.7.	Dataflash Memory Access.....	145
12.8.	Dual Partition Memory Access.....	145
12.9.	Packing Data Stored in Flash.....	145
12.10.	Allocation of Variables to Registers.....	146
12.11.	Variables in EEPROM Data Space (Device Dependent).....	146
12.12.	Dynamic Memory Allocation.....	148
12.13.	Co-Resident Applications.....	148
12.14.	Memory Models.....	148
13.	Operators and Statements.....	151

13.1. Built-In Functions.....	151
13.2. Integral Promotion.....	151
14. Register Usage.....	153
14.1. Register Variables.....	153
14.2. Changing Register Contents.....	153
15. Functions.....	155
15.1. Writing Functions.....	155
15.2. Function Size Limits.....	161
15.3. Allocation of Function Code.....	161
15.4. Changing the Default Function Allocation.....	161
15.5. Inline Functions.....	162
15.6. Memory Models.....	163
15.7. Function Call Conventions.....	163
16. Interrupts.....	166
16.1. Interrupt Operation.....	166
16.2. Writing an Interrupt Service Routine.....	166
16.3. Specifying the Interrupt Vector.....	168
16.4. Interrupt Service Routine Context Saving.....	169
16.5. Nesting Interrupts.....	170
16.6. Enabling/Disabling Interrupts.....	170
16.7. ISR Considerations.....	171
17. Main, Runtime Startup and Reset	176
17.1. The main Function.....	176
17.2. Runtime Startup and Initialization.....	176
18. Mixing C and Assembly Code.....	178
18.1. Mixing Assembly Language and C Variables and Functions.....	178
18.2. Using Inline Assembly Language.....	179
18.3. Predefined Assembly Macros.....	183
19. Library Routines.....	184
20. Optimizations.....	185
20.1. Optimization Feature Summary.....	185
20.2. How to Enable Optimization.....	186
20.3. Using Optimizations.....	187
21. Preprocessing.....	191
21.1. C Language Comments.....	191
21.2. Preprocessing Directives.....	191
21.3. Predefined Macro Names.....	192
22. Linking Programs.....	195
22.1. Default Memory Spaces.....	195
22.2. Replacing Library Symbols.....	196
22.3. Linker-Defined Symbols.....	196

22.4. Default Linker Script.....	196
23. Implementation-Defined Behavior.....	198
23.1. Translation.....	198
23.2. Environment.....	198
23.3. Identifiers.....	199
23.4. Characters.....	199
23.5. Integers.....	199
23.6. Floating Point.....	200
23.7. Arrays and Pointers.....	201
23.8. Registers.....	201
23.9. Structures, Unions, Enumerations and Bit-Fields.....	201
23.10. Qualifiers.....	202
23.11. Declarators.....	202
23.12. Statements.....	202
23.13. Preprocessing Directives.....	202
23.14. Library Functions.....	203
23.15. Signals.....	204
23.16. Streams and Files.....	204
23.17. tmpfile.....	205
23.18. errno.....	205
23.19. Memory.....	205
23.20. abort.....	205
23.21. exit.....	205
23.22. getenv.....	206
23.23. system.....	206
23.24. strerror.....	206
24. Embedded Compiler Compatibility Mode.....	207
24.1. Compiling in Compatibility Mode.....	207
24.2. Syntax Compatibility.....	207
24.3. Data Type.....	208
24.4. Operator.....	208
24.5. Extended Keywords.....	209
24.6. Intrinsic Functions.....	210
24.7. Pragmas.....	210
25. Diagnostics.....	212
25.1. Errors.....	212
25.2. Warnings.....	230
26. GNU Free Documentation License.....	250
27. Deprecated Features.....	255
27.1. Predefined Constants.....	255
27.2. Variables in Specified Registers.....	255
27.3. Changing Non-Auto Variable Allocation.....	256
27.4. Configuration Settings Using Macros.....	257

28. Built-in Functions.....	258
28.1. Built-In Functions vs. Inline Assembly.....	258
28.2. Built-In Function Descriptions.....	258
29. Document Revision History.....	298
The Microchip Website.....	302
Product Change Notification Service.....	302
Customer Support.....	302
Product Identification System.....	303
Microchip Devices Code Protection Feature.....	303
Legal Notice.....	304
Trademarks.....	304
Quality Management System.....	305
Worldwide Sales and Service.....	306

1. Preface

MPLAB[®] XC16 C Compiler documentation and support information is discussed in this section.

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	<i>MPLAB[®] IDE 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
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	#define START
	Filenames	autoexec.bat
	File paths	c:\mcc18\h
	Keywords	_asm, _endasm, static
	Command-line options	-Opa+, -Opa-
	Bit values	0, 1
	Constants	0xFF, 'A'
Italic Courier New	A variable argument	<i>file.o</i> , where <i>file</i> can be any valid filename
Square brackets []	Optional arguments	mcc18 [options] file [options]
Curly brackets and pipe character: { }	Choice of mutually exclusive arguments; an OR selection	errorlevel {0 1}

.....continued		
Description	Represents	Examples
Ellipses...	Replaces repeated text	var_name [, var_name...]
	Represents code supplied by user	void main (void) { ... }
Special Icon		
DD	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	xmemory attribute

1.2 Recommended Reading

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

MPLAB® XC16 Assembler, Linker and Utilities User's Guide (DS50002106)

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

16-Bit Language Tools Libraries (DS50001456)

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, including:

- 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.

2. Compiler Overview

The MPLAB XC16 C compiler is defined and described in this section.

2.1 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.

2.2 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[®] OS, Linux[®] OS and Mac[®] OS X[®].

The compiler can be licensed as Free or PRO. The Free license has the minimum optimizations whereas the PRO license has the maximum (for details see [20. Optimizations](#)). The basic compiler operation, supported devices and available memory are identical across all modes.

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

2.2.1 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.

2.2.2 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 section [20. Optimizations](#)

2.2.3 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, timekeeping 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.

2.2.4 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.

2.2.5 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 [10.9 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.

2.2.6 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.

2.2.7 Documentation

The compiler is supported under both the MPLAB® X IDE and MPLAB IDE v8.xx and above. In this document, only the MPLAB X IDE is discussed.

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 [1. Preface](#)).

3. 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* (DS50002106)
- MPLAB X IDE
- 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

4. Common C Interface

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.

4.1 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 can 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 can 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.

4.1.1 The ANSI Standard

The ANSI C Standard has to reconcile two opposing goals: freedom for compiler 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 also 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 cannot allow the compiler to conform.⁽¹⁾ But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would lose its effectiveness.

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 in which 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 was 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 MPLAB 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.

4.1.2 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 <code>int</code> , 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 can differ across each compiler, and any arguments to the keywords can be device specific.
Coding guidelines	The CCI can 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.

4.2 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 can choose to modify existing projects so they conform.

For your project to conform to the CCI, you must complete the following tasks.

¹ For example, 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.

- **Enable the CCI**
Select the MPLAB X 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 a non-CCI feature has been used and the CCI is enabled.

4.3 ANSI Standard Refinement

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

4.3.1 Source File Encoding

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

Example

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

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

Differences

All compilers have used this character set.

Migration to the CCI

No action required.

4.3.2 The Prototype for main

The prototype for the `main()` function is:

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

Example

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

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

Differences

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

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.

4.3.3 Header File Specification

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

Example

The following example shows two conforming include directives.

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

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 was compiled under other host operating systems. Under the CCI, no directory separators should be used.

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 X 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 X IDE project properties, or on the command line as follows:

```
-Ilcd
```

4.3.4 Include Search Paths

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

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

Header files specified in quote characters `" "` should be discoverable in the current working directory or in the same directories that are searched when the header files are specified in angle bracket delimiters (as above). 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 point to the same directories searched when the header file is specified in angle bracket delimiters.

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

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. A header file being located elsewhere does not conform to the CCI.

Differences

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

Migration to the CCI

Remove any option that specifies header file search paths other than the `-I` option (or the equivalent MPLAB X 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.

4.3.5 The Number of Significant Initial Characters in an Identifier

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

Example

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

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

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.

Migration to the CCI

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

4.3.6 Sizes of Types

The sizes of the basic C types, e.g., `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.
- Those that have a fixed size, regardless of the target.

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;
```

Differences

This is consistent with previous types implemented by the compiler.

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>`.

4.3.7 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.

Example

The following example

```
char foobar;
```

defines an `unsigned char` object called `foobar`.

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`.

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 can use the `-funsigned-char` option on MPLAB XC16 and XC32 to change the type of plain `char`, but since this option is not supported on MPLAB XC8, the code is not strictly conforming.

4.3.8 Signed Integer Representation

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

Example

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

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

Differences

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

Migration to the CCI

No action required.

4.3.9 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.

Example

The following shows an assignment of a value that is truncated.

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

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

Differences

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

Migration to the CCI

No action required.

4.3.10 Bitwise Operations on Signed Values

Bitwise operations on signed values act on the two's complement representation, including the sign bit. See also [4.3.11 Right-shifting Signed Values](#)

Example

The following example shows 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 is 0x72.

Differences

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

Migration to the CCI

No action required.

4.3.11 Right-shifting Signed Values

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

Example

The following shows an example of a negative quantity involved in a right-shift operation.

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

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

Differences

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

Migration to the CCI

No action required.

4.3.12 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) it is considered 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.

Example

The following example shows 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;
```

Differences

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

Migration to the CCI

No action required.

4.3.13 Default Bit-field `int` Type

The type of a bit-field specified as a plain `int` is identical to that of one defined using `unsigned int`. This is quite different from 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.

Example

The following example shows a structure tag containing bit-fields that are unsigned integers with the size specified.

```
struct OUTPUTS {
    int direction :1;
```

```
int parity :3;
int value :4;
};
```

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.

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 the following example:

```
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;
};
```

4.3.14 Bit-fields Straddling a Storage Unit Boundary

The standard indicates that implementations can determine whether bit-fields cross a storage unit boundary. In the CCI, bit-fields do not straddle a storage unit boundary; a new storage unit is allocated to the structure, and padding bits 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.

Example

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

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

Under the CCI and using MPLAB XC8, the storage allocation unit is byte sized. The bit-field, `second`, is 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`, is 2 bytes.

Differences

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

Migration to the CCI

No action required.

4.3.15 The Allocation Order of Bit-fields

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 is the least significant bit of the storage unit in which it is allocated.

Example

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

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

The bit-field `lo` is assigned the least significant bit of the storage unit assigned to the structure `foo`. The bit-field `mid` is assigned the next 6 least significant bits; and `hi`, the most significant bit of that same storage unit byte.

Differences

This is identical with the previous operation of all compilers.

Migration to the CCI

No action required.

4.3.16 The `NULL` Macro

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

Example

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

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

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

Differences

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

Migration to the CCI

No action required.

4.3.17 Floating-point Sizes

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

Example

The following example shows the definition for `outY`, which is at least 32 bits in size.

```
float outY;
```

Differences

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

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.

4.4 ANSI Standard Extensions

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

4.4.1 Generic Header File

A single header file `<x.c.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.

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>
```

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.

Migration to the CCI

Change:

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

previously used in 8-bit compiler code, or family-specific header files, e.g., from:

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

to:

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

4.4.2 Absolute Addressing

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

Example

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

```
int scanMode __at(0x200);
const char keys[] __at(124) = { 'r', 's', 'u', 'd' };

__at(0x1000) int modify(int x) {
    return x * 2 + 3;
}
```

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.

Migration to the CCI

Avoid making objects and functions absolute if possible.

In MPLAB XC8, change absolute object definitions, e.g., from:

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

to:

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

In MPLAB XC16 and XC32, change code, for example, from:

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

to:

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

Caveats

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

The `__at()` specifier must be placed at the beginning of function prototypes for the 16- and 32-bit compilers. If you prefer to use the specifier at the end of the prototype, use the specifier with a declaration and leave it off the definition, for example:

```
int modify(int x) __at(0x1000);
int modify(int x)
{ ... }
```

4.4.3 Far Objects and Functions

The `__far` qualifier can be used to indicate that variables or functions are 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 usually 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 is ignored. Stack-based (`auto` and `parameter`) variables cannot use the `__far` specifier.

Example

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

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

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.

Migration to the CCI

For 8-bit compilers, change any occurrence of the `far` qualifier, e.g., from:

```
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, for example, from:

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

to:

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

Caveats

None.

4.4.4 Near Objects

The `__near` qualifier can be used to indicate that variables or functions are 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 generally are 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 is ignored. Stack-based (auto and parameter) variables cannot use the `__near` specifier.

Example

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

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

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.

Migration to the CCI

For 8-bit compilers, change any occurrence of the `near` qualifier to `__near`, e.g., from:

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

to:

```
__near char template[20];
```

In 16- and 32-bit compilers, change any occurrence of the `near` attribute to `__near`, for example, from:

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

to:

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

Caveats

None.

4.4.5 Persistent Objects

The `__persistent` qualifier can 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.

Example

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

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.

Migration to the CCI

With 8-bit compilers, change any occurrence of the `persistent` qualifier to `__persistent`, e.g., from:

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

to:

```
__persistent char template[20];
```

For the 16- and 32-bit compilers, change any occurrence of the `persistent` attribute to `__persistent`, for example, from:

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

to:

```
int __persistent tblIdx;
```

Caveats

None.

4.4.6 X and Y Data Objects

The `__xdata` and `__ydata` qualifiers can be used to indicate that variables are 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 is ignored.

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];
```

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.

Migration to the CCI

For 16-bit compilers, change any occurrence of the space attributes `xmemory` or `ymemory` to `__xdata`, or `__ydata` respectively, for example, from:

```
char __attribute__((space(xmemory))) template[20];
```

to:

```
__xdata char template[20];
```

Caveats

None.

4.4.7 Banked Data Objects

The `__bank(num)` qualifier can be used to indicate that variables are 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 is ignored. The number of data banks implemented will vary from one device to another.

Example

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

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


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.

Migration to the CCI

For 8-bit compilers, change any occurrence of the `bankx` qualifiers to `__bank()`, e.g.,

from:

```
bank2 int logEntry;
```

to:

```
__bank(2) int logEntry;
```

Caveats

This feature is not yet implemented in MPLAB XC8.

4.4.8 Alignment of Objects

The `__align(alignment)` specifier can 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 2. Positive values request that the object's start address be aligned.

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];
```

Differences

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

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

Migration to the CCI

For 16- and 32-bit compilers, change any occurrence of the `aligned` attribute to `__aligned`, for example, from:

```
char __attribute__((aligned(4))) mode;
```

to:

```
__align(4) char mode;
```

4.4.9 EEPROM Objects

The `__eeprom` qualifier can 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 generates a warning. Stack-based (`auto` and `parameter`) variables cannot use the `__eeprom` specifier.

Example

The following shows a variable qualified using `__eeprom`.

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

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.

Migration to the CCI

For 8-bit compilers, change any occurrence of the `eeeprom` qualifier to `__eeeprom`, e.g., from:

```
eeeprom char title[20];
```

to:

```
__eeeprom char title[20];
```

For 16-bit compilers, change any occurrence of the `eedata space` attribute to `__eeeprom`, for example, from:

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

to:

```
int __eeeprom mainSw;
```

Caveats

MPLAB XC8 does not implement the `__eeeprom` qualifiers for any PIC18 devices; this qualifier works as expected for other 8-bit devices.

4.4.10 Interrupt Functions

The `__interrupt(type)` specifier can 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. (MPLAB XC8 - PIC18 only)
high_priority	The interrupt function corresponds to the high priority interrupt source. (MPLAB XC8)
save(symbol-list)	Save the listed symbols on entry, and restore on exit. (MPLAB XC16)
irq(irqid)	Specify the interrupt vector associated with this interrupt. (MPLAB XC16 and XC8)
altirq(altirqid)	Specify the alternate interrupt vector associated with this interrupt. (MPLAB XC16)
base(address)	Specify vector table address. (MPLAB XC8)
preprologue(asm)	Specify assembly code to be executed before any compiler-generated interrupt code. (MPLAB XC16)
shadow	Allow the ISR to utilize the shadow registers for context switching. (MPLAB XC16)
auto_psv	The ISR will set the PSVPAG register and restore it on exit. (MPLAB XC16)
no_auto_psv	The ISR will not set the PSVPAG register. (MPLAB 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 generates a warning. If the argument to the `__interrupt` specifier does not make sense for the target device, a warning or error is issued by the compiler.

Example

The following shows a function qualified using `__interrupt`.

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

Differences

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 `__interrupt()` specifier may now be used outside of the CCI.

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

Migration to the CCI

For 8-bit compilers, change any occurrence of the `interrupt` qualifier, e.g., from:

```
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, e.g., from:

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

to:

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

For 32-bit compilers, the `__interrupt()` keyword takes two parameters, the vector number and the (optional) IPL value. Change code that 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) {}
```

Caveats

None.

4.4.11 Packing Objects

The `__pack` specifier can 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 cannot pad structures with alignment gaps for some devices, and use of this specifier for such devices is ignored.

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;
```

Differences

The `__pack` specifier is a new CCI specifier that is available with MPLAB 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.

Migration to the CCI

No migration is required for MPLAB XC8.

For 16- and 32-bit compilers, change any occurrence of the `packed` attribute, for example, from:

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

to:

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

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

Caveats

None.

4.4.12 Indicating Antiquated Objects

The `__deprecated` specifier can 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 can become obsolete, or that better features have been developed and should be used in preference.

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

Example

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

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

Differences

No deprecate feature was implemented on 8-bit compilers.

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

Migration to the CCI

For 16- and 32-bit compilers, change any occurrence of the `deprecated` attribute to `__deprecate`, for example, from:

```
int __attribute__((deprecated)) intMask;
```

to:

```
int __deprecate intMask;
```

Caveats

None.

4.4.13 Assigning Objects to Sections

The `__section()` specifier can be used to indicate that an object should be located in the named section. This is typically used when the object has special and unique linking requirements that 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.

Example

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

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

Differences

The 8-bit compilers have previously used the `#pragma psect` directive to redirect objects to a new section, or `psect`; however, the `__section()` specifier is the preferred method to perform this task, even if you are not using the CCI.

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.

Migration to the CCI

For MPLAB 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, for example, from:

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

to:

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

Caveats

None.

4.4.14 Specifying Configuration Bits

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

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

where *setting* is a configuration setting descriptor (for example, `WDT`), *state* is a descriptive value (for example, `ON`) and *value* is a numerical value.

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

Example

The following shows Configuration bits being specified using this pragma.

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

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`.

Migration to the CCI

For the 8-bit compilers, change any occurrence of the `__CONFIG()` macro, e.g., `__CONFIG(WDTEN & XT & DPROT)` to the `#pragma config` directive, e.g.,

```
#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, e.g., from:

```
_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.

Caveats

None.

4.4.15 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 the following table.

Table 4-1. 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>__XC16__</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>

Example

The following example shows code that 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
```

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__`).

Migration to the CCI

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

Caveats

None.

4.4.16 In-line Assembly

The `asm()` statement can be used to insert assembly code in-line with C code. The argument is a C string literal that 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.

Example

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

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

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.

Migration to the CCI

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

```
#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.

Caveats

None.

4.5 Compiler Features

The following item details the compiler options used to control the CCI.

4.5.1 Enabling the CCI

It is assumed that you are using the MPLAB X IDE to build projects that use the CCI. The widget in the MPLAB X IDE Project Properties is used to enable CCI conformance CCI Syntax in the Compiler category.

If you are not using this IDE, then the command-line option is `-mcci` for MPLAB XC16.

Differences

This option has never been implemented previously.

Migration to the CCI

Enable the option.

Caveats

None.

5. How To's

This section contains help and references for situations that are frequently encountered when building projects with 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

- [5.1 Installing and Activating the Compiler](#)
- [5.2 Invoking the Compiler](#)
- [5.3 Writing Source Code](#)
- [5.4 Getting My Application to Do What I Want](#)
- [5.5 Understanding the Compilation Process](#)
- [5.6 Fixing Code That Does Not Work](#)

5.1 Installing and Activating the Compiler

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

- [5.1.1 How Do I Install and Activate My Compiler?](#)
- [5.1.2 How Can I Tell If the Compiler Has Installed and Activated Successfully?](#)
- [5.1.3 Can I Install More Than One Version of the Same Compiler?](#)

5.1.1 How Do I Install and Activate My Compiler?

Installation of the license is performed by the XC compiler installer. Activation is available online through mySoftware. For full instructions, refer to the following document. It is available for download from the Microchip Technology website:

www.microchip.com/mplab/compilers

"Installing and Licensing MPLAB® XC C Compilers" (DS50002059).

5.1.2 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`

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:

[20. Optimizations](#)

[7.6.6 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!

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

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:

[5.2.3 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\

5.2 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, [Which Libraries Get Included by Default?](#)

5.2.1 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:

[6. XC16 Toolchain and MPLAB X IDE](#)

[5.2.3 How Can I Select Which Compiler Version to Build With?](#)

5.2.2 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:

[7. Compiler Command-Line Driver](#)

[5.2.3 How Can I Select Which Compiler Version to Build With?](#)

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

Both the compilation and installation processes were 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; with each category having several panes which can be selected from a pull-down menu that is near the top of the pane.

5.2.4 How Can I Change the Compiler Optimizations?

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

[20. Optimizations](#)

[7.6.6 Options for Controlling Optimization](#)

5.2.5 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:

[20.1 Optimization Feature Summary](#)

5.2.6 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:

[7.6 Driver Option Descriptions](#)

5.2.7 How Do I Build Libraries?

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

[7.3.1 Library Files](#)

5.2.8 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:

[6.6 Project Setup](#)

5.2.9 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.

5.3 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](#)

5.3.1 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:

[2.2.1 ANSI C Standard](#)

5.3.2 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.

- [5.3.2.1 How Do I Port My Code To Different Device Architectures?](#)
- [5.3.2.2 How Do I Set the Configuration Bits?](#)
- [5.3.2.3 How Do I Access the User ID Locations?](#)
- [5.3.2.4 How Do I Access Special Function Registers \(SFRs\)?](#)
- [5.3.2.5 Are There Any SFRs Usage Considerations?](#)
- [5.3.2.6 Which Device-Specific Symbols Does the Compiler Define, and Can I Use Them?](#)
- See also, [5.3.3.4 How Do I Stop the Compiler From Using Certain Memory Locations?](#)

5.3.2.1 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 section in this user's guide:

[4. Common C Interface](#)

5.3.2.2 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 X 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:

[8.4 Configuration Bit Access](#)

5.3.2.3 How Do I Access the User ID Locations?

Currently, the only way to 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).

5.3.2.4 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:

[8.2 Device Header Files](#)

5.3.2.5 Are There Any SFRs Usage Considerations?

The dsPIC architecture defines various Special Function Registers that control hardware peripherals or other aspects of the machine. In general these SFRs are accessed like other C variables.

Some SFRs represent memory mapped versions of CPU registers that the compiler depends upon. These registers should not be written to by a C program as this could silently damage the operation of the application, especially at higher optimization levels. Registers that should be avoided include: the memory mapped copies of the working registers (WREG0, WREG1 and so on), parts of CORCON, ACCAx, and ACCBx.

Like luggage at an airport, many SFRs look alike. That is to say, there are subtle differences between some peripheral registers from device to device. When compiling code for a generic device, avoid referring to SFR registers.

5.3.2.6 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:

[21.3 Predefined Macro Names](#)

5.3.3 Memory Allocation

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

- [5.3.3.1 How Do I Position Variables or Functions at an Address I Nominate?](#)
- [5.3.3.2 How Do I Place Variables in Program Memory?](#)
- [5.3.3.3 How Do I Allocate Space for a Variable But Not Initialize/Load Any Value?](#)
- [5.3.3.4 How Do I Stop the Compiler From Using Certain Memory Locations?](#)

5.3.3.1 How Do I Position Variables or Functions at an Address I Nominate?

Nudging the tool chain 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.

5.3.3.2 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:

[12.3 Variables in Program Space](#)

5.3.3.3 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:

[10.10 Variable Attributes](#)

5.3.3.4 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. For more on variable attributes and options, see the following sections in this user's guide:

[10.10 Variable Attributes](#)

[7.6.1 Options Specific to 16-Bit Devices](#)

Also, you can use the option `-mreserve`. See the *MPLAB® XC16 Assembler, Linker and Utilities User's Guide* (DS50002106) for details on linker scripts.

5.3.4 Variables

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

- [5.3.4.1 Why Are My Floating-Point Results Not Quite What I Am Expecting?](#)
- [5.3.4.2 How Do I Retain the Value of a Variable Even After a Soft Reset?](#)
- [5.3.4.3 How Do I Save C Variables When an ISR Is Invoked?](#)
- [5.3.4.4 How Long Can I Make My Variable and Macro Names?](#)
- [5.3.4.5 How Do I Access Values Stored in a PSV or EDS Page?](#)
- [5.3.4.6 Why Would I Need to Place Data Into Its Own Section?](#)
- [5.3.4.7 How Can I Load a Value Into Flash Memory?](#)
- [5.3.4.8 How Can I Pack Data Into Flash Memory?](#)
- [5.3.4.9 How Can I Define a Large Array?](#)
- See also, [5.4.3 How Do I Share Data Between Interrupt and Main-line Code?](#)
- See also, [5.3.3.1 How Do I Position Variables or Functions at an Address I Nominate?](#)
- See also, [5.3.3.2 How Do I Place Variables in Program Memory?](#)
- See also, [5.4.6 How Do I Place Variables in Off-Chip Memory?](#)
- See also, [5.4.8 How Can I Rotate a Variable?](#)
- See also, [5.5.14 How Do I Learn Where Variables and Functions Have Been Positioned?](#)

5.3.4.1 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, but rather 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:

[10.3 Floating-Point Data Types](#)

5.3.4.2 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 save the values of your variables after a software Reset, using 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:

[10.10 Variable Attributes](#)

5.3.4.3 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:

[16.4 Interrupt Service Routine Context Saving](#)

5.3.4.4 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:

[4.3.5 The Number of Significant Initial Characters in an Identifier](#)

5.3.4.5 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:

[10.9.1 __psv__ Type Qualifier](#)

[10.9.3 __eds__ Type Qualifier](#)

5.3.4.6 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:

[10.10 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" (DS50002106), Section 9.5 "Contents of a Linker Script".

5.3.4.7 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:

[10.10 Variable Attributes](#)

[12.3 Variables in Program Space](#)

5.3.4.8 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:

[7.6.1 Options Specific to 16-Bit Devices](#) (`-mfillupper`)

[10.10 Variable Attributes](#) (`fillupper`)

5.3.4.9 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:

[12.2.2 Non-Auto Variable Allocation and Access](#), "Non-Auto Variable Size Limits".

5.3.5 Functions

This section includes questions that relate to functions.

- [5.3.5.1 How Do I Stop A Function From Being Removed?](#)
- [5.3.5.2 Why Should I Inline My Function?](#)
- [5.3.5.3 Why is My Function Not Inline?](#)
- [5.3.5.4 Why Should I Place a Function Into its Own Section?](#)
- [5.3.5.5 How Do I Prevent the Compiler From Saving or Restoring Any Registers?](#)
- [5.3.5.6 How Can I Tell if a Function is Being Used?](#)
- [5.3.5.7 How Can I Find Out Which Functions are Contained Inside the Compiler?](#)
- [5.3.5.8 Where are Arguments That Are Passed to Functions Located in Memory?](#)
- See also, [5.5.13 How Can I Tell How Big a Function Is?](#)
- See also, [5.5.14 How Do I Learn Where Variables and Functions Have Been Positioned?](#)
- See also, [5.3.6.1 How Do I Use Interrupts in C?](#)

5.3.5.1 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 on `keep` of this user's guide:

[15.1.2 Function Attributes](#)

5.3.5.2 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 of this user's guide:

[15.5 Inline Functions](#)

5.3.5.3 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 of this user's guide:

[15.5 Inline Functions](#)

5.3.5.4 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 of this user's guide:

[15.1.2 Function Attributes](#)

5.3.5.5 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 want to save these values. For details, see the following section of this user's guide:

[15.1.2 Function Attributes](#)

5.3.5.6 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 of this user's guide:

[15.1.2 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" (DS50002106).

5.3.5.7 How Can I Find Out Which Functions are Contained Inside 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 of this user's guide:

[7.6.2 Options for Controlling the Kind of Output](#)

5.3.5.8 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 also be helpful. See the following location of this user's guide:

[28. Built-in Functions](#)

5.3.6 Interrupts

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

- [5.3.6.1 How Do I Use Interrupts in C?](#)
- [5.3.6.2 How Do I Add a Trap Interrupt Vector to a Project?](#)
- [5.3.6.3 Can/Should My Application be able to Return from a Trap?](#)
- [5.3.6.4 How Do I Share Data Between Two Interrupt Routines?](#)
- [5.3.6.5 What is the Default Interrupt, Where is it Defined, and How Do I Use It?](#)
- See also, [5.5.8 How Can I Make My Interrupt Routine Faster?](#)
- See also, [5.4.3 How Do I Share Data Between Interrupt and Main-line Code?](#)
- See also, [5.3.4.3 How Do I Save C Variables When an ISR Is Invoked?](#)

5.3.6.1 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 of this user's guide for more information:

[16. Interrupts](#)

5.3.6.2 How Do I Add a Trap Interrupt Vector to a Project?

The compiler treats hard traps the same as normal interrupt vectors, as 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) {
}
```

5.3.6.3 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.

5.3.6.4 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 a interrupt routine is, it is important that the shared data accesses are protected from further interruption as not all accesses are atomic.

5.3.6.5 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`.

5.3.7 Assembly Code

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

- [5.3.7.1 How Should I Combine Assembly and C Code?](#)
- [5.3.7.2 What Do I Need Other Than Instructions in an Assembly Source File?](#)
- [5.3.7.3 How Do I Access C Objects from Assembly Code?](#)
- [5.3.7.4 How Can I Access SFRs From Within Assembly Code?](#)
- [5.3.7.5 When Should I Combine Assembly and C Code?](#)
- [5.3.7.6 What is the Difference Between .s and .S Files?](#)
- [5.3.7.7 How Do I Make a Function Wrapper For an Assembly Module?](#)
- [5.3.7.8 When Should Inline Assembly Be Used Instead of Assembly Modules?](#)

5.3.7.1 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 of this user's guide:

[18. Mixing C and Assembly Code](#)

5.3.7.2 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” (DS50002106).

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.

5.3.7.3 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 of this user's guide:

[18.1 Mixing Assembly Language and C Variables and Functions](#)

5.3.7.4 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 of this user's guide:

[15.7 Function Call Conventions](#)

5.3.7.5 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 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.

5.3.7.6 What is the Difference Between .s and .S Files?

Both of these files should contain assembly language. `.s` files which 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 of this user's guide:

[7.4.1 Output Files](#)

5.3.7.7 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 module, see the following section of this user's guide:

[18.1 Mixing Assembly Language and C Variables and Functions](#)

5.3.7.8 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 following occurs:

- Code complies with the run-time model
- Registers are properly used and not overwritten
- Code uses the GNU extended inline assembly code

Long sequences are often hard to debug, so ensure that you correctly follow the guidelines.

5.4 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.

- [5.4.1 How Do I Generate Debug Information?](#)
- [5.4.2 Why No Disassembly in the MPLAB X IDE Disassembly Window?](#)
- [5.4.3 How Do I Share Data Between Interrupt and Main-line Code?](#)
- [5.4.4 How to Protect My Code After It Is Programmed Into a Device?](#)
- [5.4.5 How Do I Redirect Standard I/O When Using Printf?](#)
- [5.4.6 How Do I Place Variables in Off-Chip Memory?](#)
- [5.4.7 How Can I Implement a Delay in My Code?](#)

- [5.4.8 How Can I Rotate a Variable?](#)

5.4.1 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 see the following section of this user's guide:

"MPLAB® XC16 Assembler, Linker and Utilities User's Guide" (DS50002106)

[7.6.5 Options for Debugging](#)

5.4.2 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 of this user's guide:

[5.4.1 How Do I Generate Debug Information?](#)

5.4.3 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 of this user's guide:

[10.8.2 Volatile Type Qualifier](#)

[16.6 Enabling/Disabling Interrupts](#)

5.4.4 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 of this user's guide:

[8.4 Configuration Bit Access](#)

[4.4.14 Specifying Configuration Bits \(CCI\)](#)

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

5.4.5 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)

5.4.6 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 of this user's guide:

[10.9.6 __external__ Type Qualifier](#)

[10.9.5 __pmp__ Type Qualifier](#)

5.4.7 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)

5.4.8 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:

[4.3.10 Bitwise Operations on Signed Values](#) (CCI)

[4.3.11 Right-shifting Signed Values](#) (Signed variables)

5.5 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.

- [5.5.1 How Does Licensing Affect Features and Optimization Levels?](#)
- [5.5.2 Why Can't I Debug my Code after I Optimize?](#)
- [5.5.3 How Can I Make My Code Smaller?](#)
- [5.5.4 How Can I Reduce RAM Usage?](#)
- [5.5.5 How Can I Make My Code Faster?](#)
- [5.5.6 What are the Speed vs. Size Tradeoffs?](#)
- [5.5.7 How Can I Control Where the Language Tool Places Objects in Memory?](#)
- [5.5.8 How Can I Make My Interrupt Routine Faster?](#)
- [5.5.9 How Big Can C Variables Be?](#)
- [5.5.10 Which Optimizations Will Be Applied to My Code?](#)
- [5.5.11 Which Devices are Supported by the Compiler?](#)
- [5.5.12 How Do I Know What Code the Compiler Is Producing?](#)
- [5.5.13 How Can I Tell How Big a Function Is?](#)
- [5.5.14 How Do I Learn Where Variables and Functions Have Been Positioned?](#)
- [5.5.15 How Do I Properly Reserve Memory?](#)
- [5.5.16 How Do I Know How Much Memory Is Still Available?](#)
- [5.5.17 Which Libraries Get Included by Default?](#)
- [5.5.18 How Do I Create My Own Libraries?](#)
- [5.5.19 Why Do I Get Out-of-Memory Errors When I Select a Debugger?](#)
- [5.5.20 How Do I Stop My Project's Checksum From Changing?](#)
- See also, [5.6.1 How Do I Find Out What a Warning or Error Message Means?](#)
- See also, [5.2.7 How Do I Build Libraries?](#)
- See also, [5.2.9 What is Different About an MPLAB X IDE Debug Build?](#)
- See also, [5.3.5.1 How Do I Stop A Function From Being Removed?](#)

5.5.1 How Does Licensing Affect Features and Optimization Levels?

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

[20. Optimizations](#)

5.5.2 Why Can't I Debug my Code after I Optimize?

Debugging optimized code can be challenging, but not impossible. See the following section of this user's guide:

[20.3 Using Optimizations](#)

5.5.3 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:
[20. 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 of this user's guide:
[12.14 Memory Models](#)

5.5.4 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 of this user's guide:
[5.5.3 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 of this user's guide:
[12.3 Variables in Program Space](#)

5.5.5 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 of this user's guide:
[5.5.3 How Can I Make My Code Smaller?](#)
[5.5.8 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 of this user's guide:
[20. Optimizations](#)
[7.6.6 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.

5.5.6 What are the Speed vs. Size Tradeoffs?

Microchip's dsPIC architecture is primarily a 16-bit architecture. It is often less run-time efficient to use 8-bit values as the compiler may have to extend the value to use it. There are times when this will save data space, but not always. See the items below to help you make your code faster, smaller, or both.

- Array index variables and pointer offsets should always be defined as an integer sized type; `size_t` is often a good choice. A different sized integer type will require the compiler to do a conversion at run-time.
- Automatic variables (function local variables) will often be allocated into a register at compile time. A register is a minimum of 16 bits wide, so using a smaller type can require the compiler to generate extra code. Therefore,

unless 8-bit overflow rules are required, use 16-bit types instead. Also, the stack has alignment restrictions, making stack accesses for smaller type objects possibly less efficient.

- Argument transmission (parameter passing) either happens in registers or on the stack. Reduce the chance of generating conversions by avoiding the use of smaller than 16-bit objects.
- Objects that are defined at File scope, or function static scope, will consume less space if defined as 8-bit typed objects. Be aware that data sections are aligned to 16 bits, so using named sections, or one of the other attributes that might require a new section, may not provide the data size savings that are desired.
- MPLAB XC16 is free to reorder objects in File scope or automatic scope, but it is not allowed to re-order structure members. Unless a pre-defined interface is being conformed to, try to allocate structure members to group similarly-sized objects together, with bit-fields especially being grouped together. This will reduce the number padding-bytes that may be inserted to maintain alignment requirements.

5.5.7 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 of this user's guide for details:

[5.3.3 Memory Allocation](#)

5.5.8 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 of this user's guide:
[5.5.3 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 of this user's guide:
[5.5.5 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 of this user's guide:
[10.8.2 Volatile Type Qualifier](#)
[5.4.3 How Do I Share Data Between Interrupt and Main-line Code?](#)

5.5.9 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:

[12.3.3 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 of this user's guide:

[12.2 Variables In Data Space Memory](#)

[12.2.3 Auto Variable Allocation and Access](#), "Auto Variable Size Limits"

[12.2.2 Non-Auto Variable Allocation and Access](#), "Non-Auto Variable Size Limits"

5.5.10 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 of this user's guide:

[20. Optimizations](#)

[7.6.6 Options for Controlling Optimization](#)

5.5.11 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:

[8.1 Device Support](#)

5.5.12 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" (DS50002106).

5.5.13 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 of this user's guide:

[5.5.12 How Do I Know What Code the Compiler Is Producing?](#)

5.5.14 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" (DS50002106)

5.5.15 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 of this user's guide.

"MPLAB® XC16 Assembler, Linker and Utilities User's Guide" (DS50002106)

[5.3.3.4 How Do I Stop the Compiler From Using Certain Memory Locations?](#)

5.5.16 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" (DS50002106)

5.5.17 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)

5.5.18 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 -T p33FJ256GP710.gld main.c int.c liblibrary.a
```

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

```
xc16-gcc -mcpu=33FJ256GP710 -T p33FJ256GP710.gld 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:

[7.3.1 Library Files, "User_Defined Libraries"](#)

5.5.19 Why Do I Get Out-of-Memory Errors When I Select a Debugger?

If you use a hardware-tool debugger, RAM is required for debugging. See the following section in this user's guide:

[5.4.2 Why No Disassembly in the MPLAB X IDE Disassembly Window?](#)

5.5.20 How Do I Stop My Project's Checksum From Changing?

The checksum that represents your built project (whether this is generated by the MPLAB X IDE or by tools such as HEXMATE) is calculated from the generated output of the compiler. Indeed, the algorithms used to obtain the checksum are specifically designed so that even small changes in this output are almost guaranteed to produce a different checksum result. Checksums are not calculated from your project's source code, so to ensure that your checksum does not change from build to build, you must ensure that the output of the compiler does not change.

The following actions and situations could cause changes in the compiled output and hence changes in your project's checksum.

- Changing the source code, header files, or library code used by the project between builds
- Changing the order in which source files or libraries are compiled or linked between builds
- Having source code that makes use of macros such as `__DATE__` and `__TIME__`, which produce output that is dependent on when the project was built
- Moving the location of source files between builds, where those files use macros such as `__FILE__`, which produces output that is dependent on where the source file is located
- Changing the compiler options between builds
- Changing the compiler version between builds

Also the checksum algorithms used by tools such as HEXMATE and the MPLAB X IDE can change, which can result in a different checksum for the same compiler output. Such changes are rare, but check the compiler and IDE release notes to see if the tools have been modified.

5.6 Fixing Code That Does Not Work

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

- [5.6.1 How Do I Find Out What a Warning or Error Message Means?](#)
- [5.6.2 How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?](#)
- [5.6.3 How Can I Stop Warnings from Being Produced?](#)
- [5.6.4 How Do I Know If the Stack Has Overflowed?](#)
- [5.6.5 What Can Cause Corrupted Variables and Code Failure When Using Interrupts?](#)
- See also, [5.2 Invoking the Compiler](#)
- See also, [5.5.15 How Do I Properly Reserve Memory?](#)

5.6.1 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:

[25. Diagnostics](#)

[7.6.4 Options for Controlling Warnings and Errors](#)

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

In most instances the message produced by the compiler indicates the offending line of code where the syntax error is relating to the source 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.

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" (DS50002106)

5.6.3 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:

[7.6.4 Options for Controlling Warnings and Errors](#)

5.6.4 How Do I Know If the Stack Has Overflowed?

The 16-bit devices use a stack that's 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 of this guide:

[12.2.3 Auto Variable Allocation and Access](#), "Software Stack"

[12.2.3 Auto Variable Allocation and Access](#), "The C Stack Usage"

See the software stack in your device data sheet.

5.6.5 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 of this user's guide:

[5.4.3 How Do I Share Data Between Interrupt and Main-line Code?](#)

6. XC16 Toolchain and MPLAB X IDE

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.

6.1 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 license levels) or for purchase (PRO license level) 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

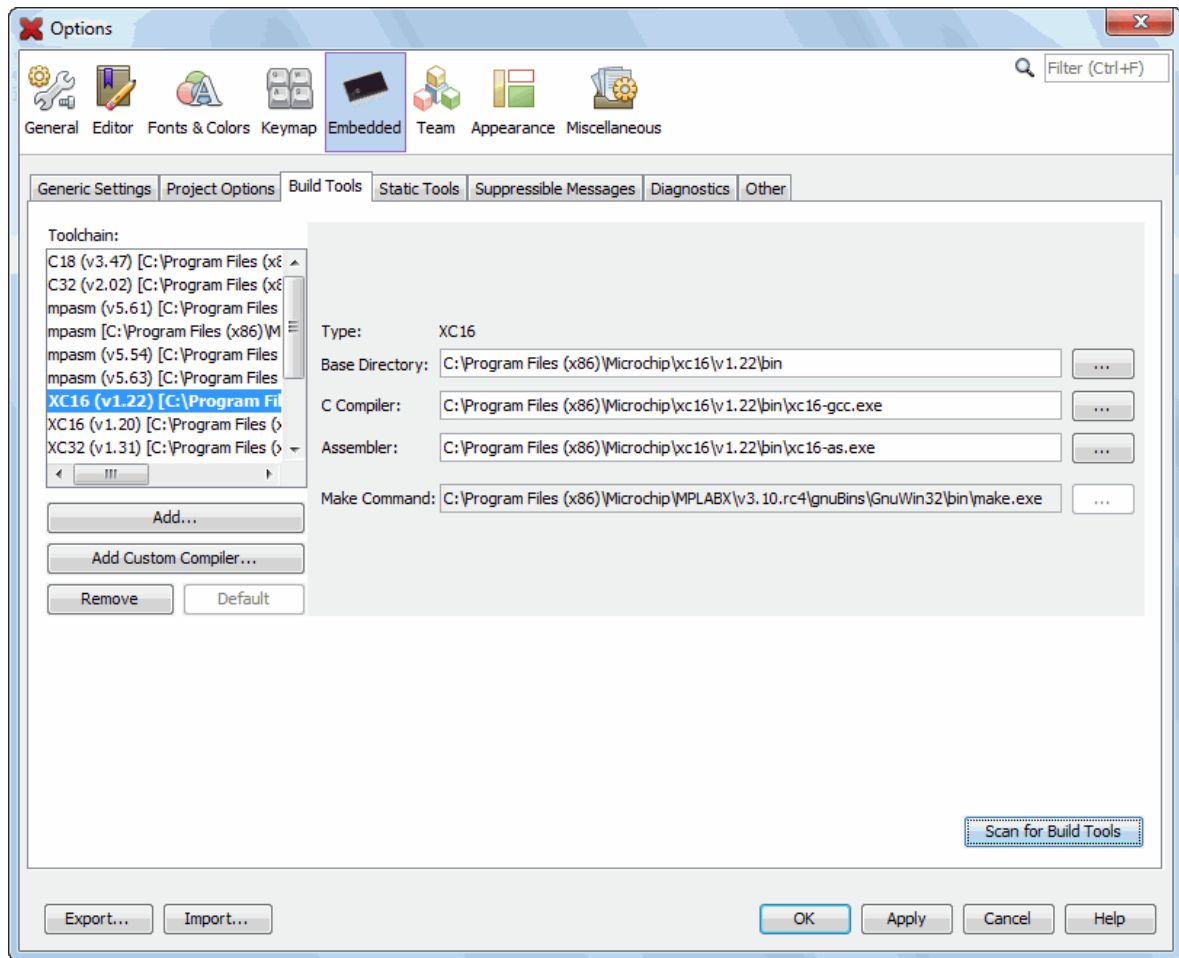
Device support files may be found under the `support` subdirectory. The generic `xc.h` C header file and `xc.inc` generic assembler include file may be found under the `support/generic` subdirectory.

6.2 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 the **OK** button.

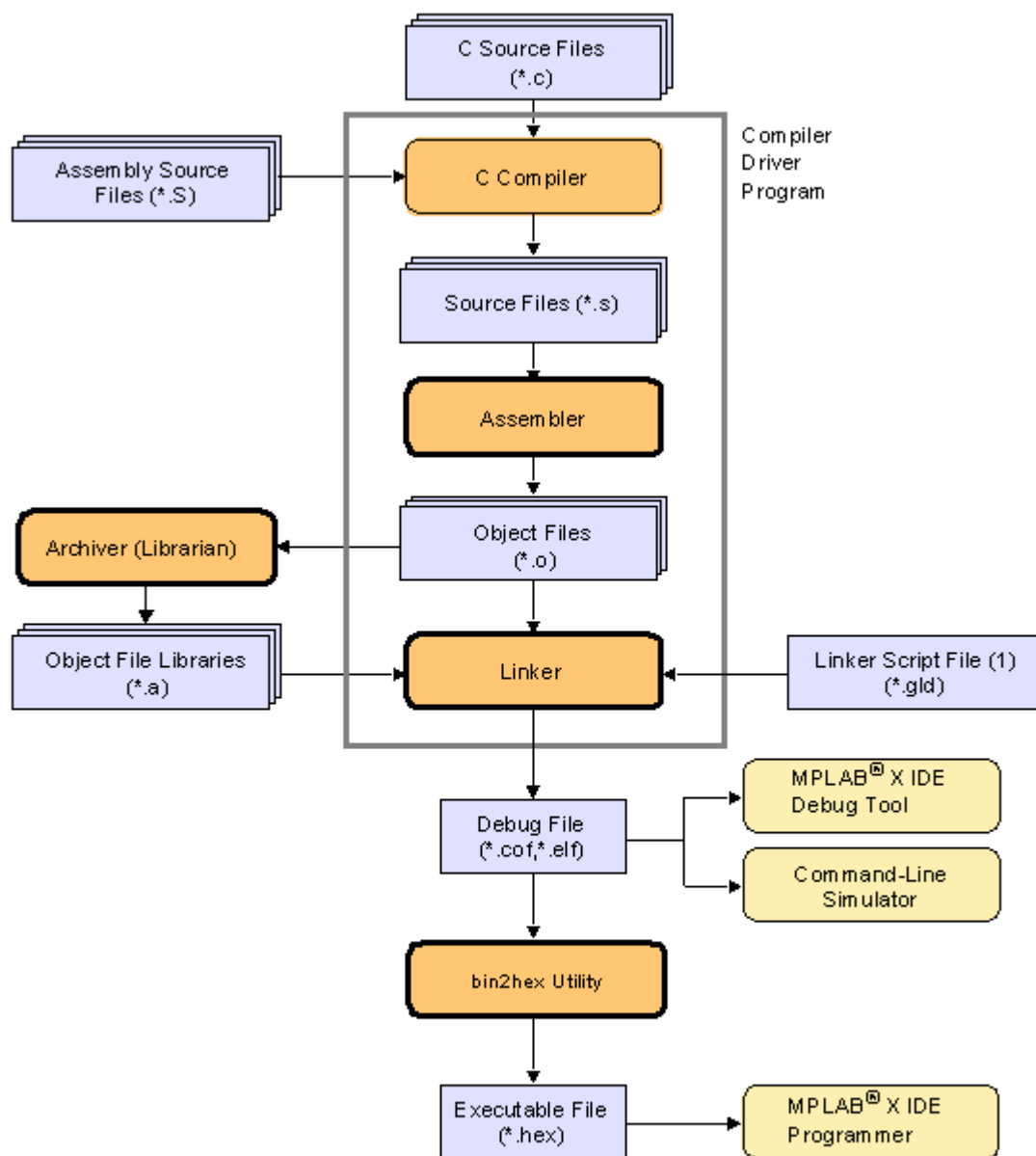
Figure 6-1. XC16 Suite Tool Locations In Windows OS



6.3 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 6-2. Compiler Project Relationships



(1) The linker can choose the correct linker script file for your project.

6.4 Operation Summary

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.

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.

Object files can be archived into a library using the archiver/librarian.

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).

6.5 References

For more information on compiler operation refer to the following points of information:

- [7. Compiler Command-Line Driver](#).
- *MPLAB® X IDE User's Guide* (DS50002027),
"Basic Tasks," "Create a New Project."
- *MPLAB® XC16 Assembler, Linker and Utilities User's Guide* (DS50002106).

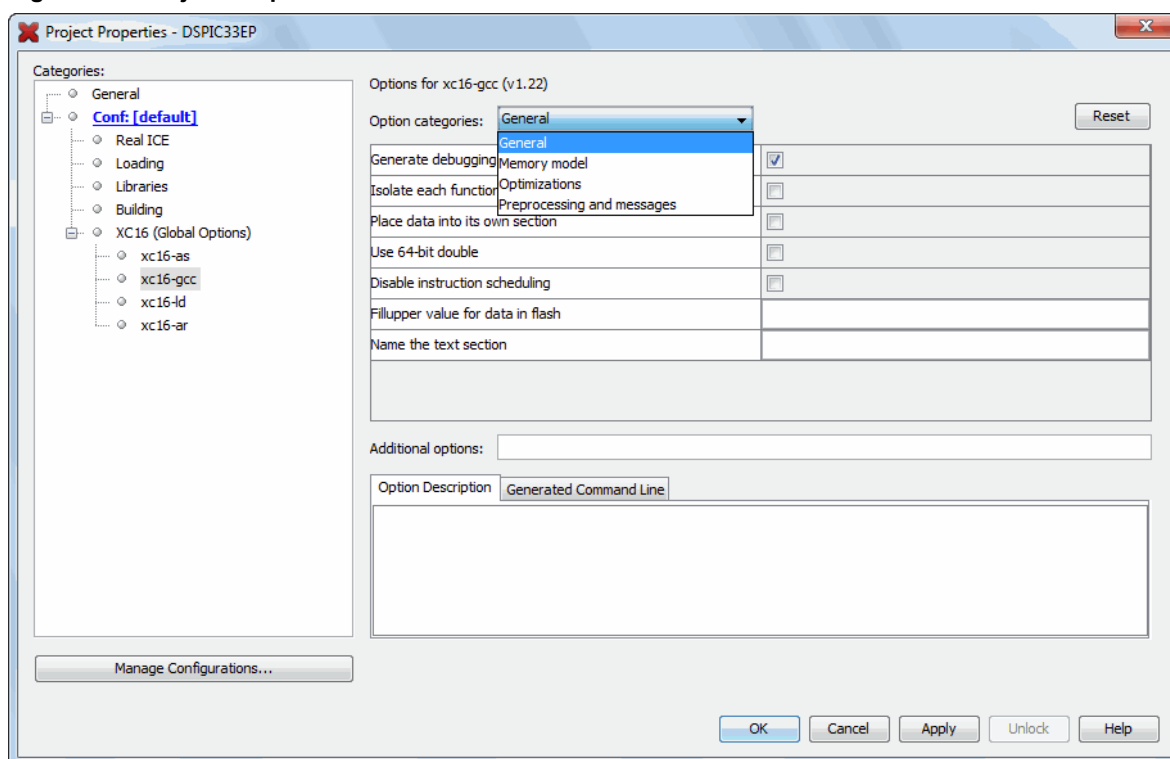
6.6 Project Setup

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 (see figure below).

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.
 - [6.6.1 XC16 \(Global Options\)](#)
 - [6.6.2 xc16-as \(16-bit Assembler\)](#)
 - [6.6.3 xc16-gcc \(16-bit C Compiler\)](#)
 - [6.6.4 xc16-ld \(16-Bit Linker\)](#)
 - [6.6.5 xc16-ar \(16-Bit Archiver/Librarian\)](#)

Figure 6-3. Project Properties Window



6.6.1 XC16 (Global Options)

Set up global options for all 16-bit language tools. See also “6.6.6 Options Page Features”.

Table 6-1. 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 libc	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
Relaxed floating point math	Check to use relaxed-compliance math library. This is a math library that follows slightly different rules than those the IEEE standard dictates for infinities, NaNs, and denormal (tiny) numbers. Uncheck to use standard floating point math library.	-relaxed-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

.....continued

Option	Description	Command Line
Common include dirs	Directory paths entered here will be appended to the already existing include paths of the compiler. Relative paths are from the MPLAB X IDE project directory.	-I dir

6.6.2 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 the *MPLAB® XC16 Assembler, Linker and Utilities User's Guide* (DS50002106). See also, "6.6.6 Options Page Features."

Table 6-2. 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 <code>.include</code> directives. For more information, see 6.6.7 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 <code>.include</code> directives. For more information, see 6.6.7 Additional Search Paths and Directories	-I"dir"
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 <code>.L</code> (upper case only). Uncheck to discard local symbols.	--keep-locals (-L)
Diagnostics level	Select warnings to display in the Output window.	
	- Generate warnings	--warn
	- Suppress warnings	--no-warn
	- Fatal warnings	--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 6-3. 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 <code>-g</code> is given to the compiler, and assembly listings (<code>-al</code>) are requested. Uncheck for a regular listing.	-ah

.....continued		
Option	Description	Command Line
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 .sbt1. 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
List to file	Check to send listing information to a file. Uncheck to send listing information to the Output window.	-a=asmfilename.lst

6.6.3 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 [7.6 Driver Option Descriptions](#).

See also [6.6.6 Options Page Features](#).

Table 6-4. General Category

Option	Description	Command Line
Generate debugging info	Create a COFF or ELF file with information to allow debugging of code in MPLAB X IDE. Note: COFF supports debugging in the .text section only.	-g

.....continued

Option	Description	Command Line
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.	-ffunction-sections
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.	-fdata-sections
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	-fno-short-double
Fill upper value for data in flash	Fill upper flash memory with the value specified.	-mfillupper=value
Name the text section	Place text (program code) in a section named <code>name</code> rather than the default <code>.text</code> section.	-mtext=name

Table 6-5. Memory Model Category

Option	Description	Command Line
Code model	Select a code (program memory/ROM) model. - default - large (>32 K words) - small (\leq 32 K words)	-msmall-code -mlarge-code -msmall-code
Data model	Select a data (data memory/RAM) model. - default - large (device dependent ¹) - small (device dependent ¹)	device dependant ² -mlarge-data -msmall-scalar
Scalar model	Select a scalar model. - default - large (device dependent ¹) - small (device dependent ¹)	-msmall-scalar -mlarge-scalar -msmall-scalar
Location of constant model	Select a memory location for constants. - default - Data - Code	-mconst-in-code -mconst-in-data -mconst-in-code

.....continued

Option	Description	Command Line
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.	-mauxflash
Put constants into auxiliary flash	When combined with -mconst-in-code, put constants into auxiliary Flash.	-mconst-in-auxflash
Allow arrays larger than 32K	Allow arrays that are larger than 32K, regardless of memory model.	-menable-large-arrays
Aggregate data model	Use aggregate data model.	-mlarge-aggregate

Note:

1. For most devices 6K of RAM is the near data space, but for some devices it is 4K of RAM.
2. For devices that have all of their data memory in the near space, the memory model is "small data" "small scalar" so that all memory will be allocated in the near space.
For all other devices the default memory model is "large data" "small scalar". This will have the effect of allowing the tool chain to place aggregate objects, such as arrays and structure, into the far memory space. This can be over-ridden by explicitly selecting "small data" in the compiler options.

Table 6-6. OPTIMIZATION CATEGORY

Option	Description	Command Line
Optimization Level	Select an optimization level. Equivalent to <code>-On</code> option, where <i>n</i> is an option. See 7.6.6 Options for Controlling Optimization . Your compiler license may support only some optimizations. See Chapter 18. "Optimizations."	<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=n</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=n</code>

.....continued

Option	Description	Command Line
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.	-falign-arrays

Table 6-7. Preprocessing and Messages Category

Option	Description	Command Line
Include C dirs	Add the directory <code>dir</code> to the head of the list of directories to be searched for header files. For more information, see 6.6.7 Additional Search Paths and Directories	-I"dir"
Define C macros	Define macro <code>macro</code> with the string 1 as its definition.	-Dmacro
ANSI-std C support	Check to support all (and only) ASCII C programs. Uncheck to support ASCII and non-ASCII programs.	-ansi
Use CCI syntax	Check if your code is written per the Common C Interface (CCI) syntax (see Chapter 2. "Common C Interface."). Uncheck if you are not.	-mcci
Use IAR syntax	Check if your code is written per the Embedded Compiler Compatibility Mode syntax for IAR (see Appendix B. "Embedded Compiler Compatibility Mode"). Uncheck if you are not.	-mext=IAR
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.	-merrata=id
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 <code>n</code> 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. 	-msmart-io=n
Smart IO format 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.	-Werror
Additional warnings	Check to enable all warnings. Uncheck to disable warnings.	-Wall

.....continued

Option	Description	Command Line
Strict ANSI warnings	Check to issue all warnings demanded by strict ANSI C. Uncheck to issue all warnings.	-pedantic
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.	-mno-isr-warn
Enable SFR warnings	Enable warnings related to SFRs.	-msfr-warn=on off

6.6.4 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 the *MPLAB® XC16 Assembler, Linker and Utilities User's Guide (DS50002106)*. See also ["Options Page Features."](#)

Table 6-8. GENERAL CATEGORY

Option	Description	Command Line
Heap size	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.	--heap size
Min stack size	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>__SP_LIM_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.	--stack size
Use local stack	Check to prevent allocating the stack in extended data space memory. Uncheck to allow allocating the stack in extended data space memory.	--local-stack --no-local-stack
Allow overlapped sections	Check to not check section addresses for overlaps. Uncheck to check for overlaps.	--check-sections --no-check-sections
Init data sections	Check to support initialized data. Uncheck to not support.	--data-init --no-data-init
Pack data template	Check to pack initial data values. Uncheck to not pack.	--pack-data --no-pack-data
Create handles	Check to support far code pointers. Uncheck to not support.	--handles --no-handles
Create default ISR	Check to create an interrupt function for unused vectors. Uncheck to not create a default ISR.	--isr --no-isr

.....continued		
Option	Description	Command Line
Remove unused sections	Check to not enable garbage collection of unused input sections (on some targets). Uncheck to enable garbage collection.	--no-gc-sections --gc-sections
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.	--fill-upper=value
Stack guardband size	Enter a stack guardband size to ensure that enough stack space is available to process a stack overflow exception.	--stackguard=size
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.	
Use response file to link	Check to create a makefile that uses a response file for the link step. In Windows, you have a maximum command line length of 8191 chars. When linking long programs, the link line might go over this limit. MPLAB XC16 provides a response file work-around. See MPLAB X IDE documentation, Troubleshooting section, for details. Uncheck to not use a response file.	

Table 6-9. Symbols and Macros Category

Option	Description	Command Line
Linker symbols	Create a global symbol in the output file containing the absolute address (<code>expr</code>). 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 <code>expr</code> in this context: you may give a hexadecimal constant or the name of an existing symbol, or use <code>+</code> and <code>-</code> to add or subtract hexadecimal constants or symbols.	--defsym=sym
Define Linker macros	Add linker macros.	-Dmacro
Symbols	Specify symbol information in the output.	
	- Keep all	—
	- Strip debugging info	--strip-debug (-S)
	- Strip all symbol info	--strip-all (-s)

Table 6-10. Fill Flash Memory Category

Option	Description	Command Line
Which areas to fill	Specify which area of Flash memory to fill. No Fill - None (default). Fill All Unused - Fill all unused memory. Provide Range to fill - Fill a range of memory. Enter a range under "Memory Address Range".	
How to fill it	Specify how to fill Flash memory. Provide sequence of values - Provide a sequence under the Sequence option. Constant or incrementing value - Provide either: <ul style="list-style-type: none"> Constant = a value, Increment/Decrement = No Incrementing Constant = a value, Increment/Decrement = Increment Const OR Decrement Const, Increment/Decrement Constant = a value 	
Sequence	When Provide sequence of values is selected, enter a sequence. The form is n1, n2, where n1 uses C syntax. Example: 0x10, 25, 0x3F, 16.	--fill=sequence
Constant	When Constant or incrementing value is selected, enter a constant. Specify the constant using C syntax (e.g., 0x for hex, 0 for octal). Example: 0x10 is the same as 020 or 16.	--fill=constant
Increment/Decrement	When Constant or incrementing value is selected, you may select to increment or decrement the initial value of "Constant" on each consecutive address. No Incrementing - Do not change constant value. Increment Const - Increment the constant value by the amount specified under the option "Increment/Decrement Constant." Decrement Const - Decrement the constant value by the amount specified under the option "Increment/Decrement Constant."	
Increment/Decrement Constant	When Increment Const or Decrement Const is selected, enter a constant increment or decrement value. Specify the constant using C syntax (e.g., 0x for hex, 0 for octal). Example: 0x10 is the same as 020 or 16.	-- fill=constant+=in cr --fill=constant- =decr
Memory Address Range	When Provide Range to fill is selected, enter the range here. Specify range as Start:End where Start and End use C syntax. Example 0x100:0x1FF is the same as 256:511	-- fill=value@range

Table 6-11. Libraries Category

Option	Description	Command Line
Libraries	Add libraries to be linked with the project files. You may add more than one.	--library=name

.....continued

Option	Description	Command Line
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 6-12. 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>
Warn on section realignment	Check to warn if start of section changes due to alignment. Uncheck to not warn.	<code>--warn-section-align</code>
Trace Symbols	Add/remove trace symbols.	<code>--trace-symbol=symbol</code>

Table 6-13. Code Guard Category

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

.....continued		
Option	Description	Command Line
Secure RAM	Specify the secure RAM segment: none, small, medium or large.	--secure=option_ram
Secure Flash	Specify the secure Flash segment: none, small, medium, or large standard or none, small, medium, or large high.	--secure=option_flash_std --secure=option_flash_high
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 MPLAB® XC16 Assembler, Linker and Utilities User’s Guide (DS50002106). Note: Not all development tools support CodeGuard programming. See tool documentation for more information.		

6.6.5 xc16-ar (16-Bit Archiver/Librarian)

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 the *MPLAB® XC16 Assembler, Linker and Utilities User’s Guide* (DS50002106). See also, [6.6.6 Options Page Features](#).

Table 6-14. General Category

Option	Description	Command Line
Break line into multiple lines	For Windows OS, you have a maximum command line length of 8191 chars. When archiving long sets of files into libraries, the link line might go over this limit. The compiler can break up the archive line into smaller lines to avoid this limitation.	true

6.6.6 Options Page Features

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

Table 6-15. 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.

6.6.7 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.

6.7 Project Example

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

- [6.7.1 Run the Project Wizard](#)
- [6.7.2 Add a File to the Project](#)
- [6.7.3 Build and Run the Project](#)
- [6.7.4 Output Files](#)
- [6.7.5 Further Development](#)

6.7.1 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 “Standalone Project” for the project. Click **Next>** to continue.
2. **Select Device:** Select PIC24FJ128GA010. Click **Next>** to continue.
3. **Select Header:** Select a header for this device if you are using one. Otherwise leave as “None.” Click **Next>** to continue.
4. **Select Tool:** Choose a development tool from the list, which for this example is the Simulator. 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 Plugin Board:** Select a plugin board if you are using one. Otherwise leave as “None.” Click **Next>** to continue.
6. **Select Compiler:** Choose a version of the XC16 toolchain installed on your PC. Click **Next>** to continue.
7. **Select Project Name and Folder:** Enter a project name, for this example `XC16_Example`. 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.

6.7.2 Add a File to the Project

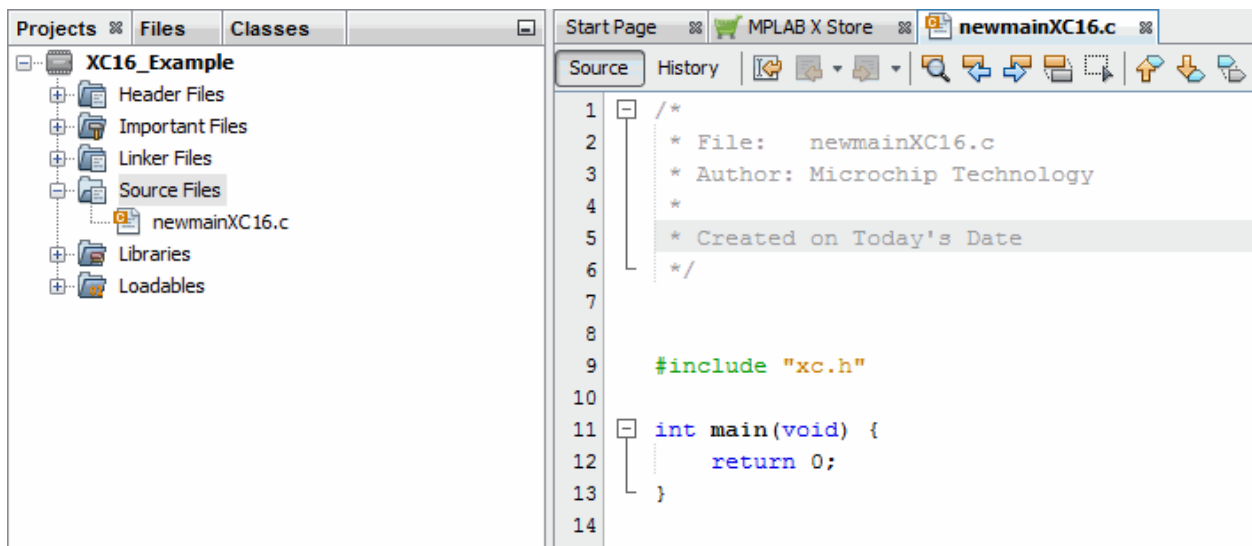
To add a C code template file to the project:

1. Right click on the “Source Files” folder in the project tree. Select *New>mainXC16.c* to open the “New mainXC16.c” dialog.
2. Enter a name for the new file. The default is `newmainXC16.c`.
3. Enter a folder in which to place the file. The default is the project folder. Keeping the file in the project folder makes the project more portable.
4. Click **Finish**.

The project tree should now have the Source Files folder open, containing the file added, as well as the new file open in an Editor window (see following figure).

Note: If you add more than one file, the order in which you add these files to the project is the order in which they will be linked.



Figure 6-4. Project Tree and Source Code



6.7.3 Build and Run the Project

To set up build options, you can select *File>Project Properties* or right click on the project name and select “Properties” to open the Project Properties dialog. For this example, default options will be used so no additional set up is required.

Click the **Debug Main Project** icon to build the code, program the target device (if a hardware tool is selected) and start the debug session. Because of the example code used, the application will run and then stop. To finish execution, click the **Finish Debugger Session** icon.

	Debug Main Project Icon
	Finish Debugger Session Icon

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

1. Review the previous steps in this example. Make sure you have installed and set up the MPLAB XC16 C compiler so that MPLAB X IDE can see it. See [6.2 MPLAB X IDE Setup](#).
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 the build again.

6.7.4 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. For example, in Windows 7 OS:
C:\Users\UserName\MPLABXProjects\XC16_Example.X
2. Under the project directory, locate the linker map file. For the example above:
C:\Users\UserName\MPLABXProjects\XC16_Example.X\
dist\default\debug\XC16_Example.X.debug.map
3. View the linker map file in an MPLAB X IDE editor window. For more on this file, see the linker documentation.
4. In the same directory there is another file, XC16_Example.X.debug.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 [6.6.1 XC16 \(Global Options\)](#).

6.7.5 Further Development

In addition to the MPLAB X Simulator used in this example, several other debug tools exist that work with MPLAB X IDE. You may choose from in-circuit emulators or in-circuit debuggers, manufactured by Microchip Technology or third-party developers. Please see the documentation for these tools to learn how they can help you.

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, use the **Make and Program Device Project** button on the debug toolbar. Please see MPLAB X IDE documentation concerning this control.

7. Compiler Command-Line Driver

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. Compiler options are selected in the GUI and passed to the compiler driver for execution.

7.1 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.

7.1.1 Drive Command-Line Format

The basic form of the compiler command line is:

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

where:

options: See [7.6 Driver Option Descriptions](#) for available options.

files: See [7.1.3 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 [7.1.2 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 -T p30f2010.gld -o hello.elf hello.c
```

7.1.2 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 7-1. Compiler-Related Environmental Variables

Variable	Definition
XC16_C_INCLUDE_PATH PIC30_C_INCLUDE_PATH	<p>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.</p> <p>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 the following include files:</p> <p><install-path>\include and <install-path>\support\h</p>
XC16_COMPILER_PATH PIC30_COMPILER_PATH	<p>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>.</p>
XC16_EXEC_PREFIX PIC30_EXEC_PREFIX	<p>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.</p> <p>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.</p> <p>Other prefixes specified with the <code>-B</code> command line option take precedence over the user- or driver-defined value of the variable.</p> <p>Under normal circumstances it is best to leave this value undefined and let the driver locate subprograms itself.</p>
XC16_LIBRARY_PATH PIC30_LIBRARY_PATH	<p>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:</p> <p><install-path>\lib; <install-path>\support\gld.</p>
XC16_OMF PIC30_OMF	<p>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.</p>
TMPDIR	<p>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.</p>

7.1.3 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 7-2. File Names

Extensions	Definition
file.c	A C source file that must be preprocessed.
file.h	A header file (not to be compiled or linked).

.....continued	
Extensions	Definition
file.i	A C source file that should not be preprocessed.
file.o	An object file.
file.p	A pre procedural-abstraction assembly language file.
file.s	Assembler code.
file.S	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.

7.2 The Compilation Sequence

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

7.2.1 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, 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 the [6.3 MPLAB X IDE Projects](#) section.

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.

The following table 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 7-3. Compiler Application Names

Name	Description
xc16-gcc	Command line driver; the interface to the compiler
xc16-as	Assembler (based on the target device)
xc16-ld	Linker
xc16-bin2hex	Conversion utility to create HEX files

.....continued	
Name	Description
xc16-strings	String extractor utility
xc16-strip	Symbol stripper utility
xc16-nm	Symbol list utility
xc16-ar	Archiver/Librarian
xc16-objdump	Object file display utility
xc16-ranlib	Archive indexer utility

7.2.2 Single-Step Compilation

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

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 the [7.1.2 Environment Variables](#) section) 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 the [8.2 Device Header Files](#) section.

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

```
xc16-gcc -mcpu=30f2010 -T p30f2010.gld -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.

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`.

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 -T p30f2010.gld -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.

7.2.3 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* (DS50002106) 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* (DS50002106).

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`.

7.2.4 Assembly Compilation

A mix of C and assembly code can be compiled together using the compiler ([6.3 MPLAB X IDE Projects](#)). For more details, see the [18. Mixing C and Assembly Code](#) section.

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.

7.3 Runtime Files

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

7.3.1 Library Files

The compiler may include library files into the output per [Figure 6-2](#).

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

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 is described in the “16-Bit Language Tool Libraries” (DS51456).

User-Defined Libraries

You may create your 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* (DS50002106).

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 7.6.9 Options for Linking](#).

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

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

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.

User-Defined Libraries Development

When creating your own libraries, follow the guidelines listed below.

Library and Supporting Files

No library file should contain a `main()` function, nor settings for configuration bits or any other such data.

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.

OMF Libraries

MPLAB XC16 supports two object file formats, often called OMF for object module format. COFF is an older standard and is not recommended. ELF, combined with its debugging format DWARF, produces executables that contain a richer language for describing the artifacts of the executable program from a debugging perspective.

Should you wish to produce generic libraries that are COFF and ELF compatible, we recommend that each library be separated and named `liblibrary-elf.a` and `liblibrary-coff.a`. Each library, of course, should contain objects built for the appropriate OMF. Naming the libraries in this way will allow the linker to choose a correct library from the standard library inclusion option and the current OMF. In other words, `-llibrary` will match first against `liblibrary.a` followed by `liblibrary-OMF.a`. This makes it easier to switch between COFF and ELF.

Device Specific and Generic Libraries

If you would like to produce a library that will be compatible with a range of 16-bit devices, you may need to include more than one copy of each object file in the library. This is perfectly acceptable, as long as each copy has a unique name. The linker will reject object files that do not match the characteristics of the user selected device.

Consider a simple library that contains one file (and one function) name `hello_world.c`; you can guess at its use. The desire of this function is to work on a range of devices, for example: dsPIC30F6014, dsPIC33EP512MU810 and PIC24F16KA302. Compile `hello_world.c` once for each device and combine them into one library:

```
xc16-gcc -O1 -c hello_world.c -mcpu=30F6014 -o hello_world.30f.o
xc16-gcc -O1 -c hello_world.c -mcpu=33EP512MU810 -o hello_world.33ep.o
xc16-gcc -O1 -c hello_world.c -mcpu=24F16KA302 -o hello_world.24f.o
xc16-ar crv libhello_world-elf.a hello_world.30f.o hello_world.33ep.o hello_world.24f.o
```

This would produce a library that can be linked against any one of those devices.

```
xc16-gcc -O1 test.c -mcpu=30F6014 -o test.exe -L. -lhello_world
```

If a library is required to link against any device, the use of a set of generic device names, listed in [Readme_XC16.html](#) or acquired directly from the tool using the compiler option `-mprint-devices`, will produce object files that will link against any device.

7.3.2 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 [17.2 Runtime Startup and Initialization](#)

7.4 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.

7.4.1 Output Files

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

Table 7-4. 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 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 7.6 Driver Option Descriptions](#).

7.4.2 Diagnostic Files

Two valuable files produced by the compiler are:

- The assembly list file, produced by the assembler.
- 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* (DS50002106). To pass the option from the compiler, see [section 7.6.8 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 7.2.3 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* (DS50002106). To specify the option from the command-line driver, see [section 7.6.9 Options for Linking](#).

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

7.5 Compiler Messages

Compiler output messages for errors, warnings, or comments are discussed in the [25. Diagnostics](#) section.

For information on options that control compiler output of errors, warnings, or comments, see [section 7.6.4 Options for Controlling Warnings and Errors](#).

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

7.6 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 either [section 7.6.8 Options for Assembling](#) or [section 7.6.9 Options for Linking](#) for information on specifying options for these tools to the compiler.

7.6.1 Options Specific to 16-Bit Devices

For more information on the memory models, see [section 12.14 Memory Models](#).

Table 7-5. 16-Bit Device-Specific Options

Option	Definition
<code>-mcodecov</code>	This option is used for MPLAB® Code Coverage support. Passing the option <code>-mcodecov=near</code> or <code>-mcodecov=far</code> to the compiler causes it to instrument the generated assembly-instruction blocks with information that the MPLAB X IDE can then use to perform a code coverage analysis. This feature requires MPLAB X IDE v5.25 or later. For more on this features, visit: www.microchip.com/mplab/codecoverage
<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 all <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>-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 21.3 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>-mno-eds-warn</code>	On some devices, there is a possibility that the stack will reside in EDS (extended data space) memory (above 0x8000), though this allocation is disabled by default in the linker. If the stack is located in this area, then taking the address of an auto variable would require an <code>__eds__</code> pointer. As the compiler does not know where the stack will be located, the default is to be conservative and warn if the address of an auto is taken and not used as an <code>__eds__</code> pointer. This option disables the warning.
<code>-merrata=id[,id]*</code>	This option enables specific errata workarounds 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 workarounds.
<code>-mno-errata=id[,id]*</code>	This option disables specific errata workarounds identified by <code>id</code> . Valid values for <code>id</code> change from time to time. This is particularly useful when specifying errata with <code>-merrata=all</code> as it can be used to disable some errata that are not required. <code>-mno-errata=foo</code> will prevent the erratum <code>foo</code> from being enabled no matter where it appears on the command line. Therefore, <code>-mno-errata=foo -merrata=foo</code> will not enable erratum <code>foo</code> .
<code>-mno-file</code>	Do not emit a <code>.file</code> directive in the generated assembly file. This is useful when creating libraries where the source code may not reside on the end-user's machine, as this will prevent the IDE from trying to load the source file during a debug session.
<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.
<code>-mlarge-arrays</code>	Specifies that arrays may be greater than or equal to the default maximum size of 32K. See 12.2.2 Non-Auto Variable Allocation and Access , Non-Auto Variable Size Limits 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.

.....continued	
Option	Definition
-mlegacy-libc	MPLAB XC16 (originally MPLAB C30) has a long history. This option allows us to support previously deployed C libraries as needed (This is the default).
-moptimize-page -setting	Attempt to reduce the number page switches when using memory modes that affect the PSVPAG. This is really an optimization, and is not enabled by default. Like all optimizations it will generally have a positive effect on performance or code size.
-mpa ²	Enable the procedure abstraction optimization. There is no limit on the nesting level. Optimization levels depend on the compiler edition (see section 20. Optimizations).
-mpa=n ¹ .	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 20. Optimizations).
-mno-pa ¹ .	Do not enable the procedure abstraction optimization (This is the default).
-mpreserve-all	Make all variables in this translation unit preserved unless explicitly marked with the update attribute.
-mprint- builtins	Display the complete list of target builtin functions available in the compiler.
-mprint-devices	Display the complete list of real and virtual devices supported by the current installation.
-mprint-mchp- search-dirs	A target-specific option to output the compiler and assembler include search paths to the console. These paths can change based upon various options.
-mno-isr-warn	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.
-omf	Selects the OMF (Object Module Format) to be used by the compiler. The <code>omf</code> 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.
-msfr-warn	By default we warn when accessing SFRs for a generic device; use <code>-mno-sfr-warn</code> to disable this feature.
-msmall-code	Compile using the small code model. Called functions are assumed to be proximate (within 32 Kwords of the caller). (This is the default.)

² 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.

.....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>-msmart-io-format=fmt</code>	When using smart-io the compiler is not able to detect the format string when it is a variable. <code>-msmart-io-format</code> can be used to tell the compiler which format specifiers to expect in such a string. For example: <pre>printf(stderr,fmt,a,b,c);</pre> can be compiled with <code>-msmart-io-fmt="%s%c%d"</code> to define the default format and smart-io will generate code to match this set of format specifiers when it cannot determine the correct format specifiers at runtime.
<code>-mtext=name</code>	Specifying <code>-mtext=name</code> will cause text (program code) to be placed in a section named <code>name</code> 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 [=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.
<code>--partition n</code>	This option targets a single partition <code>n</code> in a dual partition device and will constrain the output text to be contained within one panel.

7.6.2 Options for Controlling the Kind of Output

The following options control the kind of output produced by the compiler.

Table 7-6. 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 <code>file</code> .
<code>-S</code>	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is <code>.s</code> .
<code>--help</code>	Print a description of the command-line options.

7.6.3 Options for Controlling the C Dialect

The following options define the kind of C dialect used by the compiler.

Table 7-7. C Dialect Control Options

Option	Definition
<code>-ansi</code>	Support all (and only) ANSI-standard C programs.

.....continued	
Option	Definition
<code>-aux-info filename</code>	Output to the given file name prototype 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.
<code>-menable-fixed</code> <code>[=<i>rounding mode</i>]</code>	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 .
<code>-ffreestanding</code>	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 <code>main</code> . The most obvious example is an OS kernel. This is equivalent to <code>-fno-hosted</code> .
<code>-fno-asm</code>	Will 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> .
<code>-fno-builtin</code> <code>-fno-builtin-function</code>	Will not recognize built-in functions that do not begin with <code>__builtin_</code> as prefix.
<code>-fsigned-char</code>	Let the type <code>char</code> be signed, like <code>signed char</code> . (This is the default.)
<code>-fsigned-bitfields</code> <code>-funsigned-bitfields</code> <code>-fno-signed-bitfields</code> <code>-fno-unsigned-bitfields</code>	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.
<code>-funsigned-char</code>	Let the type <code>char</code> be unsigned, like <code>unsigned char</code> .

7.6.4 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.

7.6.4.1 Options to Control the Amount and Types of Warnings

The following options control the amount and kinds of warnings produced by the compiler.

Table 7-8. Warning/Error Options Implied by `-Wall`

Option	Definition
<code>-fsyntax-only</code>	Check the code for syntax, but don't do anything beyond that.
<code>-w</code>	Inhibit all warning messages.

.....continued

Option	Definition
-Wall	All of the -W 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> .
-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 <code>char</code> 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 <code>char</code> 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> .

.....continued

Option	Definition
-Wsequence-point	<p>Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard.</p> <p>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.</p> <p>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.</p> <p>Examples of code with undefined behavior are <code>a = a++; 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.</p>
-Wswitch	<p>Warn whenever a <code>switch</code> statement has an index of enumerual type and lacks a case 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.</p>
-Wsystem-headers	<p>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.</p>
-Wtrigraphs	<p>Warn if any trigraphs are encountered (assuming they are enabled).</p>
-Wuninitialized	<p>Warn if an automatic variable is used without first being initialized.</p> <p>These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing.</p> <p>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.</p> <p>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.</p>

.....continued	
Option	Definition
-Wunknown-pragmas	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.
-Wunused	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.
-Wunused-function	Warn whenever a static function is declared but not defined or a non-inline static function is unused.
-Wunused-label	Warn whenever a label is declared but not used. To suppress this warning, use the <code>unused</code> attribute (see section 10.10 Variable Attributes).
-Wunused-variable	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 10.10 Variable Attributes).
-Wunused-value	Warn whenever a statement computes a result that is explicitly not used. To suppress this warning, cast the expression to <code>void</code> .

7.6.4.2 Options That Are Not Implied by `-Wall`

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 7-9. Warning/Error Options not Implied by `-Wall`

Option	Definition
-Wextra, -W	Print extra warning messages for specific events. For details, see 7.6.4.3 The -W Option section.
-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> .

.....continued	
Option	Definition
-Wconversion	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> .
-Werror	Make all warnings into errors.
-Winline	Warn if a function can not be inlined and was either declared as inline, or else the <code>-finline-functions</code> option was given.
-Wlarger-than=len	Warn whenever an object of larger than <code>len</code> bytes is defined.
-Wlong-long -Wno-long-long	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.
-Wmissing-declarations	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.
-Wmissing-format-attribute	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.
-Wmissing-noreturn	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.
-Wmissing-prototypes	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.)
-Wnested-externs	Warn if an <code>extern</code> declaration is encountered within a function.
-Wno-deprecated-declarations	Do not warn about uses of functions, variables and types marked as deprecated by using the <code>deprecated</code> attribute.
-Wpadded	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.
-Wpointer-arith	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.
-Wredundant-decls	Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.
-Wshadow	Warn whenever a local variable shadows another local variable.
-Wsign-compare -Wno-sign-compare	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> .

.....continued	
Option	Definition
-Wstrict-prototypes	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.)
-Wtraditional	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.
-Wundef	Warn if an undefined identifier is evaluated in an <code>#if</code> directive.
-Wwrite-strings	Give string constants the type <code>const char[length]</code> so that copying the address of one into a non-const <code>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.

7.6.4.3 The -w Option

Use the `-w` command line option to print extra warning messages for these events:

- A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings are possible only in optimizing compilation. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` 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 `longjmp` cannot in fact be called at the place that would cause a problem.
- A function could exit both via `return value;` and `return;`. Completing the function body without passing any return statement is treated as `return;`.
- 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 void. For example, an expression such as `x[i,j]` will cause a warning, but `x[(void)i,j]` will not.
- An unsigned value is compared against zero with `<` or `<=`.
- A comparison like `x<=y<=z` appears; this is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of ordinary mathematical notation.
- Storage-class specifiers like `static` are not the first things in a declaration. According to the C Standard, this usage is obsolescent.
- If `-Wall` or `-Wunused` 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 `-Wno-sign-compare` 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 `x.h`:

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

- An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because `x.h` would be implicitly initialized to zero:

```
struct s { int f, g, h; };
```

```
struct s x = { 3, 4 };
```

7.6.5 Options for Debugging

The following options are used for debugging.

Table 7-10. 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.
<code>-save-temps</code>	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)

7.6.6 Options for Controlling Optimization

The following options control compiler optimizations. Optimization levels available depend on the compiler license (see the [20. Optimizations](#) section).

Table 7-11. General Optimization Options

Option	License	Definition
<code>-O0</code>	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.
<code>-O</code> <code>-O1</code>	All	Optimize for both speed and size. 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. The compiler turns on <code>-fthread-jumps</code> and <code>-fdefer-pop</code> and turns on <code>-fomit-frame-pointer</code> .

.....continued		
Option	License	Definition
-O2	PRO	Optimize more for speed. -O2 turns on all optional optimizations except for loop unrolling (-funroll-loops), function inlining (-finline-functions), and strict aliasing optimizations (-fstrict-aliasing). It also turns on Frame Pointer elimination (-fomit-frame-pointer). As compared to -O, this option increases both compilation time and the performance of the generated code.
-O3	PRO	Optimize even more for speed (superset of -O2). -O3 turns on all optimizations specified by -O2 and also turns on the inline-functions option.
-Os	PRO	Optimize even more for size (superset of -O2). -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

7.6.6.1 Options For Specific Optimization Control

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 7-12. Specific Optimization Options

Option	Definition
-falign-functions -falign-functions=n	Align the start of functions to the next power-of-two greater than n, skipping up to n bytes. For instance, -falign-functions=32 aligns functions to the next 32-byte boundary, but -falign-functions=24 would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less. -fno-align-functions and -falign-functions=1 are equivalent and mean that functions will not be aligned. The assembler only supports this flag when n is a power of two; so n is rounded up. If n is not specified, use a machine-dependent default.
-falign-labels -falign-labels=n	Align all branch targets to a power-of-two boundary, skipping up to n bytes like -falign-functions. 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 -falign-loops or -falign-jumps are applicable and are greater than this value, then their values are used instead. If n is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.
-falign-loops -falign-loops=n	Align loops to a power-of-two boundary, skipping up to n bytes like -falign-functions. The hope is that the loop will be executed many times, which will make up for any execution of the dummy operations. If n is not specified, use a machine-dependent default.
-fcaller-saves	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.
-fcse-follow-jumps	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 if statement with an else clause, CSE will follow the jump when the condition tested is false.

.....continued	
Option	Definition
-fcse-skip-blocks	This is similar to -fcse-follow-jumps, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if.
-fexpensive-optimizations	Perform a number of minor optimizations that are relatively expensive.
-ffunction-sections -fdata-sections	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. See also 7.6.6.2 The -ffunction-sections Option .
-fgcse	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.
-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 Read-After-Write stalls (see your device Family Reference Manual (FRM) 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.

.....continued	
Option	Definition
-fstrength-reduce	Perform the optimizations of loop strength reduction and elimination of iteration variables.
-fstrict-aliasing	<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. 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, but the following code might not:</p> <pre>int f() { a_union t; int* ip; t.d = 3.0; ip = &t.i; return *ip; }</pre>
-fthread-jumps	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.
-funroll-loops	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> .
-funroll-all-loops	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> .

7.6.6.2 The -ffunction-sections Option

The `-ffunction-sections` command-line option will try and put all functions into its own section. However, there are many conditions that can effect what exactly this means. Here is a summary:

- Normal (non interrupt) functions will have the current section name and a "." prepended to them, for example:

```
void foo() {}
```

will be placed into section `.text.foo` (the default code section name is `.text`).
- The default section name can be modified with the `-mtext` option. If this option has been used, then current section name will be changed. For example, if `-mtext=mytext` is specified, then the above function will be placed into `mytext.foo`.

- If the function has a section attribute, then it will be placed into that named section without any adulteration. Therefore,

```
void __attribute__((section("mytext"))) foo() {}
```

will always be placed into the section `mytext` regardless of whether or not `-ffunction-sections` is specified.

- Interrupt functions are normally placed into a special section with the name `.isr` prepended to the normal section name (as above). Therefore if the current section name is `.text` (the default), then the ISR is placed into `.isr.text.function_name`.

If the `-mtext` is used to change the name of the default section name, then this will be substituted instead of `.text`. However, if a named section is used with a section attribute, `.isr` will still be prepended to the section name.

The `.isr` is prepended to allow the `--gc-sections` option to not throw away interrupt functions. These must be kept.

7.6.6.3 Options that Specify Machine-Independent Flags

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 7-13. 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). <code>n</code> is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of <code>n</code> 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</code> inline 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.

.....continued	
Option	Definition
-fno-inline	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.
-fomit-frame-pointer	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.
-foptimize-sibling-calls	Optimize sibling and tail recursive calls.

7.6.7 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

Table 7-14. Preprocessor Options

Option	Definition
-Aquestion (answer)	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> . -A- 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 -A 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: -Aenviron(freestanding)
-A -predicate =answer	Cancel an assertion with the predicate <i>predicate</i> and answer <i>answer</i> .
-A predicate =answer	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.
-C	Tell the preprocessor not to discard comments. Used with the -E option.
-dD	Tell the preprocessor to not remove macro definitions into the output, in their proper sequence.
-Dmacro	Define macro <i>macro</i> with the string 1 as its definition.
-Dmacro=defn	Define macro <i>macro</i> as <i>defn</i> . All instances of -D on the command line are processed before any -U options.
-dM	Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the -E option.
-dN	Like -dD except that the macro arguments and contents are omitted. Only <code>#define</code> name is included in the output.
-fno-show-column	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 dejagnu.

.....continued

Option	Definition
-H	Print the name of each header file used, in addition to other normal activities.
-iquote, -I-	<p>Any directories you specify with -I options before the -iquote options are searched only for the case of #include "file"; they are not searched for #include <file>.</p> <p>If additional directories are specified with -I options after the -iquote, these directories are searched for all #include directives (ordinarily all -I directories are used this way).</p> <p>In addition, the iquote option inhibits the use of the current directory (where the current input file came from) as the first search directory for #include "file". There is no way to override this effect of iquote. With -I . 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.</p> <p>iquote does not inhibit the use of the standard system directories for header files. Thus, iquote and -nostdinc are independent.</p>
-I dir	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 -I option, the directories are scanned in left-to-right order; the standard system directories come after.
-idirafter dir	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 directory that -I adds within).
-imacros file	<p>Process file 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 -imacros file is to make the macros defined in file available for use in the main input.</p> <p>Any -D and -U options on the command line are always processed before -imacros file, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.</p>
-include file	<p>Process file as input before processing the regular input file. In effect, the contents of file are compiled first. Any -D and -U options on the command line are always processed before -include file, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.</p>
-iprefix prefix	Specify <i>prefix</i> as the prefix for subsequent -iwithprefix options.
-isystem dir	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.
-iwithprefix dir	Add a directory to the second include path. The directory's name is made by concatenating prefix and <i>dir</i> , where prefix was specified previously with -iprefix. If a prefix has not yet been specified, the directory containing the installed passes of the compiler is used as the default.
-iwithprefixbefore dir	Add a directory to the main include path. The directory's name is made by concatenating prefix and <i>dir</i> , as in the case of -iwithprefix.

.....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 the 7.6.2 Options for Controlling the Kind of Output section).
-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 file	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: <pre>test.o: test.c test.h test.h:</pre>
-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 target	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>
-nostdinc	Do not search the standard system directories for header files. Only the directories you have specified with -I options (and the current directory, if appropriate) are searched. See 7.6.10 Options for Directory Search for information on -I. By using both -nostdinc and -I-, the include-file search path can be limited to only those directories explicitly specified.

.....continued	
Option	Definition
-P	Tell the preprocessor not to generate #line directives. Used with the -E option (see the 7.6.2 Options for Controlling the Kind of Output section).
-trigraphs	Support ANSI C trigraphs. The -ansi option also has this effect.
-Umacro	Undefine macro <i>macro</i> . -U options are evaluated after all -D options, but before any -include and -imacros options.
-undef	Do not predefine any nonstandard macros (including architecture flags).

7.6.8 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* (DS50002106).

Table 7-15. Assembly Options

Option	Definition
-Wa, <i>option</i>	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 -Wa, -a.

7.6.9 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* (DS50002106).

Table 7-16. Linking Options

Option	Definition
--fill= <i>options</i>	<p>Fill unused program memory. The format is:</p> <pre>--fill=[wn:]expression[@address[:end_address] unused]</pre> <p><i>address</i> and <i>end_address</i> will specify the range of program memory addresses to fill. If <i>end_address</i> is not provided then the <i>expression</i> will be written to the specific memory location at address <i>address</i>. The optional literal value <i>unused</i> 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. <i>expression</i> 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 [wn:], where <i>n</i> is the fill value's width and <i>n</i> 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>

.....continued

Option	Definition
<code>--gc-sections</code>	Remove dead functions from code at link time. 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.
<code>-Ldir</code>	Add directory <i>dir</i> to the list of directories to be searched for libraries specified by the command-line option <code>-l</code> .
<code>-legacy-libc</code>	Use legacy include files and libraries (v3.24 and before). The format of include file and libraries changed in v3.25 to match HI-TECH C compiler format.
<code>-llibrary</code>	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. 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. The directories searched include several standard system directories, plus any that you specify with <code>-L</code> . 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: <code><install-path>\lib</code> for libraries specified with the <code>-l</code> option. This behavior can be overridden using the environment variables defined in the 21.3 Predefined Macro Names section.
<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.

.....continued	
Option	Definition
<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.

7.6.10 Options for Directory Search

The following options specify to the compiler where to find directories and files to search.

Table 7-17. Directory Search Options

Option	Definition
<code>-specs=file</code>	Process file after the compiler reads in the standard <i>specs</i> 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.

7.6.11 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 7-18. 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 used 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.

.....continued	
Option	Definition
-fcall-used-reg	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.
-ffixed-reg	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., -ffixed-w3.
-fno-ident	Ignore the #ident directive.
-fpack-struct	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.
-fpcc-struct-return	Return short <i>struct</i> and <i>union</i> 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.
-fno-short-double	By default, the compiler uses a <i>double</i> type equivalent to <i>float</i> . This option makes <i>double</i> equivalent to <i>long double</i> . 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.
-fshort-enums	Allocate to an <i>enum</i> type only as many bytes as it needs for the declared range of possible values. Specifically, the <i>enum</i> type will be equivalent to the smallest integer type which has enough room.
-fverbose-asm -fno-verbose-asm	Put extra commentary information in the generated assembly code to make it more readable. -fno-verbose-asm, the default, causes the extra information to be omitted and is useful when comparing two assembler files.

7.6.12 Miscellaneous Options

The following options do not fit in any of the previous categories.

Table 7-19. Miscellaneous Options

Option	Definition
-fnofallback	By default, the tool will fall back to a free license when a network, or other license, is unavailable. Specifying this option will prevent fallback and cause the compilation to fail instead.
-v	Print the commands executed during each stage of compilation.

.....continued	
Option	Definition
-x	<p>You can specify the input language explicitly with the <code>-x</code> option:</p> <p><u><code>-x language</code></u></p> <p>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.</p> <p>The following values are supported by the compiler:</p> <pre>c c-header cpp-output assembler assembler-with-cpp</pre> <p><u><code>-x none</code></u></p> <p>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>xcl6-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>

7.7 MPLAB X IDE Toolchain Equivalents

For information on related compiler options in MPLAB X IDE, see the [6. XC16 Toolchain and MPLAB X IDE](#).

8. Device-Related Features

The MPLAB XC16 C Compiler provides some features that are purely device-related.

8.1 Device Support

As discussed in the [2. Compiler Overview](#) section, 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
```

8.2 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 in your code, as this reduces 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 *MPLAB® XC16 Assembler, Linker and Utilities User's Guide* (DS50002106).

8.2.1 Register Definition Files

The processor header files described in [8.2 Device Header Files](#) 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.

8.2.2 Device Support Information

The following definitions are provided in each device header file.

Item	Description
<code>__XC16_PART_SUPPORT_VERSION</code>	A manifest constant representing the compiler release that this part-support was released with.
<code>__XC16_PART_SUPPORT_UPDATE</code>	A manifest constant representing the update increment of the part-support data.

.....continued	
Item	Description
<code>__write_to_IEC(X)</code>	<p>A macro that wraps the expression <code>X</code> with an appropriate number of <code>nop</code> instructions to ensure that the write to the IEC register has taken effect before the program executes. For example:</p> <pre>__write_to_IEC(IEC0bits.T1IE = 0);</pre> <p>will not progress until the device has disabled the interrupt enable bit for T1 (Timer1).</p>

8.2.3 Compile Time Memory Information

Each device header file incorporates macros to help identify memory sizes. For each memory region (RAM, Flash, vector table, configuration words, etc.) the header file will define two symbols: a base address and a length in bytes.

The symbol name is formed from the following template: `__<region_id>_BASE` or `__<region_id>_LENGTH`. These symbols may be used anywhere that preprocessing symbols are used.

For example:

```
#if __DATA_LENGTH < 0x1000
#error Please use a device with at least 4K of data memory
#endif
```

8.3 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 [16. 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* (DS50002106) 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* (DS50002106) for more information.

Related Links

[12.2.3 Auto Variable Allocation and Access](#)

8.4 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, watchdog timer, programming mode and code protection. Failure to correctly set these bits may result in code failure or a non-running device.

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
```

8.5 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 [8.2 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* (DS50002106) 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`.

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;
}

```

8.6 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 the [15.1.1 Function Specifiers](#) section. For more information on bit-reversed or modulo addressing, see your device Family Reference Manual (FRM).

8.7 Using EDS

EDS (Extended Data Space) is an architectural concept that allows the mapping of extra RAM into the 16-bit data addressable area. This feature uses a paging scheme, mapping in 32K pages into the address range from 0x8000 to 0xFFFF (bit 15 is set). EDS is similar to PSV (Program Space Visibility) which all 16-bit devices support.

`__eds__` is a compiler concept that allows this space to be used by either additional RAM or by FLASH. This is an extension of the architectural PSV (Program Space Visibility) window and compiler's `__psv__`/`__prog__` interpretation. PSV only facilitates the mapping of FLASH pages; EDS allows mapping RAM or Flash pages.

`__eds__` can be used on all 16-bit devices, even older devices that do not have any EDS memory. In this case, `__eds__` can be used to access Flash.

`__eds__`, `__psv__`, and `__prog__` are treated as address space qualifiers and define an access method for the compiler. These words are also used inside an address attribute to define permissible allocation.

`__attribute__((space(psv)))`, or `__attribute__((space(__psv__)))`, describe to the language tool where an object may be placed. Access and allocation are separated to allow access to be defined by the individual customer, if needed. Typically, an object allocated in a named address space would also be tagged with an address space qualifier. Here are some examples:

```
__psv__ Tau object1 __attribute__((space(psv)));
__eds__ Tau object2 __attribute__((space(psv)));
```

`object1` is allocated somewhere in PSV (Flash), and accessed through the compiler's `__psv__` mechanism; the compiler will manage the setting of the EDS page to access the object. `object2` is also allocated in PSV but accessed through the more general `__eds__` mechanism, also completely managed by the compiler.

Pointer declarations may also have address space qualifiers, but not be allocated in a named address space. In this case, it would represent a normal data space pointer, which is pointing to an object in one of the named address spaces:

```
__psv__ int *pointer_to_psv_int;
int * __psv__ psv_pointer_to_int __attribute__((space(psv)));
```

The way to decode these is to read outwards from the object name. The first defines an object, in data space, that points to an object in `psv` space. The 2nd defines an object that lives in `psv` space which points to an object in data space; this object needs a `space` attribute to get located into an appropriate named address space.

See also [12.6 Extended Data Space Access](#)

8.7.1 Memory Models and Address Spaces

By default, the compiler uses the `const-in-code` memory model, which will allocate `const` qualified objects into a single PSV window, limited to 32K in size. This window will be the default page that is mapped into the EDS area in

the address map. The compiler requires that this page always be available if it is setup. This is often referred to as the auto PSV memory model because the compiler automatically manages the PSV.

Other objects can be explicitly placed into different areas of memory using a `space` attribute, such as `__attribute__((space(psv)))` for a windowed Flash area or `__attribute__((space(eds)))` for a RAM area in the EDS space. When there is a need to nominate a Y memory space and have that be in EDS, use `__attribute__((space(eds)))`, which is equivalent to `__attribute__((eds))`.

When specifying an address space it is normal, but not required, to also use an address space qualifier. Doing so will ask the compiler to manage the access to the named address space for you, which will mean always maintaining the const-in-code page access if it is enabled.

8.7.2 Optimizations

Using the EDS/PSV address space qualifiers may be expensive. If your application footprint does not need to access these variables often, then it may be more efficient to use the const-in-data memory model and manually place the less often accessed data into a Flash space (using the named address attributes and qualifiers). This can reduce the overhead by allowing the compiler to not force the EDS access to return to a single page. This model does not preclude the use of an automatically managed PSV space; using `space(auto_psv)` allows the programmer to nominate which `const` variables go into this area.

By default the compiler will arrange to assert the const-in-code page at the start of an interrupt service routine. If an ISR does not need to access const data, then specifying `__attribute__((no_auto_psv))` will let the compiler know that the ISR, or any function it calls, does not use any auto PSV data.

The compiler also has an optimization setting that attempts to reduce the number of page swaps; not as a cache, but modifications of the DSR/DSW/PSV Page SFR register. This is a separate switch, `-moptimize-page-setting`, which can be applied at any optimization level. Like many optimizations, it generally reduces the number page setting operations which may reduce code-size and improve application performance.

8.7.3 C Library Function Extensions

Named address space qualifiers are not part of the definition for the standard C library. In order to maintain C compliance, MPLAB XC16 adds extended versions of some library functions instead of supporting a modified signature.

For example, it is not possible to pass an `__eds__ char *` pointer to a `printf %s` argument.

MPLAB XC16 supports extended versions of `memcpy`, `strcpy` and `strncpy` standard C functions. Please see the *16-Bit Language Tools Libraries Reference Manual* (DS0001456), "Functions for Specialized Copying and Initialization."

8.8 Stack Usage Guidance

MPLAB XC16, in common with other Microchip MPLAB XC compiler offerings, provides a feature we are calling 'Stack Usage Guidance'. This is a compile-time static analysis of the executable meant to guide the user to an appropriate worst-case stack usage requirement. As will be seen, this guidance may need to be tempered with other facts that cannot be determined by a static analysis.

8.8.1 Usage

This feature requires a 'PRO' license and is enabled by adding `-mchp-stack-usage` to the normal link line, either through the `xc16-ld` or `xc16-gcc` interface. The MPLAB X IDE Project Properties window provides a convenient check box to enable to this feature under the category *XC16>Analysis Tools*.

When this feature is enabled, MPLAB XC16 will provide some additional information pertaining to the linked executable to the terminal or build-output window of MPLAB X. Additionally, if a 'map' file is being specified, the language tool will emit more detailed information to the Map file for a permanent record.

8.8.2 Operation

When enabled, MPLAB XC16 will analyze the complete executable and attempt to provide the amount of stack at the deepest point in memory. Again, this is a static analysis and may be limited by the information that is available at

compile-time. During analysis, the tool may discover that a complete picture of the stack usage cannot be made. If this is the case, additional information will be displayed on the output terminal. This information will also be recorded in the Map file, if specified.

The following items may trigger a partial result to be made: recursion; explicit dynamic stack assignments or adjustments; stack usage caused by interrupt functions; unconnected, or disjoint, functions or other executable code. By definition, these types of items are dynamic and therefore their stack usage cannot be determined via static analysis. More detail on these items follows.

The summary information is provided in the following form:

```
===== STACK USAGE GUIDANCE =====

In the call graph beginning at __reset(),
324 bytes of stack are required.
```

All MPLAB XC16 executables start at `__reset`; so we provide stack guidance starting from this point.

8.8.2.1 Recursion

Recursion occurs when the program directly or indirectly causes a loop where there is a net adjustment of the stack usage. MPLAB XC16 does not limit this identification across function boundaries; indeed (un)carefully constructed assembly language programs could cause this error to be emitted even within a function boundary. The most obvious example would be a traditional, recursive, implementation of factorial. A more subtle version from the standard C library is `fflush()` - which, if given a NULL parameter, will call `fflush()` on all the currently open IO streams.

Any recursion that is detected will cause MPLAB XC16 to emit the following messages:

```
1. Recursion has been detected:
   _function_name (address 0x8bc)
   No stack usage predictions can be made.
```

Each function, where the recursion has been detected, will appear in the list. The output to the Map file will also contain the code address near where recursion has been detected.

8.8.2.2 Stack Adjustments

Any stack adjustments that cannot be determined at compile time will trigger a message. This may be caused by assembly code writing a value explicitly to the stack pointer (W15) or by a variable allocation using `alloca()`. In this case, we will emit the following diagnostic:

```
2. Indeterminate stack adjustment has been detected:
   _function_name (address 0x286c)
   No stack usage predictions can be made.
```

Each function where stack adjustment has been detected will appear in the list. The output to the Map file will also contain the code address near where adjustment has been detected.

8.8.2.3 Interrupt Functions

Every embedded application will have some interrupt functions. Unfortunately we cannot determine when these will occur during the run-time flow of the application. When interrupt functions are discovered, we will emit the following information:

```
3. The following labels are interrupt functions:
   __T1Interrupt uses 18 bytes (address 0x2f8)
   We cannot determine the stack impact of these events. Please adjust the guidance according to
   the application flow.
```

Each interrupt function will appear in the list. It will contain the amount of stack that is used by that interrupt including any functions that may be called. The code address of the interrupt handler will be emitted to the Map file.

The total amount of allocated stack should be increased by the total amount of stack that can be consumed by the largest interrupt flow; this would include any interrupts that may themselves be interrupted. Your application may limit which interrupts are themselves interruptible using various means. As the application developer, please adjust the stack usage based upon your knowledge of how the nested interrupt system will work in your application.

8.8.2.4 Unconnected Executable Code

Unconnected executable code cannot be reached by normal flow analysis. Perhaps this code represents functions that are never called, or functions that are called indirectly via function pointers. In any event, we cannot determine when or if these functions are actually called. When detected, the tool will emit the following messages:

```
4. The following labels cannot be connected to the main call graph.  
This is usually caused by some indirection:  
_function_name uses 10 bytes (address 0x598)  
We cannot determine the stack impact of these events. Please adjust the guidance according to  
the application flow.
```

Each block of code that cannot be reached will be displayed along with the amount of stack consumed. The address of the code block will be presented only in the Map file. These may not be complete functions, though it is unlikely for C code.

8.8.3 Using the Guidance

The value given in the stack guidance only represents what the tool can see. It does not include any 'stack guard' setting that might be present. The default stack guard is 16-bytes, but this is configurable for each project. Adjust the guidance based upon the caveats that may be presented in the report. An explicit stack can be allocated using the features described in the [12.2.3.2 The C Stack Usage](#) section. A minimum stack size can be enforced using the features described in the [12.2.3.1 Software Stack](#) section.

9. Differences Between MPLAB XC16 and ANSI C

This compiler conforms to the ANSI 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.

9.1 Divergence from the ANSI C Standard

There are no divergences from the ANSI C standard.

9.2 Extensions to the ANSI C Standard

The MPLAB XC16 C Compiler provides extensions to the ANSI C standard in these areas: keywords and expressions.

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 – [10.10 Variable Attributes](#)
- Specifying Attributes of Functions – [15.1.1 Function Specifiers](#)
- Inline Functions – [15.5 Inline Functions](#)
- Variables in Specified Registers – [12.10 Allocation of Variables to Registers](#)

Expression Differences

Expression differences are:

Binary Constants – [10.7 Literal Constant Types and Formats](#)

9.3 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 [23. Implementation-Defined Behavior](#).

10. Supported Data Types and Variables

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 section [12. Memory Allocation and Access](#).

10.1 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* (DS50002106).

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.

10.2 Integer Data Types

The following table 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 10-1. Integer Data Types

Type	Bits	Min.	Max.
<code>char</code> , signed <code>char</code>	8	-128	127
unsigned <code>char</code>	8	0	255
<code>short</code> , signed <code>short</code>	16	-32768	32767
unsigned <code>short</code>	16	0	65535
<code>int</code> , signed <code>int</code>	16	-32768	32767
unsigned <code>int</code>	16	0	65535
<code>long</code> , signed <code>long</code>	32	-2 ³¹	2 ³¹ - 1
unsigned <code>long</code>	32	0	2 ³² - 1
<code>long long*</code> , signed <code>long long*</code>	64	-2 ⁶³	2 ⁶³ - 1
unsigned <code>long long*</code>	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:

<code>0x100</code>	<code>0x101</code>	<code>0x102</code>	<code>0x103</code>
<code>0x78</code>	<code>0x56</code>	<code>0x34</code>	<code>0x12</code>

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 [23.5 Integers](#).

10.2.1 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.

10.2.2 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 [7.6.3 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.

10.3 Floating-Point Data Types

The compiler uses the IEEE-754 format. The following table shows floating point data types that are supported. All floating point data types are arithmetic type real.

Table 10-2. 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 <code>-fno-short-double</code> is used.					

All floating point 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 `double` value of 1.2345678 is stored at address `0x100` as follows:

0x100	0x101	0x102	0x103
0x51	0x06	0x9E	0x3F

As another example, the `double` 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 23.6 [Floating Point](#).

10.4 Fixed-Point Data Types

The following table shows fixed-point data types that are supported by the compiler when the `-menable-fixed` command line option is specified. See 11. [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 10-3. 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

10.5 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 the [23.9 Structures, Unions, Enumerations and Bit-Fields](#) section.

10.5.1 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};
```

10.5.2 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 the [7.6.3 Options for Controlling the C Dialect](#) section).

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.

10.6 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.

10.6.1 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.

10.6.2 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 [7.6.1 Options Specific to 16-Bit Devices](#) for more information.)

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 [12. Memory Allocation and Access](#) for more information.

10.6.3 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.

Because function pointers are only 16 bits wide, these pointers cannot point beyond the first 64K of Flash. If the address of a function that is allocated beyond the first 64K of Flash is 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.

10.6.4 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.

10.7 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 the following table. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Table 10-4. Radix formats

Radix	Format	Example
binary	<code>0b</code> number or <code>0B</code> number	<code>0b10011010</code>
octal	<code>0</code> number	<code>0763</code>
decimal	number	<code>129</code>
hexadecimal	<code>0x</code> number or <code>0X</code> number	<code>0x2F</code>

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. The table below 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 lineal constant; otherwise, it will assign `long long int`. If the literal constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

Table 10-5. 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
}
```

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 characters 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 "Bjørk'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."
```

10.8 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.

10.8.1 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 the [12.3 Variables in Program Space](#) section). 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.

10.8.2 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. 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 [18. Mixing C and Assembly Code](#) if you have assembly code in your project.

10.9 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.

10.9.1 __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 the [10.10 Variable Attributes](#) section. For basic information on the memory layout and how program memory is accessed by the device, see the [12.1 Address Spaces](#) section.

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 the [10.9.2 __prog__ Type Qualifier](#) section) and an appropriate space attribute.

10.9.2 __prog__ Type Qualifier

The `__prog__` qualifier is similar to the `__psv__` qualifier (see [10.9.1 __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 [10.10 Variable Attributes](#) and [12.1 Address Spaces](#) for basic information on the memory layout and how program memory is accessed by the device.

10.9.3 __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 the [12.6 Extended Data Space Access](#) section.

10.9.4 `__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 the [12.9 Packing Data Stored in Flash](#) section.

10.9.5 `__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 the [10.10 Variable Attributes](#) section. For basic information on the memory layout and how program memory is accessed by the device, see the [12.1 Address Spaces](#) section.

For more on the qualifier, see the [12.4 Parallel Master Port Access](#) section.

10.9.6 `__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 the [10.9.5 __pmp__ Type Qualifier](#) section). 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 the [12.5 External Memory Access](#) section for more information on how the memory space is configured and access routines are defined.

The qualifier is used as in the following example.

```
__external__ int external_array[256]
        __attribute__((space(external(external_memory))));
__external__ char * myExternalPointer;
```

In addition to the qualifier, the array 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 the [10.10 Variable Attributes](#) section. For basic information on the memory layout and how program memory is accessed by the device, see the [12.1 Address Spaces](#) section.

For more on the qualifier, see the [12.5 External Memory Access](#) section.

10.10 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 the [15.1.2 Function Attributes](#) section. Qualifiers are used independently to attributes (see the [10.9 Compiler-Specific Type Qualifiers](#) section). 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 attributes below are currently supported for variables.

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)).
```

Note: It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in file A with the `far` attribute and declared `extern` in file B without `far`, then a link error may result.

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 have the `__eds__` type qualifier (see the [12.6 Extended Data Space Access](#) section) 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

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 };
```

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.

page

This attribute specifies that the object cannot exceed a page boundary. The page boundary applied depends upon where the object is allocated. An object located in a `psv` space cannot cross a 32K boundary; an object located in `prog` space cannot cross a 64K boundary.

```
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 [12.6 Extended Data Space Access](#) for more information.

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;
    }
}

```

This attribute can only be used in conjunction with a RAM resident object, i.e. not in FLASH.

preserved

The `preserved` attribute can be applied to a variable to indicate that this variable's value should be preserved on a restart. A restart is a user-defined event which can be different from a cold or warm reset. Preserved variables require information from a previously linked executable in order to function; please see the linker option `--preserved=`.

priority(n)

The `priority` attribute can be applied to a variable to group initializations together. `n` must be between 1 and 65535, with 1 being the highest level. All initializations with the same priority are initialized before moving onto the next priority level. Level 1 variables are initialized first and variables without a priority level are initialized last. The attribute can also be applied to `void` functions (`void` result and argument types); in this case the function(s) for level `n` will be executed immediately after all the initializations for level `n` are complete.

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}};
```

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.

shared

Used with co-resident applications. The variable may be used outside of the application. A data item will be initialized at startup of any application in the co-resident set.

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 [12.1 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.
- **dataflash** – Allocate the variable in dataflash.
- **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)));
```

- **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 [12.3.2 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.

- **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 [28. 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.
- **eedata - PIC24F, dsPIC30F/33F DSCs only** – Allocate the variable in EEPROM Data (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 the [12.4 Parallel Master Port Access](#) section.
- **external** – Allocate the variable in a user defined memory space. For complete details please see the [12.5 External Memory Access](#) section.

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.

update

The update attribute can be applied to a variable to indicate that this variable should be initialized on a restart. This is particularly useful if `-mpreserve-all` or `--preserve-all` is being used.

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 inaccessible.

The behavior is identical, regardless of the type of the symbol; functions and variables behave in the same manner.

11. Fixed-Point Arithmetic Support

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, can be accessed using the following link:

www.open-std.org/JTC1/SC22/WG14/www/projects#18037

This chapter describes the implementation-specific details of the types and operations supported by the compiler under this draft standard.

11.1 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 the [7.6 Driver Option Descriptions](#) section.

11.2 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 11-1. 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 11-2. 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
<code>signed long _Fract</code>	1.31
<code>unsigned long _Fract</code>	1.31 (sign bit 0)
<code>_Accum</code>	9.31
<code>short _Accum</code>	9.31
<code>long _Accum</code>	9.31
<code>signed _Accum</code>	9.31
<code>signed short _Accum</code>	9.31

.....continued	
Type	Format
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). Therefore 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.)

11.3 Rounding

Three rounding modes are supported, corresponding to the three rounding modes supported by the 16-bit device fixed-point multiplication facilities.

Table 11-3. 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 the [7.6 Driver Option Descriptions](#) section or on a function-by-function basis, via the `-round` attribute, as described in [15.1.2 Function Attributes](#).

These modes are described in more detail in the “16-bit MCU and DSC Programmer’s Reference Manual” (DS70157).

11.4 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.

11.5 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.

11.6 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 the [15.1.2 Function Attributes](#) section. Programs constructed in this manner will operate correctly, at the expense of some state saves and restores around calls to the indicated assembly routines.

12. Memory Allocation and Access

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.

12.1 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 dsPIC and PIC24 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. See sections [12.2 Variables In Data Space Memory](#) and [16.7.2 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. It is also possible to map Flash and other areas, see section [12.6 Extended Data Space Access](#).

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.

12.2 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.

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.

12.2.1 Auto and Non-Auto Variables vs. Local and Global Variables

The terms "local" and "global" are commonly used to describe variables, but are not 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.

12.2.2 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 [10.8.1 Const Type Qualifier](#)) into the data space memory if the constants-in-data memory model is selected; otherwise, they are located in program memory.

12.2.2.1 Default Allocation of Non-auto Variables

The compiler considers several categories of `static` and `external` variables which all relate to the value which the variable should contain at the time the program begins. That is, those that should be cleared at program startup (uninitialized variables), those that should hold a non-zero value (initialized variables), and those that should not be altered at all during 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.

12.2.2.2 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 which 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 the [7.3.2 Startup and Initialization](#) section).

12.2.2.3 Non-Auto Variable Size Limits

The compiler option `-mlarge-arrays` allows you to define and access arrays greater than or equal to 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 require 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));
}
```

12.2.2.4 Changing Non-Auto Variable Allocation

The compiler arranges for data to be placed into sections, depending on the memory model used and whether or not the data is initialized, as described in the [12.1 Address Spaces](#) section. 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 section [9. 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;
```

12.2.2.5 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 EEPROM Data (EEData) space 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;
```

12.2.3 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 below in [12.2.3.1 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.

12.2.3.1 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 [16. 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* (DS50002106) 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* (DS50002106) for more information.

12.2.3.2 The C Stack Usage

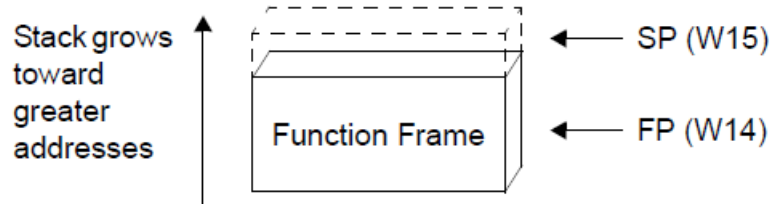
The C compiler uses the software stack to:

- 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.

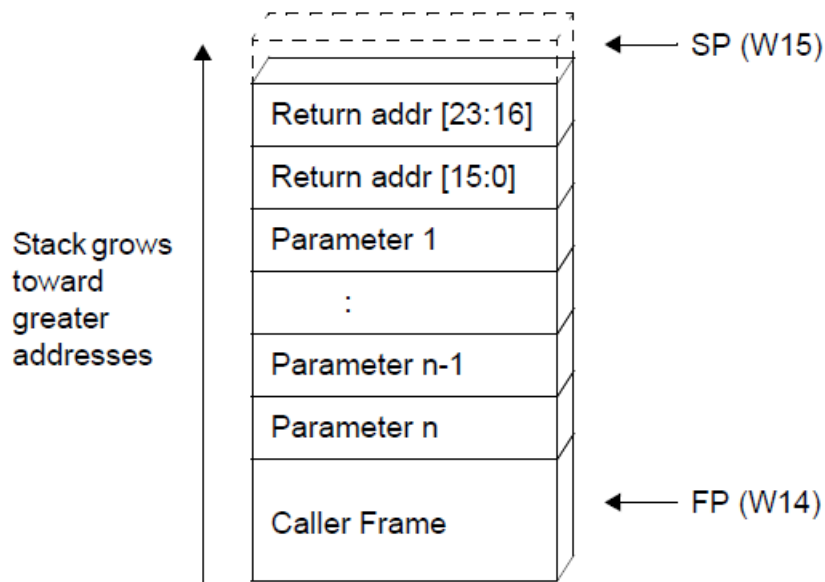
Figure 12-1. Stack and Frame Pointers



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.

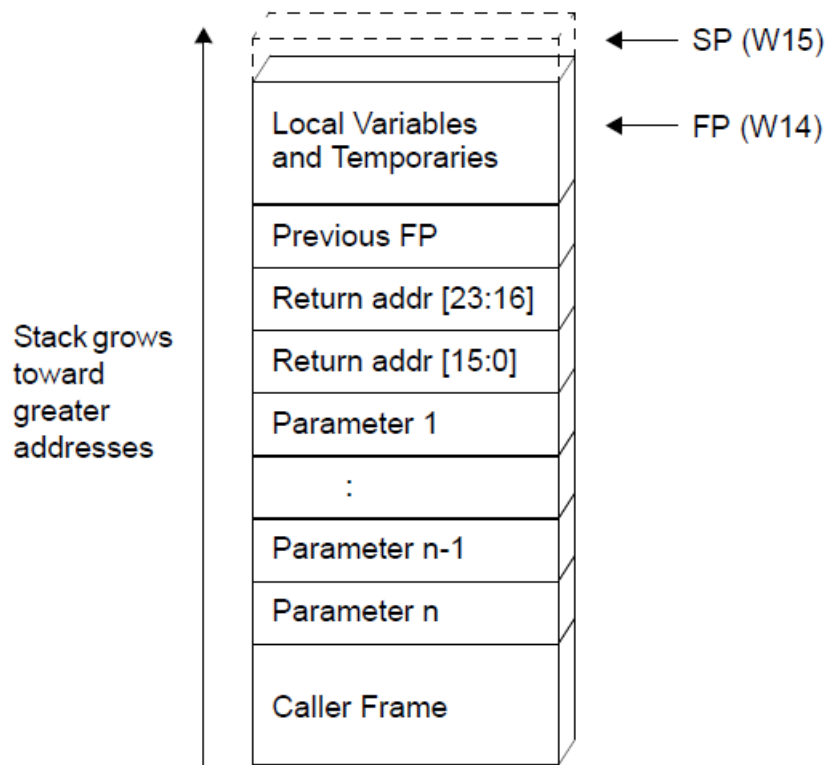
The following diagrams illustrate the steps involved in calling a function. Executing a `CALL` or `RCALL` instruction pushes the return address onto the software stack.

Figure 12-2. CALL or RCALL



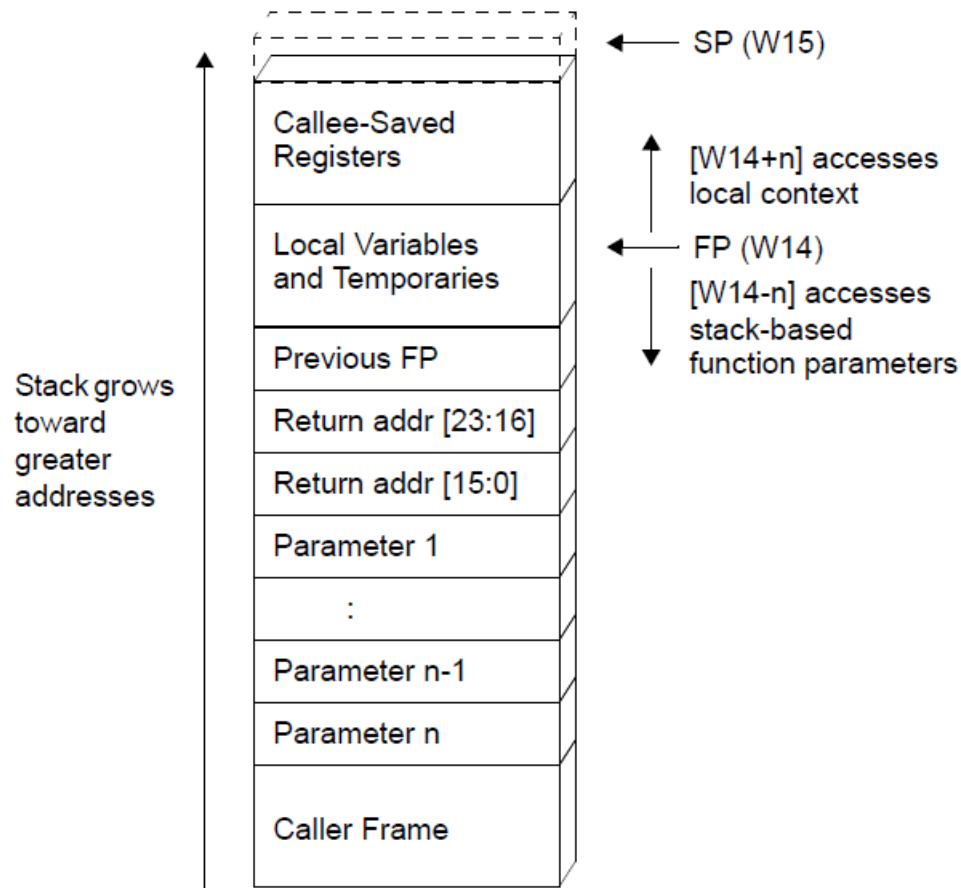
The called function (callee) can now allocate space for its local context.

Figure 12-3. Callee Space Allocation



Any callee-saved registers that are used in the function are pushed.

Figure 12-4. Push Callee-Saved Registers



12.2.3.3 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. [12.2.2 Non-Auto Variable Allocation and Access](#), [Non-Auto Variable Size Limits](#), which describes defining and using large arrays is also applicable to `auto` objects.

12.2.4 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.

12.3 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 (see sections [12.3.1 Allocation and Access of Program Memory Objects](#) and [16.7.2 PSV Usage with Interrupt Service Routines](#)).

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 Iotype = 'A'; // initialized const object
const char buffer[10]; // I reserve memory in RAM
```

See the [18. Mixing C and Assembly Code](#) section for the equivalent assembly symbols that are used to represent `const`-qualified variables in program memory.

12.3.1 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.

12.3.1.1 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 [12.3.2 Access of Objects in Program Memory](#), Managed PSV Access.

12.3.1.2 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.

12.3.1.3 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.

12.3.2 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.

Note that `boot` or `secure` interrupt service routines will use a different setting of the `PSVPAG` register for their constant data.

12.3.2.1 Managed PSV Access

The compiler introduces several new qualifiers (or more specifically, CV-qualifiers). 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 [12.6 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;
```

12.3.2.2 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.

12.3.2.3 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)) _TlInterrupt(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.

12.3.3 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 [10.5 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. The [12.2.2 Non-Auto Variable Allocation and Access](#) section, describes defining and using large arrays is also applicable to objects in program space memory.

12.3.4 Changing Program Memory Variable Allocation

The variables allocated to program memory can, to some degree, be allocated to alternate memory locations. [12.2.2 Non-Auto Variable Allocation and Access](#), 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 [9. 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* (DS50002106).

12.4 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 your device Family Reference Manual (FRM) or data sheet.

Note: PMP attributes are not supported on devices with EPMP. Use Extended Data Space (EDS) instead (see the [12.6 Extended Data Space Access](#) section).

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:

- [12.4.1 Initialize PMP](#) - define the initialization function: `void __init_PMP(void)`
- [12.5.1 Declare a New Memory Space](#)
- [12.4.3 Define Variables within PMP Space](#)

12.4.1 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 */
}
```

12.4.2 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 [9. 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.

12.4.3 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.

12.5 External Memory Access

Not all of Microchip's 16-bit devices have a parallel master port peripheral (see [12.4 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.

12.5.1 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 9. [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`.

12.5.2 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.

12.5.3 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.

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.

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.

12.5.4 An External Example

The following snippets 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 defines 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.

12.6 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 `DSRPAG` to represent extended read access to Flash or the extended data space (EDS).

12.7 Dataflash Memory Access

The language tool cannot generate access to this kind of memory as it can generate access to other on-board memories. Please see your device data sheet or Family Reference Manual (FRM) for suggested access routines.

The compiler defines the following features to assist in defining and using dataflash memory that is available on certain devices.

1. The tool defines a new memory space, `space(dataflash)`, which may be applied as an attribute to any variable.
2. The tool provides a new builtin to determine the address of a properly attributed dataflash variable:

```
int foo[5] __attribute__((space(dataflash))) = { 1,2,3,4,5 };

offset = __builtin_dataflashoffset(&foo);
```

12.8 Dual Partition Memory Access

The language tool chain supports a new option that directs the compiler and linker to target a single partition in a dual partition device. This option will constrain the output text to be contained within one panel and is selectable from with the MPLAB X IDE or from the command line using `--partition n`.

12.9 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 the [12.3 Variables in Program Space](#) section. 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.

12.9.1 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.

12.9.2 Usage Considerations

When using this qualifier, consider the following:

1. The following attributes are not compatible with `__pack_upper_byte`:

boot	near	reverse
dma	noload	xmemory
eedata	psv, auto_psv	ymemory

2. `__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.
3. Sequential accesses to `__pack_upper_byte` data objects will improve access performance.

4. 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);
```

5. 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 [12.9.1 Packed Example](#)).

6. The TBLPAG SFR may be corrupted during access of a packed variable.

12.9.3 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.

12.10 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.

12.11 Variables in EEPROM Data Space (Device Dependent)

The compiler provides some convenience macro definitions to allow placement of data into the device's EEPROM Data (EEData) 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 EEData 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).

12.11.1 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 EEData 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 12-1. 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 EEData 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.

12.11.2 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 EEData, 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 12-2. EEData Access Via Table Read

```
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 */
}

```

12.11.3 Accessing EEData Using Managed Access

On most device the EEData space is part of the program address space. Therefore EEData can be accessed automatically using one of the managed access qualifiers `__psv__` or `__eds__`.

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 */
}
```

12.11.4 Additional Sources of Information

Device Family Reference Manuals (FRMs) have an excellent discussion on using the Flash program memory and EEPROM data memory provided. These manuals also have information on run-time programming of both types of memory.

There are many library routines provided with the compiler. See the *"16-Bit Language Tools Libraries Reference Manual"* (DS50001456) manual for more information.

12.12 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 -T pdevice.gld 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`.

12.13 Co-Resident Applications

Co-resident applications are programs that share the same physical memory space on an MCU or DSC. These applications are linked together in such a way that they can share the device memory resource.

See the *"MPLAB® XC16 Assembler, Linker and Utilities User's Guide"* (DS50002106), Section 10.15 "Co-resident Application Linking," for details.

12.14 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 12-1. Memory Model Command-Line Options

Option	Memory Definition	Description
<code>-msmall-data</code>	Up to 6 KB of data memory ¹ . The default is device dependent ² .	Permits use of PIC18 like instructions for accessing data memory.
<code>-msmall-scalar</code>	Up to 6 KB of data memory ¹ . This is the default.	Permits use of PIC18 like instructions for accessing scalars in data memory.

.....continued

Option	Memory Definition	Description
<code>-mlarge-data</code>	Greater than 6 KB of data memory ¹ . The default is device dependent ² .	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-auxflash</code>	Constants in auxiliary Flash.	Values are accessed via Program Space visibility window.

Notes:

1. For most devices 6K of RAM is the near data space, but for some devices it is 4K of RAM.
2. For devices that have all of their data memory in the near space, the memory model is “small data” “small scalar” so that all memory will be allocated in the near space.
For all other devices the default memory model is “large data” “small scalar”. This will have the effect of allowing the tool chain to place aggregate objects, such as arrays and structure, into the far memory space. This can be over-ridden by explicitly selecting “small data” in the compiler options.

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 [10.10 Variable Attributes](#) and [15.1.1 Function Specifiers](#).

12.14.1 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 6 KB (or 4 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 6 KB (or 4 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 6 KB (or 4 KB) near data space.

12.14.2 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 the near (small) or far (large) code models by tagging their declarations with the `near` or `far` 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. Operators and Statements

The MPLAB XC16 C Compiler supports all the ANSI operators. The exact results of some of these operators are implementation defined and this behavior is fully documented in the [23. Implementation-Defined Behavior](#) section. The following sections illustrate code operations that are often misunderstood as well as additional operations that the compiler is capable of performing.

13.1 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 the [28. Built-in Functions](#) section.

13.2 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 complement 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. 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.

14. Register Usage

Certain registers play import roles in the C runtime environment. Therefore creating code concerning these registers requires knowledge about their use by the compiler.

14.1 Register Variables

Register variables use one or more working registers, as shown in the following table.

Table 14-1. 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, W12}; W12 is included only if W14 is not used as a frame pointer.
Note 1: double is equivalent to long double if -fno-short-double is used.	

14.2 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.

15. Functions

The compiler supports C code functions and handles assembly code functions, as discussed in this chapter.

15.1 Writing Functions

Implementation and special features associated with functions are discussed in the following sections.

15.1.1 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.

15.1.2 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:

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.

15.1.2.1 `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)));
```

15.1.2.2 `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.

15.1.2.3 `auto_psv, no_auto_psv`

The `auto_psv` attribute, when combined with the `interrupt`, `boot` or `secure` 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`, `boot` or `secure` 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`.

15.1.2.4 `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 *MPLAB® XC16 Assembler, Linker and Utilities User’s Guide* (DS50002106).

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()`.

The `boot` attribute may be combined with the `auto_psv` or `no_auto_psv` attribute. For details, see the section `auto_psv`, `no_auto_psv`.

15.1.2.5 `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 int));
```

says that the hypothetical function `square` is safe to call fewer times than the program states.

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.

15.1.2.6 context

The `context` attribute may be used to associate the current routine with an alternate register set. Typically this is used with interrupt service routines to reduce the amount of context that must be preserved, which will improve interrupt latency.

15.1.2.7 deprecated

See [10.10 Variable Attributes](#) for information on the `deprecated` attribute.

15.1.2.8 far

The `far` attribute tells the compiler that the function may be located too far away to use short call instruction.

15.1.2.9 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`.

15.1.2.10 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 [16. 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.

15.1.2.11 keep

The `keep` attribute will prevent the linker from removing the function with the ELF linker option `--gc-sections`, even if it is unused.

in C:

```
void __attribute__((keep)) foo(void);
```

in Assembly:

```
.section *,code,keep
.global _foo
_foo:
    return
```

15.1.2.12 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();
```

15.1.2.13 near

The `near` attribute tells the compiler that the function can be called using a more efficient form of the call instruction.

15.1.2.14 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)) {
    ...
}
```

15.1.2.15 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. It also 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.

15.1.2.16 optimize

Use the `optimize` attribute to specify different optimization options for various functions within a source file. Arguments can be either numbers or strings. Numbers are assumed to be an optimization level. Strings that begin with `O` are assumed to be an optimization option. This feature can be used, for example, to have frequently executed functions compiled with more aggressive optimization options that produce faster and larger code, while other functions can be called with less aggressive options.

This optimization setting overrides the file or project optimization setting.

```
int __attribute__((optimize("-O3"))) pandora (void)
{
    if (maya > axton) return 1;
    return 0;
}
```

15.1.2.17 priority(n)

The `priority` attribute can be applied to a variable to group initializations together. `n` must be between 1 and 65535, with 1 being the highest level. All initializations with the same priority are initialized before moving onto the next priority level. Level 1 variables are initialized first and variables without a priority level are initialized last. The attribute can also be applied to `void` functions (`void` result and argument types); in this case the function(s) for level `n` will be executed immediately after all the initializations for level `n` are complete.

15.1.2.18 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.

15.1.2.19 save(list)

Functions declared with the `save(list)` attribute will direct the compiler to save and restore the C variables expressed in `list`.

15.1.2.20 section ("section-name")

Normally, the compiler places the code it generates in the `.text` section. Sometimes you need additional sections or 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.

15.1.2.21 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()`.

The `secure` attribute may be combined with the `auto_psv` or `no_auto_psv` attribute. For details, see the section `auto_psv`, `no_auto_psv`.

15.1.2.22 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);
```

15.1.2.23 shared

Used with co-resident applications. The function may be used outside of the application. A data item will be initialized at startup of any application in the co-resident set.

15.1.2.24 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.

15.1.2.25 `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.

15.1.2.26 `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
}
```

15.1.2.27 `weak`

See [10.10 Variable Attributes](#) for information on the `weak` attribute.

15.2 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 [section 15.3 Allocation of Function Code](#) for more details).

15.3 Allocation of Function Code

Code associated with functions is always placed in the program memory of the target device. The compiler arranges for code to be placed in the `.text` section (as described in [12.1 Address Spaces](#)), 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.

15.4 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 15.1.1 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.

15.5 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.

15.6 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 part that you are using and the type of memory usage.

For details, see [15.6 Memory Models](#).

15.7 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.
- DSRPAG/PSVPAG should be preserved if the `-mconst-in-code (auto_psv)` memory model is being used.

Table 15-1. Registers Required

Data Type	Number of Working Registers Required
<code>char</code>	1
<code>int</code>	1
<code>short</code>	1
<code>pointer</code>	1 (eds pointer requires 2)
<code>long</code>	2 (contiguous – aligned to even numbered register)
<code>float</code>	2 (contiguous – aligned to even numbered register)
<code>double*</code>	2 (contiguous – aligned to even numbered register)
<code>long double</code>	4 (contiguous – aligned to quad numbered register)
<code>structure</code>	1 register per 2 bytes in structure
<code>_Fract</code>	1
<code>long _Fract</code>	2 (contiguous – aligned to even numbered register)
<code>_Accum</code>	3 (contiguous – aligned to quad numbered register)
* <code>double</code> is equivalent to <code>long double</code> if <code>-fno-short-double</code> 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.

15.7.1 Function Parameters

The first eight working registers (W0-W7) are used for function parameters. Parameters are allocated to registers in left-to-right order, with the parameter being 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 15-1. 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 15-2. 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 15-3. 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), as 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 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 the example [Example 15-3](#). `b` would be accessed via W15-20; 4 bytes for the return address and 16 bytes to access the start of `b`.

15.7.2 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.

15.7.3 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.

16. Interrupts

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:

- [16.1 Interrupt Operation](#) – An overview of how interrupts operate.
- [16.2 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.
- [16.3 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.
- [16.4 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.
- [16.5 Nesting Interrupts](#) – The time between when an interrupt is called and when the first ISR instruction is executed is the latency of the interrupt.
- [16.6 Enabling/Disabling Interrupts](#) – How interrupt priorities are determined. Enabling and disabling interrupt sources occurs at two levels: globally and individually.
- [16.7 ISR Considerations](#) – Sharing memory with mainline code, PSV usage with ISRs, and calling functions within ISRs.

16.1 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 allows 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.

16.2 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. For details, see section [7.6.6.2 The -ffunction-sections Option](#).

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.

16.2.1 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 [16.5 Nesting Interrupts](#) for more information.

16.2.2 Syntax for Writing ISRs

To declare a C function as an interrupt handler, tag the function with the interrupt attribute (see the [15.1.2 Function Attributes](#) section 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, with the optional `altirq` parameter allowing you to place an interrupt vector at a specified alternate interrupt. Each parameter requires a parenthesized interrupt ID number (see the [16.3 Specifying the Interrupt Vector](#) section 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.

16.2.3 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 [15.1.1 Function Specifiers](#)). For example:

```
void __attribute__((interrupt(auto_psv, shadow))) isr0(void);
```

16.2.4 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);
```

16.3 Specifying the Interrupt Vector

All 16-bit devices have a primary interrupt vector table. Some 16-bit devices have a fixed alternate vector table, some have no alternate vector table and some have an alternate vector table which may be disabled and moved.

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.

The alternate vector name is used when the `ALTIVT` bit is set in the `INTCON2` register. For devices with alternate vector tables which may be disabled and moved, AIVT support is configured via configuration words, notably:

- `AIVTDIS` to enable the vector table
- `BSLIM` to locate the vector table

On these devices, the tool chain will inspect the values of these configuration words to determine whether or not to allocate space and fill in the value of the alternate vector tables. Simply specify device appropriate values for these configuration words:

```
#pragma config AIVTDIS = ON
#pragma config BSLIM = 0x1FFD
```

and define the alternate vectors in the normal way, i.e.:

```
void __attribute__((interrupt)) _AltT1Interrupt(void) {}
```

Each exception vector occupies a program word. For tables of interrupt vectors by device family, see the [16.3.2 Interrupt Vector Tables](#) section.

16.3.1 Interrupt Vector Usage

To field an interrupt, a function's address must be placed at the appropriate address in one of the vector tables, with the function preserving 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 vector tables (see section [16.3.2 Interrupt Vector Tables](#)). 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 vector tables ([16.3.2 Interrupt Vector Tables](#)), or you may use names more appropriate to the application, 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 53, 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.

16.3.2 Interrupt Vector Tables

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 ([6.1 MPLAB X IDE and Tools Installation](#)). Open the `XC16MasterIndex` file and click on the "Interrupt Vector Tables Reference" link.

16.4 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.

16.4.1 Assembly and ISRs

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 the [16.7.3 Latency](#) section.

16.4.2 context Attribute

The `context` attribute may be applied to an interrupt service routine to inform the compiler that this ISR executes at a particular Interrupt Priority Level (IPL), which has also been assigned to an alternate register set. Please see your device data sheet or Family Reference Manual (FRM) for details on how to properly configure the device to use alternate register sets. This feature is set up using configuration bits.

When using this attribute, it is important that the priority level of the interrupt matches the priority level of the context that has been assigned. Changing the priority of the interrupt service routine may cause runtime corruption.

Example of use:

```
// Priority Level 7 routines will use context 1
#pragma config CTXT1 = 7
// T1 Interrupt uses its own context
void __attribute__((interrupt, context)) _T1Interrupt(void);
main() {
    // Timer 1 is configured to use priority level 7
    IPC0bits.T1IP = 7;
```

16.5 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.

16.6 Enabling/Disabling Interrupts

Note: Traps, such as the address error trap, cannot be disabled. Only IRQs can be disabled.

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.

The safe method of enabling and disabling peripheral interrupts is to use the `__write_to_IEC()` macro, which is defined in the device header files. This is helpful because some devices require one cycle of delay for this to take effect, but some require two. A different version of `__write_to_IEC()` will be generated based on device-specific information.

In addition, the processor has a disable interrupt instruction (DISI) that can disable all interrupts for a specified number of instruction cycles. 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);
}
```

16.7 ISR Considerations

The following sections describe how to ensure your interrupt code works as expected.

16.7.1 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.

Development Issues

Here are some examples of the issues that should be considered:

Example 16-1. 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 16-2. 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 16-3. 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 16-4. 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.

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.

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 */

```

16.7.2 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)) _T1Interrupt()
{
    /* 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.

16.7.3 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.

17. Main, Runtime Startup and Reset

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; as well as Reset conditions that must be handled.

17.1 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()`, 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);
```

17.2 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 (see the [8.3 Stack](#) section for more information).
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 [15.1.2 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.

Zipped 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.

You can override the normal startup behavior by defining the function `int _crt_start_mode(void)`. This function should return 0 to indicate that a normal start up procedure is used. Any other return value will indicate

that `preserved` variables should not be initialized. If you have not defined this function, the compiler will always initialize everything.

18. Mixing C and Assembly Code

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:

- [18.1 Mixing Assembly Language and C Variables and Functions](#) – separate assembly language modules may be assembled then linked with compiled C modules.
- [18.2 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).
- [18.3 Predefined Assembly Macros](#) – a list of predefined assembly-code macros to be used in C code is provided.

18.1 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 [14.1 Register Variables](#). In particular, registers W0-W7 are used for parameter passing. An assembly language function will receive parameters and 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`.

Example 18-1. 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 18-2. 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 [15.7.2 Return Value](#). 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.

18.2 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 either be in a register (the usual case) or that it must be an immediate value.

Constraint letters and modifiers supported by the compiler are listed in the following two tables, respectively.

Table 18-1. 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.
C	An even-odd register pair
D	An even-numbered register
T	A near or far data operand.
U	A near data operand.

Table 18-2. 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.

Example 18-3. 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 18-4. 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 18-5. 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 18-6. 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 18-7. 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 18-8. 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 18-9. 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 18-10. 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 18-2](#).

18.3 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 `PWRSAB` instruction with an argument of #0 and #1, respectively.

19. 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 "*MPLAB XC16 Libraries Reference Guide*" (DS5001456) 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 19-1. Using the Math Library

```
#include <math.h>    // declare function prototype for sqrt

void main(void)
{
    double i;

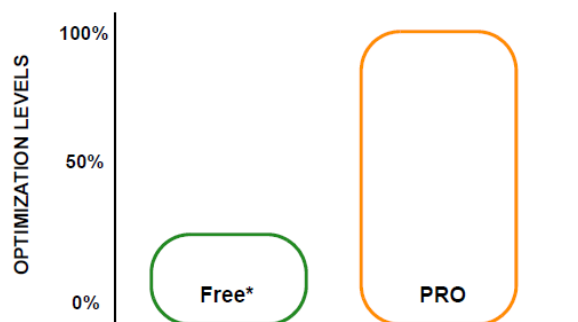
    // sqrt referenced; sqrt will be linked in from library file
    i = sqrt(23.5);
}
```


20. Optimizations

MPLAB XC16 C Compiler license types are Free, EVAL and PRO. The initial compiler download begins as an Evaluation (EVAL) license allows 60 days to evaluate the compiler as a Professional (PRO) license with the most optimizations. The Free license has minimal optimizations. The PRO license can be purchased any time.

Different optimizations may be set ranging from no optimization to full optimization, depending on your compiler license. When debugging code, you may prefer not to optimize your code to ensure expected program flow.

Figure 20-1. Optimization Levels per License



License	Cost	Optimization Options*
Professional (PRO)	Yes	-O0, -O1, -O2, -O3, -Os, -mpa
Free	No	-O0, -O1
Evaluation (EVAL)	No	PRO optimizations enabled for 60 days; afterward reverts to Free optimizations.

* See [7.6.6 Options for Controlling Optimization](#).

20.1 Optimization Feature Summary

Each license supports optimizations equal to specific features. Lists of currently-supported optimization features are shown below. These features are subject to change.

Table 20-1. License Optimization Features

Free	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 • predictive commoning • inline functions • unswitch loops • gcse after reload • tree vectorize • ipa cp clone • Whole-program optimizations

20.2 How to Enable Optimization

MPLAB XC16 is derived from the GCC source-base which provides many different individual switches for controlling optimizations.

It is recommended that you use the 'big O' optimization switches. The more common variants of GCC (Arm, MIPS, ix86, etc.) may update optimization switches more quickly than the dsPIC port, so consequently there may be individual switches that appear and disappear. Sticking with the big O optimizations is a good way to remove surprises.

The big O optimizations are organized into four categories, declared using the `-On` option, where *n* is a numerical value in the range of 0 to 3. Additionally, there is the `-Os` option, which falls somewhere between `-O2` and `-O3`. Free (unlicensed) compilers provide `-O0` and `-O1` options, and PRO licenses add `-O2`, `-O3` and `-Os` options.

In general, the larger the value of *n*, the more optimizations are performed. Some techniques are designed to reduce code size, some are designed to increase the performance of the generated code, and some do both. You will need to determine what is most important for you (speed vs. size trade-offs) and choose the option appropriate for your application.

MPLAB XC16 also supports procedural abstraction, sometimes called function-outlining, via the `-mpa` option. This optimization intends to reduce the code size by abstracting common generated code into individual functions (i.e., the opposite of function inlining). This can reduce code size and is performed on assembly code, post compilation.

Additionally MPLAB XC16 supports `-moptimize-page-setting` which attempts to minimize the effect of DSRPAG swapping with the named address space qualifiers.

20.3 Using Optimizations

The MPLAB XC16 C compiler supports general, as well as several specific, optimization options (see section [7.6.6 Options for Controlling Optimization](#)). In most cases, only general optimization options (`-On`) should be used (for details, see [20.2 How to Enable Optimization](#)).

The more a compiler optimizes the output code, the further away from the C program the code might become. This is the nature of optimization and is often exasperated by a weakness of debug information to represent these changes. Optimizations tend to:

- convert structures into scalar variables (to remove unused members)
- re-order flow, or duplicate it, for speed improvements
- capture results, or partial results, for re-use later
- remove unreachable code
- remove unused variables or promote an object to a register variable
- remove code that has no externally visible effect

Many of these transformations can make debugging code much more difficult. Some of them can turn a “working” program into something that no-longer executes correctly; typically, this means that the working program is not a well-formed C program and the optimizer has exposed this.

20.3.1 Coding for an Optimizing Compiler

How do you get the best out of the optimizer? It turns out the answer to this question is remarkably straight-forward.

Code clearly. The C language can be quite complex. Perhaps you may remember the obfuscated C one-liner competitions that many magazines from the early days of C used to have. These are the opposite of “code clearly.” The chances are that if you can’t read the code, then it is not correct (well-formed) and the optimizer will expose this. Also, some clever C coding tricks will have side-effects that prevent the compiler from doing its best to get concise executable code. Remember, you want the *output* of the compiler to be concise and take advantage of the architecture...not the *input*!

Use well typed objects. It is recommended that you make use of the types defined in `stdint.h` instead of native C types. For example, the fastest way to represent a 8-bit value on a 16-bit device, such as Microchip’s dsPIC line of products, is to use `int_fast8_t` from `stdint.h`. While you can cast (or let the compiler cast) an object to a different type, this may indicate that you should have chosen a better type in the first place. Additionally, the size and signed-ness of the picked type can have an impact on code generation; unfortunately, this is architecture dependent.

Take care using inline assembly (or don’t use it at all). GCC-based compilers, such as MPLAB XC16, have an extended inline assembly format that is provided to communicate how data flows through inline assembly. This allows the compiler to properly optimize the flow of values around assembly statements. This is often a source of “compiler bugs” and the first place to look when investigating whether or not the compiler is broken. Make sure you tell the compiler when writing to a register in inline assembly; the compiler might be using it already! Make sure you tell the compiler if the result of a C expression is used within the inline assembly; if the compiler does not see the use of an expression, it might discard it. Extended GNU inline assembly is described more fully in [18.2 Using Inline Assembly Language](#). Also a rich set of builtin functions are provided that perform many of the tasks for which inline assembly is often used.

Don’t be afraid to use the optimizer. Once you are confident that the code does what you want to do (test early, test often), don’t be afraid to enable optimizations. Optimizations can make it harder to debug, so it is important to take this step once you have tested and have working code.

20.3.2 Help! Optimizing Broke my Code!

If turning the optimizer on has changed the execution behavior of the code, then the following sections may help you to determine your issue and resolve it.

20.3.2.1 Sharing Data Between Different Threads of Execution

Sharing data between different threads of execution (such as between mainline code and an interrupt service routine or between two different threads in a Realtime Operating System like environment) can sometimes be complex.

Make sure that any objects that may be shared in this way are marked as `volatile` (both read or write sharing). `volatile` instructs the compiler to honor all accesses to memory, which will prevent the compiler from caching a value in a register. If the variable is shared, then this is a good thing! The compiler needs to know that the variable might change because of a hardware or other external event, such as in this example where we wish to wait for the buffer to have some data in it before progressing:

```
IOPORT.buffer_empty = 1;
while (IOPORT.buffer_empty);
```

If the object is not marked as `volatile` when optimizing the compiler, then it might determine that the value will never change and do something horrible to the loop or worse. Consider the rest of the code to be unreachable and replace the expression with `while(1);`.

There are times however, when `volatile` is not enough. The compiler may not be able to compute an expression without going through an intermediate register. This means there may be a window of time when a value is stored in a register while the new value is being computed. If it is marked as `volatile`, then it will be written back. This could be the source of data corruption, especially if the object is a single memory location that has many separate data values like a C bitfield. In the following structure example, `status` is a flag set by some external process and `blinky` is a heartbeat in the mainline code.

```
volatile struct corruptable {
    uint16_t status:3;
    uint16_t blinky:1;
} object;

...

while (object.status != 0) {
    object.blinky ^= 1;
}
```

If the compiler has not been able to generate an atomic, uninterruptable sequence to XOR `blinky` then this can be a possible source of corruption. Consider the flow where `status` is updated but the `blinky` update is not complete. Writing back the new value of `blinky`, which shares a word with `status`, might over-write the possibly new value of `status` causing the generated code to never see when `status` has been updated.

If your code is similar to the above example, you can see that `volatile` is not a sufficient solution. Consider coding styles that will prevent this overwrite from occurring, such as not sharing memory in that way and the use of critical-sections to control access of shared data. Often it is efficient and clearer to make use of the object-oriented principal of accessor functions where the access of each object is tightly controlled in one place. A well-defined gating of shared data can allow the code to be written without using `volatile` at all, thus allowing the code to be safe and share data efficiently.

20.3.2.2 Intermixing C and Assembly

If your code has any inline assembly, ensure that the code is written in such a way that the compiler knows about register access and data flow.

For example, the following code will likely cause a failure when optimized.

```
int foo;

int bar() {
    asm("mov _foo,w7");
    asm("inc w7,w0");
    asm("return");
}
```

Though the example could be written in pure C as `return foo+1;`, the correct way to write this in inline assembly requires the use of extended Asm syntax:

```
int bar() {
    int result;

    asm("inc %1,%0" : "=r"(result) : "r"(foo));
    return result;
}
```

This allows the compiler to connect C variables to registers. The compiler will pick an appropriate register and arrange to save the values as needed.

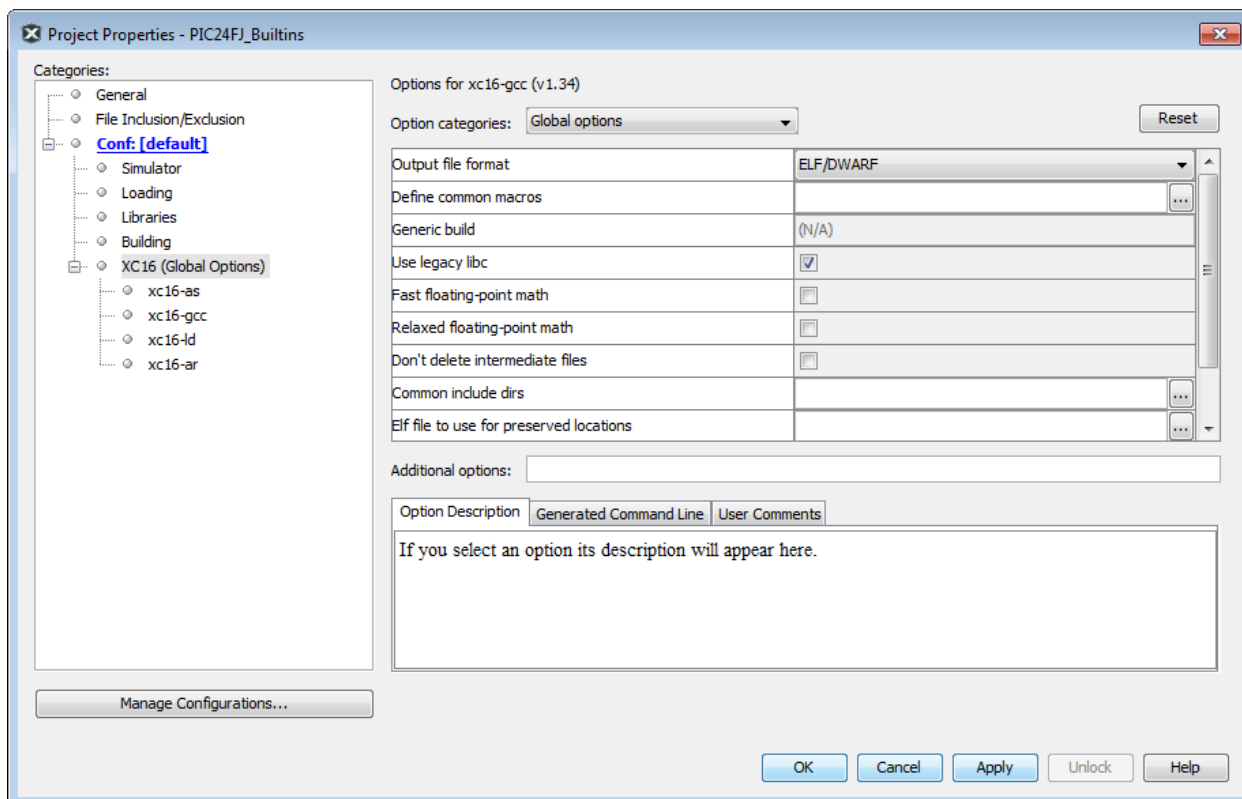
Ensure any assembly function that you write satisfies the rules specified in [18. Mixing C and Assembly Code](#).

20.3.3 Debugging Strategies for Optimized Code

The optimizer can introduce challenges for debugging code which increase with higher levels of optimization. For the best debugging experience, make sure that the ELF/DWARF object file format is selected (as opposed to COFF) whenever possible. The output file format is selected in MPLAB X IDE under *Project Properties>XC16 (Global Options)* (see figure below).

The DWARF symbol language has advanced features that allow the compiler to provide more information when optimized. The compiler will be able to describe how object values flow in and out of registers, even if the register changes. For this reason, ELF/DWARF at `-O1` will provide a reasonably smooth debugging experience with some optimizations.

Figure 20-2. Project Properties XC16 Global Options



Earlier ([20.3 Using Optimizations](#)) we mentioned some of the effects of optimizing code. Some of these effects will prevent the debugger from displaying a value (the variable is not needed and has been optimized away) or placing a breakpoint (the line of code does not exist).

Sometimes it is more effective to debug in a mixed C-assembly display, or to follow the C code along with the Program Memory view.

Additionally, MPLAB XC C compilers provide a couple of tools that can be helpful.

- A variant of the standard C assertion mechanism can be used to return to the debugger at certain execution points. The macro `__conditional_software_breakpoint(X)` is available in `assert.h` and can be used to halt the debugger.
- The optimization level can be set on a function-by-function basis. For example, to make debugging of a particular function easier while still optimizing the rest of the application, define the function like this:
`Tau __attribute__((optimize(1))) fn(...){}`

A declaration of this form will override the current global optimization setting on a function-by-function basis.

- The MPLAB X IDE defines the pre-processor symbol `__DEBUG` when a debug build is being produced. This can be useful for enabling code changes to support debugging only when actually debugging. For example, conditionally changing the optimization level for a given function can be implemented with a simple macro:

```
#ifdef __DEBUG
#define DBG_OPTIMIZE(X) __attribute__((optimize(X)))
#else
#define DBG_OPTIMIZE(X) /* not debugging */
#endif

Tau DBG_OPTIMIZE(1) fn(...) {
}
```

Multiple attributes can be combined. This is valid:

```
void __attribute__((interrupt)) DBG_OPTIMIZE(1) _T1Interrupt(void) {
}
```

21. Preprocessing

All C source files are preprocessed before compilation. The `-E` option can be used to preprocess and then stop the compilation. See [7.6.2 Options for Controlling the Kind of Output](#)

Assembler files can also be preprocessed if the file extension is `.S` rather than `.s`. See [7.1.3 Input File Types](#)

21.1 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>

21.2 Preprocessing Directives

The compiler accepts several specialized preprocessor directives in addition to the standard directives. All of these are listed in the following table.

Table 21-1. 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>
<code>#ifdef</code>	Include source lines if preprocessor symbol defined	<code>#ifdef FLAG do_loop(); #elif SIZE == 5 skip_loop(); #endif</code>

.....continued		
Directive	Meaning	Example
#ifndef	Include source lines if preprocessor symbol not defined	#ifndef FLAG jump(); #endif
#include	Include text file into source	#include <stdio.h> #include "project.h"
#line	Specify line number and file name for listing	#line 3 final
#pragma	Compiler-specific options	#pragma config WDT=OFF
#undef	Undefines preprocessor symbol	#undef FLAG
#warning	Generate a warning message	#warning Length not set

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 pastel(a,b) a##b
#define paste(a,b) pastel(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 the [23.13 Preprocessing Directives](#) section.

21.3 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 the [27. Deprecated Features](#) section.

21.3.1 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.

21.3.2 Compiler Settings Macro

The following symbols are defined if compiler features are enabled.

Table 21-2. Compiler Settings Macros/Symbols

Symbol	Description
<code>__OPTIMIZATION_LEVEL__</code>	Set to the value of the big O number. For example, both <code>-O0</code> and <code>-O2</code> would be set to 2.
<code>__OPTIMIZE_SIZE__</code>	Defined if <code>-Os</code> enabled, undefined otherwise.
<code>__LARGE_ARRAYS__</code>	Set to 1 for <code>-menable-large-arrays</code> , set to 0 otherwise.

21.3.3 Compiler Output Type Macros

The following symbols are defined with the `-ansi` command line option.

Table 21-3. 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>__dsPICC30__</code>	
<code>__XC16ELF</code>	<code>__XC16ELF__</code>	If defined, compiler is producing ELF output.
<code>__C30ELF</code>	<code>__C30ELF__</code>	
<code>__dsPIC30ELF</code>	<code>__dsPIC30ELF__</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 21-4. Macros Defined without `-ansi`

Symbol	Description
<code>XC16</code>	16-bit compiler is in use.
<code>C30</code>	
<code>dsPIC30</code>	

21.3.4 Device Name and Family (Architecture) Macros

The compiler defines a symbol based on the target device set with `-mcpu=`. For example, `-mcpu=30F6014`, which defines the symbol `__dsPIC30F6014__`.

In addition, one of the following symbols is defined for the target device family.

```
__dsPIC30F__
__PIC24FK__
__PIC24F__
__PIC24E__
__PIC24H__
__dsPIC33F__
__dsPIC33E__
__dsPIC33C__
```

21.3.5 Device Features Macros

The following symbols are defined if device features are enabled.

Table 21-5. Device Features Macros/Symbols

Symbol	Description
<code>__HAS_DSP__</code>	Device has a DSP engine

.....continued

Symbol	Description
__HAS_EEDATA__	Device has EEPROM data (EEData) memory
__HAS_DMA__	Device has a DMA controller This is a generic macro which is set if any DMA controller is present. This DOES NOT indicate that DMA memory is present. To determine if there is any DMA memory, use the __DMA_BASE or __DMA_LENGTH manifest constants which should be defined the device header file.
__HAS_DMAV2__	Device has a DMA V2 controller This macro is set if a DMA V2 controller is present. This DOES NOT indicate that DMA memory is present. To determine if there is any DMA memory, use the __DMA_BASE or __DMA_LENGTH manifest constants which should be defined the device header file.
__HAS_CODEGUARD__	Device has CodeGuard™ Security
__HAS_PMP__	Device has Parallel Master Port
__HAS_PMPV2__	Device has Parallel Master Port V2
__HAS_PMP_ENHANCED__	Device has Enhanced Parallel Master Port
__HAS_EDS__	Device has Extended Data Space
__HAS_5VOLTS__	Device is a 5-volt device

21.3.6 Other Macros

The following symbols define other features.

Table 21-6. Other Macros/Symbols

Symbol	Description
__FILE__	Current file name as a C string
__LINE__	Current line number as a decimal integer
__DATE__	Current date as a C string

22. Linking Programs

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 [7.6.9 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* (DS50002106) for an explanation of the detailed information in this file.

22.1 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 [10.10 Variable Attributes](#).

data

General data space. Variables in general data space can be accessed using ordinary C statements. This is the default allocation.

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.

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__`.

eedata - Devices with EEPROM Data (EEData) Memory only

EEData space, a region of 16-bit wide non-volatile memory located at high addresses in program memory. Variables in EEData space cannot 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

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.

22.2 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* (DS50002106). However, you can easily replace a library module that is linked into your program without having to do this.

If you add a source file to your project 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 the [10.10 Variable Attributes](#) section.

22.3 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* (DS50002106) 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);
```

22.4 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 address 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 Files” folder.

23. Implementation-Defined Behavior

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.

23.1 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.

23.2 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

23.3 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.

23.4 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 23-1. 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`.

23.5 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 23-2. 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.

23.6 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 23-3. 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.

23.7 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`.

23.8 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.

23.9 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` bitfield? (ISO 6.5.2.1)

User defined. By default, `signed int` bitfield. 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.

23.10 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.

23.11 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.

23.12 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.

23.13 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 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 23-4. #pragma Behavior

Pragma	Behavior
#pragma code section-name	Names the code section.
#pragma code	Resets the name of the code section to its default (viz., .text).

.....continued	
Pragma	Behavior
#pragma config	Sets configuration bits or registers.
#pragma idata section-name	Names the initialized data section.
#pragma idata	Resets the name of the initialized data section to its default value (viz., .data).
#pragma udata section-name	Names the uninitialized data section.
#pragma udata	Resets the name of the uninitialized data section to its default value (viz., .bss).
#pragma interrupt function-name	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.

23.14 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`, `isctrl`, `islower`, `isprint` and `isupper` functions? (ISO 7.3.1)

Table 23-5. Characters Tested by `is` Functions

Function	Characters tested
<code>isalnum</code>	One of the letters or digits: <code>isalpha</code> or <code>isdigit</code> .
<code>isalpha</code>	One of the letters: <code>islower</code> or <code>isupper</code> .
<code>isctrl</code>	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> .
<code>islower</code>	One of the letters 'a' through 'z'.
<code>isprint</code>	A graphic character or the space character: <code>isalnum</code> or <code>ispunct</code> or space.
<code>isupper</code>	One of the letters 'A' through 'Z'.
<code>ispunct</code>	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.

23.15 Signals

What is the set of signals for the signal function? (ISO 7.7.1.1)

Table 23-6. 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.

23.16 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.

23.17 tmpfile

Is an open temporary file removed if the program terminates abnormally? (ISO 7.9.4.3)

Yes.

23.18 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.

23.19 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.

23.20 abort

*What happens to open and temporary files when the abort function is called?
(ISO 7.10.4.1)*

Nothing.

23.21 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.

23.22 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.

23.23 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.

23.24 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 23-7. 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

24. Embedded Compiler Compatibility Mode

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.

24.1 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 the table below for a list of all keys usable with the MPLAB XC compilers.

Table 24-1. Vendor Keys

Vendor key	Syntax	XC8 Support	XC16 Support	XC32 Support
cci	Common C Interface	Yes	Yes	Yes
iar	IAR C/C++ Compiler™ for ARM	Yes	Yes	Yes

The Common C 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 [4. Common C Interface](#) and are not repeated here.

24.2 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.

The following table indicates the various levels of compatibility used in the tables that are presented throughout this guide.

Table 24-2. 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.

.....continued

Level	Explanation
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.

24.3 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 mode⁽³⁾, as shown in the table below.

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 24-3. 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.

24.4 Operator

The `@` operator may be used with other compilers to indicate the desired memory location of an object. As the following table 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 24-4. Support for Non-standard Operator

IAR Compatibility Mode			
Operator	XC8	XC16	XC32
<code>@</code>	native support	error	error

³ Not all C99 features have been adopted by all Microchip MPLAB XC compilers.

24.5 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 the table below, as well as the level of compatibility offered by MPLAB XC compilers. The table notes offer more information about some extensions.

Table 24-5. 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> ⁴	support ⁵	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> ⁶	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 ⁷	support ⁴
<code>__packed</code>	ignore ⁸	support	support
<code>__root</code>	ignore	support	support

⁴ All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.

⁵ The keyword, `asm`, is supported natively by MPLAB XC8, but this compiler only supports the `__asm` keyword in IAR compatibility mode.

⁶ This is the default (and only) endianism used by all MPLAB XC compilers.

⁷ When used with MPLAB XC32, this must be used with the `__longcall__` macro for full compatibility.

⁸ Although this keyword is ignored, by default, all structures are packed when using MPLAB XC8, so there is no loss of functionality.

.....continued			
IAR Compatibility Mode			
Keyword	XC8	XC16	XC32
<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

24.6 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 the following table.

Table 24-6. Support for Non-Standard Intrinsic Functions

IAR Compatibility Mode			
Function	XC8	XC16	XC32
<code>__disable_fiq⁹</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

The header file `<xc.h>` must be included for supported functions to operate correctly.

24.7 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. The table below 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.

⁹ These intrinsic functions map to macros which disable or enable the global interrupt enable bit on 8-bit PIC[®] devices.

Table 24-7. Support for Non-Standard Pragmas

IAR Compatibility Mode			
Pragma	XC8	XC16	XC32
bitfields	ignore	ignore	ignore
data_alignment	ignore	support	support
diag_default	ignore	ignore	ignore
diag_error	ignore	ignore	ignore
diag_remark	ignore	ignore	ignore
diag_suppress	ignore	ignore	ignore
diag_warning	ignore	ignore	ignore
include_alias	ignore	ignore	ignore
inline	support (no args)	support (no args)	support (no args)
language	ignore	ignore	ignore
location	ignore	support	support
message	support	native support	native support
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

25. Diagnostics

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.

25.1 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 '-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 *x* arguments, but takes just *y*

Too many arguments were passed to macro ‘*name*’.

macro ‘*name*’ requires *y* arguments, but only *z* 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:
`#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 ‘, double quote ” or right angle bracket >.

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.

25.2 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 $X <= Y <= Z$ 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 $X <= Y <= Z$ 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 `noinline`

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 `noinline`

previous declaration of function ‘name’ was inline

Function ‘name’ was declared a second time with the `noinline` 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 noline

The function '*name*' has been declared as inline, but the noline 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 `':'`.

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 >= 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 *0xnumber* 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 *-dletters* 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. 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 (*""*).

26. GNU Free Documentation License

GNU Free Documentation License:

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<www.gnu.org/licenses/fdl-1.3.html>

Copyright (C) 2020 Microchip Technology Inc. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. 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.

1. 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.

2. 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.

3. 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.

4. 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," as well as its Title, and add 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.
- O. 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.

5. 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.”

6. 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.

7. 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.

8. 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.

9. 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.

10. 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 Software 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.

11. 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.

27. Deprecated Features

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.

27.1 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 [21.3 Predefined Macro Names](#).

27.2 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 [18. 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 that the register you specify fits the constraints specified for that operand in the inline assembly statement).

27.2.1 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 each global register variable to 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.

27.2.2 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.

27.3 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.

27.4 Configuration Settings Using Macros

Note: Do not use this deprecated method for setting configuration bits with pragma statements used to set configuration bits (see [8.4 Configuration Bit Access](#)) in the same code.

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 2.7V, and initialize the Power-up timer to 64 ms, 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.

28. Built-in Functions

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.

28.1 Built-In Functions vs. Inline Assembly

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.

28.2 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.

28.2.1 `__builtin_ACCL`, `__builtin_ACCH`, `__builtin_ACCU`

Description

This function can be used to gain access to the low, high, or upper portion of an accumulator value. For example:

```
volatile register int value asm("A");
int result = __builtin_ACCL(value);
```

The example result will be the low 16-bits stored in the accumulator which holds *value*.

These builtins allow access to the parts of an accumulator in a way that is optimizer safe.

Prototype

```
int __builtin_ACCL(int value);
int __builtin_ACCH(int value);
int __builtin_ACCU(int value);
```

Argument

value – Integer number to set accumulator value.

Return Value

Returns the low, high or upper portion of the accumulator.

Assembler Operator/ Machine Instruction

None

Error Messages

None

28.2.2 __builtin_add**Description**

Add value to the accumulator specified by result with a shift specified by literal shift. For example:

```
volatile register int result asm("A");
int value;
result = builtin_add(result,value,0);
```

If value is held in w0, the following will be generated:

```
add w0, #0, A
```

Prototype

```
int __builtin_add(int Accum,int value, const int shift);
```

Argument

Accum – Accumulator to add.

value – Integer number to add to accumulator value.

shift – Amount to shift resultant accumulator value.

Return Value

Returns the shifted addition result to an accumulator.

Assembler Operator/ Machine Instruction

```
add
```

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- argument 0 is not an accumulator
- the shift value is not a literal within range

28.2.3 __builtin_addab**Description**

Add accumulators A and B with the result written back to the specified accumulator.

```
volatile register int result asm("A");
volatile register int B asm("B");

result = __builtin_addab(result,B);
```

will generate:

```
add A
```

Prototype

```
int __builtin_addab(int Accum_a, int Accum_b);
```

Argument

Accum_a – First accumulator to add.

Accum_b – Second accumulator to add.

Return Value

Returns the addition result to an accumulator.

Assembler Operator/ Machine Instruction

```
add
```

Error Messages

An error message will be displayed if the result is not an accumulator register.

28.2.4 __builtin_addr_low, __builtin_addr_high, __builtin_addr**Description**

Determine the offset address of a symbol.

Prototype

```
unsigned int __builtin_addr_low(&symbol);
unsigned int __builtin_addr_high(&symbol);
unsigned int __builtin_addr(&symbol);
```

Argument

&symbol – The literal address of the symbol

Return Value

Returns the low, high or full address of a symbol, without any adjustment for physical address paging requirements. Therefore, the values returned represent a literal offset and cannot be used for addressing purposes without manipulation.

Assembler Operator/ Machine Instruction

addr_low, addr_high, addr

Error Messages

An error message will be displayed if the argument is not a literal address.

28.2.5 __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.

28.2.6 __builtin_clr**Description**

Clear the specified accumulator.

For example:

```
volatile register int result asm("A");
result = __builtin_clr();
```

will generate:

```
clr A
```

Prototype:

```
nt __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.

28.2.7 __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

xptr – Integer pointer to x prefetch

xval – Integer value of x prefetch

xincr – Integer increment value of x prefetch

yptr – Integer pointer to y prefetch

yval – Integer value of y prefetch

yincr – Integer increment value of y prefetch

AWB – Accumulator write back location

AWB_accum – 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

28.2.8 __builtin_clrwdt**Description**

Clear watchdog timer.

Prototype

```
void __builtin_clrwdt(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

clrwdt

Error Messages

None

28.2.9 __builtin_dataflashoffset**Description**

Disable the specified interrupts.

Prototype

```
int __builtin_dataflashoffset(unsigned int &var);
```

Argument

&var = address of pointer to a dataflash variable.

Return Value

Offset value as an integer.

Assembler Operator/ Machine Instruction**Error Messages**

None

28.2.10 __builtin_disable_interrupts**Description**

Disable the specified interrupts.

Prototype

```
void __builtin_disable_interrupts(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

Error Messages

None

28.2.11 __builtin_disi**Description**

Disable all interrupts for a specified number of instruction cycles. See [16.6 Enabling/Disabling Interrupts](#).

Will emit the specified DISI instruction at the point it appears in the source program: `disi #<disi_count>`

Prototype

```
void __builtin_disi(int disi_count);
```

Argument

disi_count instruction cycle count. Must be a literal integer between 0 and 16383.

Return Value

N/A

Assembler Operator/ Machine Instruction

`disi.f`

28.2.12 __builtin_divf

Description

Computes the quotient *num* / *den*. A math error exception occurs if *den* is zero. Function arguments are signed, as is the function result.

Prototype

```
signed int __builtin_divf(signed int num, signed int den);
```

Argument

num – numerator

den – denominator

Return Value

Returns the signed integer value of the quotient *num* / *den*.

Assembler Operator/ Machine Instruction

`div.f`

28.2.13 __builtin_divmodsd

Description

Issues the 16-bit architecture's native signed divide support with the same restrictions given in the “*dsPIC30F/33F Programmer's Reference Manual*” (DS70000157). 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

```
signed int __builtin_divmodsd(
    signed long dividend, signed int divisor,
    signed int *remainder);
```

Argument

dividend – number to be divided

divisor – number to divide by

remainder – pointer to remainder

Return Value

Quotient and remainder.

Assembler Operator/ Machine Instruction

`divmodsd`

Error Messages

None

28.2.14 __builtin_divmodud**Description**

Issues the 16-bit architecture's native unsigned divide support with the same restrictions given in the "*dsPIC30F/33F Programmer's Reference Manual*" (DS70000157). 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

```
unsigned int __builtin_divmodud(
    unsigned long dividend, unsigned int divisor,
    unsigned int *remainder);
```

Argument

dividend – number to be divided

divisor – number to divide by

remainder – pointer to remainder

Return Value

Quotient and remainder.

Assembler Operator/ Machine Instruction

divmodud

Error Messages

None

28.2.15 __builtin_divsd**Description**

Computes the quotient num / den . A math error exception occurs if *den* is zero. Function arguments are signed, as is the function result. The command-line option `-Wconversions` can be used to detect unexpected sign conversions.

Prototype

```
int __builtin_divsd(const long num, const int den);
```

Argument

num – numerator

den – denominator

Return Value

Returns the signed integer value of the quotient num / den .

Assembler Operator/ Machine Instruction

div.sd

28.2.16 __builtin_divud**Description**

Computes the quotient num / den . A math error exception occurs if *den* is zero. Function arguments are unsigned, as is the function result. The command-line option `-Wconversions` can be used to detect unexpected sign conversions.

Prototype

```
unsigned int __builtin_divud(const unsigned long num, const unsigned int den);
```

Argument

num – numerator

den – denominator

Return Value

Returns the unsigned integer value of the quotient *num* / *den*.

Assembler Operator/ Machine Instruction

div.ud

28.2.17 __builtin_dmaoffset**Description**

Obtains the offset of a symbol within DMA memory.

For example:

```
unsigned int result;
char buffer[256] __attribute__((space(dma)));
result = __builtin_dmaoffset(&buffer);
```

Might generate:

```
mov #dmaoffset(buffer), w0
```

Prototype

```
unsigned int __builtin_dmaoffset(const void *p);
```

Argument

**p* – 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.

28.2.18 __builtin_dmapage**Description**

Obtains the page number of a symbol within DMA memory.

For example:

```
unsigned int result;
char buffer[256] __attribute__((space(dma)));

result = __builtin_dmapage(&buffer);
```

Might generate:

```
mov #dmapage(buffer), w0
```

Prototype

```
unsigned int __builtin_dmapage(const void *p);
```

Argument

**p* – pointer to DMA address value

Return Value

Returns the page number of a variable located in DMA memory.

Assembler Operator/ Machine Instruction

dmamapage

Error Messages

An error message will be displayed if the parameter is not the address of a global symbol.

28.2.19 __builtin_ed**Description**

Squares *sqr*, returning it as the result. Also prefetches data for future square operation by computing ***xptra* – ***yptra* and storing the result in **distance*.

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, *ymemory;
int distance;

result = __builtin_ed(distance,
                      &xmemory, 2,
                      &ymemory, 2,
                      &distance);
```

might generate:

```
ed w4*w4, A, [w8]+=2, [W10]+=2, w4
```

Prototype

```
int __builtin_ed(int sqr, int **xptra, int xincr, int **yptra, int yincr, int *distance);
```

Argument

sqr – Integer squared value.

xptr – Integer pointer to pointer to x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yincr – Integer increment value of y prefetch.

distance – 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:

- the result is not an accumulator register
- *xptr* is null
- *yptr* is null

- *distance* is null

28.2.20 `__builtin_edac`

Description

Squares *sqr* and sums with the nominated accumulator register, returning it as the result. Also prefetches data for future square operation by computing `**xptr - **yptr` and storing the result in **distance*.

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, *ymemory;
int distance;

result = __builtin_edac(result, distance,
                        &xmemory, 2,
                        &ymemory, 2,
                        &distance);
```

might generate:

```
edac w4*w4, A, [w8]+=2, [W10]+=2, w4
```

Prototype

```
int __builtin_edac(int Accum, int sqr,
                  int **xptr, int xincr, int **yptr, int yincr,
                  int *distance);
```

Argument

Accum – Accumulator to sum.

sqr – Integer squared value.

xptr – Integer pointer to pointer to x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yincr – Integer increment value of y prefetch.

distance – 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:

- the result is not an accumulator register
- *Accum* is not an accumulator register
- *xptr* is null
- *yptr* is null
- *distance* is null

28.2.21 `__builtin_edsoffset`

Description

Returns the eds page offset of the object whose address is given as a parameter. The argument *p* 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 `space` attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_edsoffset(const void *p);
```

Argument

p – object address

Return Value

Returns the eds page number offset of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

`edsoffset`

Error Messages

The following error message is produced when this function is used incorrectly:

“Argument to `__builtin_edsoffset()` is not the address of an object in extended data space.”

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_edsoffset(&obj);
```

28.2.22 __builtin_edspage**Description**

Returns the eds page number of the object whose address is given as a parameter. The argument *p* 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 `space` attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_edspage(const void *p);
```

Argument

p – object address

Return Value

Returns the eds page number of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

`edspage`

Error Messages

The following error message is produced when this function is used incorrectly: “Argument to `__builtin_edspage()` is not the address of an object in extended data space.”

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_edspage(&obj);
```

28.2.23 __builtin_enable_interrupts**Description**

Enable the specified interrupts.

Prototype

```
void __builtin_enable_interrupts(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction:

Error Messages

None

28.2.24 __builtin_fbcl**Description**

Finds the first bit change from left in value. This is useful for dynamic scaling of fixed-point data. For example:

```
int result, value;
result = __builtin_fbcl(value);
```

might generate:

```
fbcl w4, w5
```

Prototype

```
int __builtin_fbcl(int value);
```

Argument

value – 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

```
fbcl
```

Error Messages

None

28.2.25 __builtin_flim**Description**

Force (Signed) Data Range Limit. Simultaneously compares a 16-bit signed data value to a maximum signed limit value and a minimum signed limit value.

If the data value is greater than the maximum, the data value is set to the maximum value.

If the data value is less than the minimum, the data value is set to the minimum value.

If the data value is within the maximum-minimum values, the data value is not changed.

Prototype

```
int __builtin_flim(int value, int high, int low);
```

Argument

value – Data value

high – Maximum limit value

low – Minimum limit value

Return Value

Returns value clamped between high and low.

Assembler Operator/ Machine Instruction`flim`**Error Messages**

None

28.2.26 __builtin_flim_excess**Description**

Force (Signed) Data Range Limit with Limit Excess Result. Simultaneously compares a 16-bit signed data value to a maximum signed limit value and a minimum signed limit value.

Return the sign of the excess value.

Prototype

```
int __builtin_flim(int value, int high, int low, int *excess):
```

Argument

value – Data value

high – Maximum limit value

low – Minimum limit value

excess – excess over limit

Assembler Operator/ Machine Instruction

```
flim.v (when used with __builtin_flimv_excess)
```

Error Messages

None

28.2.27 __builtin_flimv_excess**Description**

Force (Signed) Data Range Limit with Limit Excess Result. Simultaneously compares a 16-bit signed data value to a maximum signed limit value and a minimum signed limit value.

Return the amount of the excess value.

Prototype

```
int __builtin_flimv_excess(int value, int high, int low, int *excess);
```

Argument

value – Data value

high – Maximum limit value

low – Minimum limit value

excess – excess over limit

Return Value

Return the value of the excess.

Assembler Operator/ Machine Instruction

```
flim.v (when used with __builtin_flimv_excess)
```

Error Messages

None

28.2.28 __builtin_get_isr_state**Description**

Determine the current CPU interrupt state.

Prototype

```
unsigned int __builtin_get_isr_state(void);
```

Argument

None

Return Value

Returns an integer value specifying the current CPU interrupt state.

Assembler Operator/ Machine Instruction

```
get_isr_state
```

Error Messages

None

28.2.29 __builtin_lac**Description**

Shifts value by shift (a literal between -8 and 7) and returns the value to be stored into the accumulator register. For example:

```
volatile register int result asm("A");
int value;
result = __builtin_lac(value,3);
```

Prototype

```
int __builtin_lac(int value, int shift);
```

Argument

value – Integer number to be shifted.

shift – Literal amount to shift.

Return Value

Returns the shifted result to an accumulator.

Assembler Operator/ Machine Instruction

```
lac
```

Error Messages

An error message will be displayed if: the result is not an accumulator register the shift value is not a literal within range

28.2.30 __builtin_lacd**Description**

Shifts a value by *shift* and returns the value to be stored into the accumulator register. For example:

```
volatile register int result asm("A");
long value;
result = __builtin_lacd(value,3);
```

Prototype

```
int __builtin_lacd(long value, unsigned int shift);
```

Argument

value – Long integer number to be shifted.

shift – Literal amount to shift between -16 and 15.

Return Value

Returns the shifted result to an accumulator.

Assembler Operator/ Machine Instruction

None

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- the shift value is not a literal within range

28.2.31 `__builtin_mac`

Description

Computes $a \times b$ and sums with 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 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 *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mac(result, xVal, yVal,
                      &xmemory, &xVal, 2,
                      &ymemory, &yVal, 2, 0, B);
```

might generate:

mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5

Prototype

```
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);
```

Argument

Accum – Accumulator to sum.

a – Integer multiplicand.

b – Integer multiplier.

xptr – Integer pointer to pointer to x prefetch.

xval – Integer pointer to value of x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yval – Integer pointer to value of y prefetch.

yincr – Integer increment value of y prefetch.

AWB – Accumulator write back location.

AWB_accum – Accumulator to write back.

Return Value

Returns the value of accumulator plus the result of a x b.

Assembler Operator/ Machine Instruction

mac

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

28.2.32 `__builtin_modsd`

Description

Issues the 16-bit architecture's native signed divide support with the same restrictions given in the “*dsPIC30F/33F Programmer's Reference Manual*” (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

```
signed int __builtin_modsd(signed long dividend,
signed int divisor);
```

Argument

dividend – number to be divided

divisor – number to divide by

Return Value

Remainder.

Assembler Operator/ Machine Instruction

modsd

Error Messages

None

28.2.33 `__builtin_modud`

Description

Issues the 16-bit architecture's native unsigned divide support with the same restrictions given in the “*dsPIC30F/33F Programmer's Reference Manual*” (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

```
unsigned int __builtin_modud(unsigned long dividend,
unsigned int divisor);
```

Argument

dividend – number to be divided

divisor – number to divide by

Return Value

Remainder.

Assembler Operator/ Machine Instruction

modud

Error Messages

None

28.2.34 __builtin_movsac**Description**

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;

__builtin_movsac(&xmemory, &xVal, 2,
                &ymemory, &yVal, 2, 0, 0);
```

might generate:

```
movsac A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype

```
void __builtin_movsac(
    int **xptr, int *xval, int xincr,
    int **yptr, int *yval, int yincr, int *AWB
    int AWB_accum);
```

Argument

xptr – Integer pointer to pointer to x prefetch.

xval – Integer pointer to value of x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yval – Integer pointer to value of y prefetch.

yincr – Integer increment value of y prefetch.

AWB – Accumulator write back location.

AWB_accum – Accumulator to write back.

Return Value

None

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

28.2.35 `__builtin_mpy`

Description

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 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

a – Integer multiplicand.

b – Integer multiplier.

xptr – Integer pointer to pointer to x prefetch.

xval – Integer pointer to value of x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yval – Integer pointer to value of y prefetch.

yincr – Integer increment value of y prefetch.

AWB – Integer pointer to accumulator selection.

Return Value

Returns the value of *a* x *b*.

Assembler Operator/ Machine Instruction

mpy

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

28.2.36 __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

a – Integer multiplicand.

b – Integer multiplier.

xptr – Integer pointer to pointer to x prefetch.

xval – Integer pointer to value of x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yval – Integer pointer to value of y prefetch.

yincr – Integer increment value of y prefetch.

AWB – 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

28.2.37 __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 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_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

Accum – Accumulator to subtract.

a – Integer multiplicand.

b – Integer multiplier.

xptr – Integer pointer to pointer to x prefetch.

xval – Integer pointer to value of x prefetch.

xincr – Integer increment value of x prefetch.

yptr – Integer pointer to pointer to y prefetch.

yval – Integer pointer to value of y prefetch.

yincr – Integer increment value of y prefetch.

AWB – Accumulator write back location.

AWB_accum – 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

28.2.38 `__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 `-Wconversions` can be used to detect unexpected sign conversions.

For example:

```
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);
```

Prototype

```
signed long __builtin_mulss(const signed int p0, const signed int p1);
```

Argument

p0 – multiplicand

p1 – 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

mul.ss

28.2.39 `__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 `-Wconversions` 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 *p1*.

For example:

```
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);
```

Prototype

```
signed long __builtin_mulsu(const signed int p0, const unsigned int p1);
```

Argument

p0 – multiplicand

p1 – 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

mul.su

28.2.40 __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 `-Wconversions` can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction.

For example:

```
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);
```

Prototype

```
signed long __builtin_mulus(const unsigned int p0, const signed int p1);
```

Argument

p0 – multiplicand

p1 – 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

mul.us

28.2.41 __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 `-Wconversions` 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 *p1*.

For example:

```
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);
```

Prototype

```
unsigned long __builtin_muluu(const unsigned int p0, const unsigned int p1);
```

Argument

p0 – multiplicand

p1 – 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

`mul.uu`

28.2.42 __builtin_nop**Description**

Generates a `nop` instruction.

Prototype

```
void __builtin_nop(void);
```

Argument

None

Return Value

Returns a no operation (`nop`).

Assembler Operator/ Machine Instruction

`nop`

28.2.43 __builtin_popcount**Description**

Count the number of 1's (set bits) in an integer.

Prototype

```
int __builtin_popcount(unsigned int num);
```

Argument

num – integer number

Return Value

Returns the number of 1's found.

Assembler Operator/ Machine Instruction

N/A

28.2.44 __builtin_popcountl**Description**

Count the number of 1's (set bits) in a long integer.

Prototype

```
int __builtin_popcountl(unsigned long num);
```

Argument

num – long integer number

Return Value

Returns the number of 1's found.

Assembler Operator/ Machine Instruction

N/A

28.2.45 __builtin_psvoffset**Description**

Returns the psv page offset of the object whose address is given as a parameter. The argument *p* 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 *space* attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_psvoffset(const void *p);
```

Argument

p – object address

Return Value

Returns the psv page number offset of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

psvoffset

Error Messages

The following error message is produced when this function is used incorrectly:

"Argument to __builtin_psvoffset() is not the address of an object in code, psv, or eedata section".

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_psvoffset(&obj);
```

28.2.46 __builtin_psvpage**Description**

Returns the psv page number of the object whose address is given as a parameter. The argument *p* 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 *space* attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_psvpage(const void *p);
```

Argument

p – object address

Return Value

Returns the psv page number of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

psvpage

Error Messages

The following error message is produced when this function is used incorrectly: “Argument to `__builtin_psvpage()` is not the address of an object in code, psv, or eedata section”.

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_psvpage(&obj);
```

28.2.47 `__builtin_pwrsav`

Description

Enables/disables PIC32 MCU power saving modes.

Prototype

```
void __builtin_pwrsav(unsigned int p);
```

Argument

p 1 = enable, 0 = disable

Return Value

None

Assembler Operator/ Machine Instruction

`pwrsav`

Error Messages

None

28.2.48 `__builtin_readsfr`

Description

Reads the Special Function Register (SFR).

Prototype

```
unsigned int __builtin_readsfr(const void *p);
```

Argument

p – object address

Return Value

Returns the SFR value.

Assembler Operator/ Machine Instruction

`readsfr`

Error Messages

If the object address is not in the range of SFR memory space, an error will be produced. Consult your device data sheet for the memory range.

28.2.49 `__builtin_return_address`

Description

Returns the return address of the current function, or of one of its callers. For the *level* 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

```
int __builtin_return_address (const int level);
```

Argument

level – 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

return_address

28.2.50 __builtin_sac**Description**

Shifts a value by shift and returns the value. For example:

```
volatile register int value asm("A");
long result;
result = __builtin_sac(value,3);
```

Prototype

```
long __builtin_sac(int value, int shift);
```

Argument

value – Integer number to be shifted.

shift – Literal amount to shift between -8 and 7

Return Value

Returns the shifted result.

Assembler Operator/ Machine Instruction

None

Error Messages

An error message will be displayed if: the result is not an accumulator register the shift value is not a literal within range

28.2.51 __builtin_sacd**Description**

Shifts value by shift and returns the value. For example:

```
volatile register int value asm("A");
int result;

result = __builtin_sacd(value,3);
```

Might generate:

sacd A, #3, w0

Prototype

```
int __builtin_sacd(int value, int shift);
```

Argument

value – Integer number to be shifted.

shift – Literal amount to shift between -16 and 15

Return Value

Returns the shifted result.

Assembler Operator/ Machine Instruction

sacd

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- the shift value is not a literal within range

28.2.52 __builtin_sacr**Description**

Shifts value by *shift* and returns the value which is rounded using the rounding mode determined by the CORCONbits.RND control bit.

For example:

```
volatile register int value asm("A");
int result;

result = __builtin_sacr(value,3);
```

Might generate:

```
sac.r A, #3, w0
```

Prototype

```
int __builtin_sacr(int value, int shift);
```

Argument

value – Integer number to be shifted.

shift – Literal amount to shift between -8 and 7

Return Value

Returns the shifted result to CORCON register.

Assembler Operator/ Machine Instruction

sacr

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- the shift value is not a literal within range

28.2.53 __builtin_section_begin, __builtin_section_end**Description**

Get run-time information about a section beginning or ending address.

Prototype

```
unsigned long __builtin_section_begin("section_name");
unsigned long __builtin_section_end("section_name");
```

Argument

section_name – name of the section

Return Value

Returns the beginning or ending address of the named section.

Assembler Operator/ Machine Instruction

section_begin

```
section_end
```

Error Messages

An error message will be displayed if the named section cannot be found.

28.2.54 __builtin_section_size

```
__builtin_section_size
```

Description

Get run-time information about a section's size.

Prototype

```
unsigned long __builtin_section_size("section_name");
```

Argument

section_name – name of the section

Return Value

Returns the size of the named section.

Assembler Operator/ Machine Instruction

```
section_size
```

Error Messages

An error message will be displayed if the named section cannot be found.

28.2.55 __builtin_set_isr_state**Description**

Set the current CPU interrupt state.

Prototype

```
void __builtin_get_isr_state(unsigned int state);
```

Argument

state – Integer value specifying the current CPU interrupt state.

Return Value

None

Assembler Operator/ Machine Instruction

```
set_isr_state
```

Error Messages**28.2.56 __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

28.2.57 __builtin_software_breakpoint**Description**

Provides for a software breakpoint. If a debugger is attached, the IDE will halt. If no debugger is attached, the device will reset.

Prototype

```
void __builtin_software_breakpoint(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

software_breakpoint

Error Messages

None

28.2.58 __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.

28.2.59 __builtin_swap**Description**

For a 16-bit word, swap the bytes in the word; 0x1234 -> 0x3412.

Prototype

```
uint16_t __builtin_swap(uint16_t word);
```

Argument

word – 16-bit word

Return Value

Returns the swapped value.

Assembler Operator/ Machine Instruction

`swap`

Error Messages

None.

28.2.60 __builtin_swap_byte**Description**

For a byte, swap the nibbles in the byte; 0x12 -> 0x21.

Prototype

```
uint8_t __builtin_swap(uint8_t byte);
```

Argument

byte – byte

Return Value

Returns the swapped value.

Assembler Operator/ Machine Instruction

`swap.b`

Error Messages

None.

28.2.61 __builtin_tbladdress**Description**

Returns a value that represents the address of an object in program memory. The argument *p* 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 `space` attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned long __builtin_tbladdress(const void *p);
```

Argument*p* object address**Return Value**Returns an `unsigned long` value that represents the address of an object in program memory.**Assembler Operator/ Machine Instruction**

tbladdress

Error Messages

The following error message is produced when this function is used incorrectly: "Argument to `__builtin_tbladdress()` is not the address of an object in code, psv, or eedata section".

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned long page = __builtin_tbladdress(&obj);
```

28.2.62 __builtin_tbloffset**Description**

Returns the table page offset of the object whose address is given as a parameter. The argument *p* 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 `space` attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_tbloffset(const void *p);
```

Argument*p* – object address**Return Value**

Returns the table page number offset of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

tbloffset

Error Messages

The following error message is produced when this function is used incorrectly:

"Argument to `__builtin_tbloffset()` is not the address of an object in code, psv, or eedata section."

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_tbloffset(&obj);
```

28.2.63 __builtin_tblpage**Description**

Returns the table page number of the object whose address is given as a parameter. The argument *p* 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 `space` attribute in [10.10 Variable Attributes](#).

Prototype

```
unsigned int __builtin_tblpage(const void *p);
```

Argument

p – object address

Return Value

Returns the table page number of the object whose address is given as a parameter.

Assembler Operator/ Machine Instruction

`tblpage`

Error Messages

The following error message is produced when this function is used incorrectly: "Argument to `__builtin_tblpage()` is not the address of an object in code, psv, or eedata section."

The argument must be an explicit object address.

For example, if *obj* is object in an executable or read-only section, the following syntax is valid:

```
unsigned page = __builtin_tblpage(&obj);
```

28.2.64 `__builtin_tblrh`

Description

Issues the `tblrh.w` 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 `__builtin_tbloffset()` and `__builtin_tblpage()`.

Please refer to your device data sheet or Family Reference Manual (FRM) for complete details regarding reading and writing program Flash.

Prototype

```
unsigned int __builtin_tblrh(unsigned int offset);
```

Argument

offset – desired memory offset

Return Value

Contents of the memory address in Flash or EEData memory.

Assembler Operator/ Machine Instruction

`tblrh`

Error Messages

None

28.2.65 `__builtin_tblrld`

Description

Issues the `tblrld.w` 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 `__builtin_tbloffset()` and `__builtin_tblpage()`.

Please refer to your device data sheet or Family Reference Manual (FRM) for complete details regarding reading and writing program Flash.

Prototype

```
unsigned int __builtin_tblrld(unsigned int offset);
```

Argument

offset – desired memory offset

Return Value

Contents of the memory address in Flash or EEData memory.

Assembler Operator/ Machine Instruction

tblrdl
Error Messages

None

28.2.66 __builtin_tblwth**Description**

Issues the `tblwth.w` 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 `__builtin_tbloffset()` and `__builtin_tblpage()`.

Please refer to your device data sheet or Family Reference Manual (FRM) for complete details regarding reading and writing program Flash.

Prototype

```
void __builtin_tblwth(unsigned int offset
unsigned int data);
```

Argument

offset – desired memory offset

data – data to be written

Return Value

None

Assembler Operator/ Machine Instruction

tblwth

Error Messages

None

28.2.67 __builtin_tblwtl**Description**

Issues the `tblrdl.w` 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 `__builtin_tbloffset()` and `__builtin_tblpage()`.

Please refer to your device data sheet or Family Reference Manual (FRM) for complete details regarding reading and writing program Flash.

Prototype

```
void __builtin_tblwtl(unsigned int offset
unsigned int data);
```

Argument

offset – desired memory offset

data – data to be written

Return Value

None

Assembler Operator/ Machine Instruction

tblwtl

Error Messages

None

28.2.68 __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 `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_CRYOTP(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

```
mov    #0x55, Wn
mov    Wn, _CRYKEY
mov    #0xAA, Wn
mov    Wn, _CRYKEY
bset   _CRYCON, #0
nop
nop
```

Error Messages

None

28.2.69 __builtin_write_DISICNT**Description**

Enables the Flash for writing by issuing the correct unlock sequence and enabling the Write bit of the DISICNT register.

Interrupts may need to be disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_DISICNT(DISI_save);
```

Argument

DISI_save - Specified value to save to DISICNT register

Return Value

None

Assembler Operator/ Machine Instruction**Error Messages**

None

28.2.70 __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 disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_NVM(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

```
mov    #0x55, Wn
mov    Wn, _NVMKEY
mov    #0xAA, Wn
mov    Wn, _NVMKEY
bset   _NVMCON, #15
nop
nop
```

Error Messages

None

28.2.71 __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. After completion, the two keys are cleared to zero.

Interrupts may need to be disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_NVM_secure(unsigned int key1, unsigned int key2);
```

Argument

key1 – first key in the NVM unlock sequence

key2 – second key in the NVM unlock sequence

Return Value

None

Assembler Operator/ Machine Instruction

Depending on the location of the keys:

```
mov    W0, Wn
mov    Wn, _NVMKEY
mov    W1, Wn
mov    Wn, _NVMKEY
bset   _NVMCON, #15
nop
nop
```

Error Messages

None

28.2.72 __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 your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_OSCCONH(unsigned char value);
```

Argument

value – character to be written

Return Value

None

Assembler Operator/ Machine Instruction*

```
mov    #0x78, w0
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.

28.2.73 __builtin_write_OSCCONL

Description

Unlocks and writes its argument to OSCCONL.

Interrupts may need to be disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_OSCCONL(unsigned char value);
```

Argument

value – character to be written

Return Value

None

Assembler Operator/ Machine Instruction*

```
mov    #0x46, w0
mov    #0x57, w1
mov    __OSCCON, w2
mov.b  w0, [w2]
mov.b  w1, [w2]
mov.b  value, [w2]
```

Error Messages

None

* The exact sequence may be different.

28.2.74 __builtin_write_PWMSFR**Description**

Writes the PWM unlock sequence to the SFR pointed to by *PWM_KEY* and then writes *value* to the SFR pointed to by *PWM_sfr*

Prototype

```
void __builtin_write_PWMSFR(volatile unsigned int *PWM_sfr,
unsigned int value, volatile unsigned int *PWM_KEY);
```

Argument

PWM_sfr – register to be written

value – value to write

PWM_KEY – hardware unlock key location

Return Value

None

Assembler Operator/ Machine Instruction

```
mov    #PWM_KEY, w3
mov    #value, w2
mov    #0x4321, w1
mov    #0xABCD, w0
mov    w1, [w3]
mov    w0, [w3]
mov    w2, [w3]
```

Error Messages

None

Examples

Example 1:

```
__builtin_write_PWMSFR(&PWM1CON1, 0x123, &PWM1KEY);
```

Example 2:

```
__builtin_write_PWMSFR(&P1FLTACON, 0x123, &PWMKEY);
```

The choice of *PWM_KEY* may depend upon architecture.

28.2.75 __builtin_write_RPCON**Description**

Initiates a write to RPCON register by issuing the correct unlock sequence and setting the RPCON register. Interrupts may need to be disabled for proper operation.

Prototype

```
void __builtin_write_RPCON(unsigned int value);
```

Argument

value – Specified value to save to RPCON register

Return Value

None

Assembler Operator/Machine Instruction

```
mov    #0x55, Wn
mov    Wn, _NVMKEY
mov    #0xAA, Wn
```

```

mov    Wn, _NVMKEY
mov    value, _RPCON

```

Error Messages

None

28.2.76 __builtin_write_RTCWEN**Description**

Used to write to the RTCC Timer by implementing the unlock sequence by writing the correct unlock values to NVMKEY, and then setting the RTCWREN bit of RCFGAL SFR. Interrupts may need to be disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_RTCWEN(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

```

mov    #0x55, w0
mov    w0, _NVMKEY
mov    #0xAA, w0
mov    w0, _NVMKEY
bset   _RCFGAL, #13
nop
nop

```

Error Messages

None

28.2.77 __builtin_write_RTCC_WRLOCK**Description**

Used to write to the RTCC Timer by implementing the unlock sequence by writing the correct unlock values to NVMKEY, and then setting the RTCWREN bit of RCFGAL SFR. Interrupts may need to be disabled for proper operation.

This `builtin` function can be used as a part of a complex sequence discussed in your device data sheet or Family Reference Manual (FRM). See these documents for more information.

Prototype

```
void __builtin_write_RTCC_WRLOCK(void);
```

Argument

None

Return Value

None

Assembler Operator/ Machine Instruction

```

mov    #0x55, w0
mov    w0, _NVMKEY
mov    #0xAA, w0
mov    w0, _NVMKEY

```



```
bclr _RTCCON1L, #11  
nop  
nop
```

Error Messages

None

29. Document Revision History

Revision A (April 2012)

Initial revision of the document.

Revision B (July 2012)

- Chapter 2. "Common C Interface." was added.
- Figure 4-2 "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 8.10.4 "`__pack__`".
- Added DSRPAG/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 (September 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 19.4.3 "Compiler Output Type 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”.

Revision E (December 2014)

- Throughout - Remove mention of MPLAB IDE v8.xx, except where necessary.
- Preface - Update to add “How To’s” chapter reference and remove “XC16 Toolchain and MPLAB IDE v8” chapter reference.
- Section 2.5.10 “Interrupt Functions” - corrected a function.
- Section 4.2 “MPLAB X IDE and Tools Installation” - updated the licensing information.
- Section 4.5 “Project Setup” - updated compiler options in MPLAB X IDE.
- Section 5.4.1.2 “User-Defined Libraries” - added information on contents.
- Section 5.7.1 “Options Specific to 16-Bit Devices” - added `--partition` option for dual partition devices.
- Section 5.7.4 “Options for Controlling Warnings and Errors” - split into subsections. Took information from the table and made it into a subsection for `-W`.
- Section 5.7.6 “Options for Controlling Optimization” - split into subsections. Added info to make a subsection for `--function-section` option. Added a cross-reference from Section 14.3 “Writing an Interrupt Service Routine”.
- Section 6.3.3 “Compile Time Memory Information” - added section Added `dataflash` argument to that `space` attribute.
- Section 10.3.1 “Auto and Non-Auto Variables vs. Local and Global Variables” section created from the last two paragraphs of Section 10.3.
- Section 13.2.2 “Function Attributes” and Section 14.5.2 “context Attribute” - added information for the `context` attribute.
- Section 13.8 “Function Call Conventions” - updated the table for EDS pointer requirements.
- Section 14.3.3 “Coding ISRs”, Section 14.4.1 “Interrupt Vector Usage” and Section 14.5.1 “Assembly and ISRs” - updated code snippets.
- Section 14.4 “Specifying the Interrupt Vector” - added information about movable alternate interrupt tables, and split remaining text into two subsections.
- Section 16.3 “Using Inline Assembly Language” - added compiler constraint letters.
- Section 19.4.5 “Device Features Macros” - clarified `__HAS_DMA__` macros.
- Appendix G. “Built-in Functions.” - added `__builtin_write_NVM_secure` and `__builtin_software_breakpoints`; updated `__builtin_enable_interrupts` and `__builtin_disable_interrupts`.

Revision F (July 2016)

- Updated DS numbers for XC16 ASM/LINK user's guide and 16-bit libraries.
- Optimization information updated per license - Chapter 18. “Optimizations.”, Section 4.5.3 “xc16-gcc (16-Bit C Compiler)” (Table 4-6), and Section 5.7.6 “Options for Controlling Optimization”
- Section 6.5 “Configuration Bit Access” - Added “Configuration Settings Using Macros” moved to Appendix F. “Deprecated Features.”
- Complex numbers not supported so removed references to support (Section 8.8).
- Section 10.14 “Co-resident Applications” - Co-resident applications information and reference.
- Section 14.7 “Enabling/Disabling Interrupts” - Built-in name corrected, `__write_to_IEC()` documented.
- Section 19.4.6 “Other Macros” - Updated `__LINE__` description.
- Appendix G. “Built-in Functions.” - Added `__builtin_addr_low`, `__builtin_addr_high`, `__builtin_addr`, `__builtin_clrwdt`, `__builtin_lacd`, `__builtin_sacd`, `__builtin_ACCL`, `__builtin_ACCH`, `__builtin_ACCU`, `__builtin_write_DISCNT`, `__builtin_pwrsav`. Updated `__builtin_movsac`, `__builtin_sacr`, `__builtin_write_RTCC_WRLock` - replaces `__builtin_write_RTCWEN`.

Revision G (February 2018)

- Removed reference to obsolete Standard (STD) license.
- Section 3.3.7 “How Do I Build Libraries?” - updated for new 5.4.1.3.

- Section 3.4.2.5 “Are There Any SFRs Usage Considerations?” added section.
- Section 3.6.2 “Why Can’t I Debug my Code after I Optimize?” - updated for new 18.4.
- Section 3.6.6 “What are the Speed vs. Size Tradeoffs?” added section.
- Section 4.5.3 “xc16-gcc (16-Bit C Compiler)” - Table 4-5 updated and footnotes added, -mnear-char removed. Table 4-6 updated to remove-mno-override-inline.
- Section 4.5.4 “xc16-ld (16-Bit Linker)” - corrected definition for “Use Local Stack”.
- Section 5.2.1 “Drive Command-Line Format” - added linker script to command-line example to avoid inconsistent warnings depending on device.
- Section 5.4.1.2 “User-Defined Libraries” - moved some content to 5.4.1.3.
- Section 5.4.1.3 “User-Defined Libraries Development” - added section.
- Section 5.7.1 “Options Specific to 16-Bit Devices” - added -mno-eds-warn, -mno-file, -moptimize-page-setting, -mlegacy-libc, -mprint-builtins, -mprint-devices, -mprint-mchp-search-dirs, -mno-errata, -msmart-io-format, -msfr-warn.
- Section 5.7.6.3 “Options that Specify Machine-Independent Flags” - added -fnofallback.
- Section 5.7.4.1 “Options to Control the Amount and Types of Warnings” - remove from Table 5-8 -pedantic, -pedantic-errors and -Wunused-parameter.
- Section 5.7.4.2 “Options that are not Implied by -Wall” - added -Wextra; fixed -Wlarger-than=len.
- Section 5.7.7 “Options for Controlling the Preprocessor” - added -iquote.
- Section 6.3.2 “Device Support Information” - added section.
- Section 6.8 “Using EDS” - added section.
- Section 8.11 “Variable Attributes” - first paragraph updated; clarified persistent attribute usage.
- Section 13.2.2 “Function Attributes” - added optimize attribute.
- Section 14.4 “Specifying the Interrupt Vector” - changed to AIVTDIS = ON.
- Section 18.3 “How to Enable Optimization” - added section.
- Section 18.4 “Using Optimizations” - added section.
- Section 19.4.6 “Other Macros” - __LINE__ macro description corrected.
- Appendix G. “Built-in Functions.” - __builtin_write_RTCC_WRLock does not replace __builtin_write_RTCWEN, corrected; __builtinindisi corrected to __builtin_disi; __builtin_movsac and __builtin_sac return value corrected;

Revision H (November 2018)

- Microchip website addresses now using https instead of http.
- Section 3.6.20 “How Do I Stop My Project’s Checksum From Changing?” - added section to “How To’s” chapter.
- Section 10.15 “Memory Models” - Table 10-1 content updated.
- Section 13.7 “Memory Models” - “Near and Far Code” content moved to Section 10.15 “Memory Models” and a reference to this section added.

Revision J (December 2019)

- The guidance for using -mlarge-arrays has been changed to “used when allocating arrays greater than or equal to 32K” in Section 5.7.1 “Options Specific to 16-Bit Devices” and Section 10.3.2.3 “Non-Auto Variable Size Limits”
- Section 5.7.1 “Options Specific to 16-Bit Devices”: Added option -mcodecov for MPLAB Code Coverage.
- Section 13.8 “Function Call Conventions”: In the second bullet, “caller” changed to “callee.”
- Appendix G. “Built-in Functions.”: For __builtin_lacd and __builtin_sacd, the argument “shift” value is changed from [-16:15] to [-8:7].

Revision K (June 2020)

- Section 19.4 “Predefined Macro Names”: Added macros: __OPTIMIZATION_LEVEL__, __OPTIMIZE_SIZE__, __LARGE_ARRAYS__, __HAS_AUXFLASH__, and eight representing __DeviceFamily__.

- Appendix G. “Built-in Functions.”: Added built-in functions: `__builtin_write_RPCON`, `__builtin_flim`, `__builtin_flim_excess`, `__builtin_flimv_excess`, `__builtin_swap`, `__builtin_popcount`, `__builtin_popcountl`.

Revision M (February 2021)

- Section 8.8 “Stack Usage Guidance”: Added section on Stack Usage Guidance analytic tool for use on command line and in MPLAB X IDE.
- Section 15.7 “Function Call Conventions”: Fixed typo. DBRPAG changed to DSRPAG.
- Section 15.1.2 “Function Attributes”: Update that attributes `auto_psv` and `no_auto_psv` can be combined with attributes `boot` and `secure` in addition to `interrupt`.

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **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 design partner 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

Product Change Notification Service

Microchip's product change notification service helps keep customers current on Microchip products. Subscribers will receive email notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Product Identification System

To order or obtain information, e.g., on pricing or delivery, refer to the factory or the listed sales office.

PART NO.
Device [X]⁽¹⁾ - X /XX XXX
Tape and Reel Temperature Package Pattern
Option Range

Device:	PIC16F18313, PIC16LF18313, PIC16F18323, PIC16LF18323	
Tape and Reel Option:	Blank	= Standard packaging (tube or tray)
	T	= Tape and Reel ⁽¹⁾
Temperature Range:	I	= -40°C to +85°C (Industrial)
	E	= -40°C to +125°C (Extended)
Package: ⁽²⁾	JQ	= UQFN
	P	= PDIP
	ST	= TSSOP
	SL	= SOIC-14
	SN	= SOIC-8
	RF	= UDFN
Pattern:	QTP, SQTP, Code or Special Requirements (blank otherwise)	

Examples:

- PIC16LF18313- I/P Industrial temperature, PDIP package
- PIC16F18313- E/SS Extended temperature, SSOP package

Notes:

1. Tape and Reel identifier only appears in the catalog part number description. This identifier is used for ordering purposes and is not printed on the device package. Check with your Microchip Sales Office for package availability with the Tape and Reel option.
2. Small form-factor packaging options may be available. Please check www.microchip.com/packaging for small-form factor package availability, or contact your local Sales Office.

Microchip Devices Code Protection Feature

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

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods being used in attempts to breach the code protection features of the Microchip devices. We believe that these methods require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Attempts to breach these code protection features, most likely, cannot be accomplished without violating Microchip's intellectual property rights.
- Microchip is willing to work with any customer who is concerned about the integrity of its code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is "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.

Legal Notice

Information contained in this publication is provided for the sole purpose of designing with and using Microchip products. Information 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.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. 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 ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, COST OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION. 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 unless otherwise stated.

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, FlashTec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICKit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SMART-I.S., storClad, SQL, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA 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.

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, and Symmcom are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark 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.

© 2021, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

ISBN: 978-1-5224-7644-3

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamiQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Quality Management System

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

Worldwide Sales and Service

AMERICAS	ASIA/PACIFIC	ASIA/PACIFIC	EUROPE
Corporate Office 2355 West Chandler Blvd. Chandler, AZ 85224-6199 Tel: 480-792-7200 Fax: 480-792-7277 Technical Support: www.microchip.com/support Web Address: www.microchip.com	Australia - Sydney Tel: 61-2-9868-6733 China - Beijing Tel: 86-10-8569-7000 China - Chengdu Tel: 86-28-8665-5511 China - Chongqing Tel: 86-23-8980-9588 China - Dongguan Tel: 86-769-8702-9880 China - Guangzhou Tel: 86-20-8755-8029 China - Hangzhou Tel: 86-571-8792-8115 China - Hong Kong SAR Tel: 852-2943-5100 China - Nanjing Tel: 86-25-8473-2460 China - Qingdao Tel: 86-532-8502-7355 China - Shanghai Tel: 86-21-3326-8000 China - Shenyang Tel: 86-24-2334-2829 China - Shenzhen Tel: 86-755-8864-2200 China - Suzhou Tel: 86-186-6233-1526 China - Wuhan Tel: 86-27-5980-5300 China - Xian Tel: 86-29-8833-7252 China - Xiamen Tel: 86-592-2388138 China - Zhuhai Tel: 86-756-3210040	India - Bangalore Tel: 91-80-3090-4444 India - New Delhi Tel: 91-11-4160-8631 India - Pune Tel: 91-20-4121-0141 Japan - Osaka Tel: 81-6-6152-7160 Japan - Tokyo Tel: 81-3-6880-3770 Korea - Daegu Tel: 82-53-744-4301 Korea - Seoul Tel: 82-2-554-7200 Malaysia - Kuala Lumpur Tel: 60-3-7651-7906 Malaysia - Penang Tel: 60-4-227-8870 Philippines - Manila Tel: 63-2-634-9065 Singapore Tel: 65-6334-8870 Taiwan - Hsin Chu Tel: 886-3-577-8366 Taiwan - Kaohsiung Tel: 886-7-213-7830 Taiwan - Taipei Tel: 886-2-2508-8600 Thailand - Bangkok Tel: 66-2-694-1351 Vietnam - Ho Chi Minh Tel: 84-28-5448-2100	Austria - Wels Tel: 43-7242-2244-39 Fax: 43-7242-2244-393 Denmark - Copenhagen Tel: 45-4485-5910 Fax: 45-4485-2829 Finland - Espoo Tel: 358-9-4520-820 France - Paris Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79 Germany - Garching Tel: 49-8931-9700 Germany - Haan Tel: 49-2129-3766400 Germany - Heilbronn Tel: 49-7131-72400 Germany - Karlsruhe Tel: 49-721-625370 Germany - Munich Tel: 49-89-627-144-0 Fax: 49-89-627-144-44 Germany - Rosenheim Tel: 49-8031-354-560 Israel - Ra'anana Tel: 972-9-744-7705 Italy - Milan Tel: 39-0331-742611 Fax: 39-0331-466781 Italy - Padova Tel: 39-049-7625286 Netherlands - Drunen Tel: 31-416-690399 Fax: 31-416-690340 Norway - Trondheim Tel: 47-72884388 Poland - Warsaw Tel: 48-22-3325737 Romania - Bucharest Tel: 40-21-407-87-50 Spain - Madrid Tel: 34-91-708-08-90 Fax: 34-91-708-08-91 Sweden - Gothenberg Tel: 46-31-704-60-40 Sweden - Stockholm Tel: 46-8-5090-4654 UK - Wokingham Tel: 44-118-921-5800 Fax: 44-118-921-5820