MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide

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

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

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

Trademarks

ISBN: 978-1-61341-294-7

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2009

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Table of Contents

Preface

NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site (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® IDE on-line help. Select the Help menu, and then Topics to open a list of available on-line help files.

INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs. Items discussed include:

- Document Layout
- Conventions Used in this Guide
- Recommended Reading
- The Microchip Web Site
- Development Systems Customer Change Notification Service
- Customer Support

DOCUMENT LAYOUT

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

- **Chapter 1: Compiler Overview** – describes the compiler, development tools and feature set.
- **Chapter 2: Differences between 16-Bit Device C and ANSI C** – describes the differences between the C language supported by the compiler syntax and the standard ANSI-89 C.
- **Chapter 3: Using the Compiler on the Command Line** – describes how to use the compiler from the command line.
- **Chapter 4: Run Time Environment** – describes the compiler run-time model, including information on sections, initialization, memory models, the software stack and much more.
- **Chapter 5: Data Types** – describes the compiler integer, floating point and pointer data types.
- **Chapter 6: Additional C Pointers** – describes additional C pointers available.
- **Chapter 7: Device Support Files** – describes the compiler header and register definition files, as well as how to use with SFRs.
- **Chapter 8: Interrupts** – describes how to use interrupts.
- **Chapter 9: Mixing Assembly Language and C Modules** – provides guidelines to using the compiler with 16-bit assembly language modules.
- **Appendix A: Implementation-Defined Behavior** – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- **Appendix B: Built-in Functions –** lists the built-in functions of the C compiler.
- **Appendix C: Diagnostics** – lists error and warning messages generated by the compiler.
- **Appendix D: MPLAB C Compiler for PIC18 MCUs vs. 16-Bit Devices** – highlights the differences between the PIC18 MCU C compiler and the 16-bit C compiler.
- **Appendix E: Deprecated Features –** details features that are considered obsolete.
- **Appendix F: ASCII Character Set** – contains the ASCII character set.
- **Appendix G: GNU Free Documentation License** – usage license for the Free Software Foundation.

CONVENTIONS USED IN THIS GUIDE

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* 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	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	Sample source code	`#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	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options]` *`file`* `[options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`
Sidebar Text		
STD	Standard edition only. This feature supported only in the standard edition of the software, i.e., not supported in standard evaluation (after 60 days) or lite editions.	`-mpa` option
DD	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	`xmemory` attribute

RECOMMENDED READING

This documentation describes how to use the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

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

16-Bit Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools for 16-bit devices. Examples using the 16-bit simulator SIM30 (a component of MPLAB SIM) are provided.

MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide (DS51317)

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

16-Bit Language Tools Libraries (DS51456)

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

Device-Specific Documentation

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

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

C Standards Information

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability and efficient execution of C language programs on a variety of computing systems.

C Reference Manuals

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

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

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

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

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

THE MICROCHIP WEB SITE

Microchip provides online support via our web site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

Microchip's customer notification service helps keep customers current on Microchip products. Subscribers will receive e-mail notification whenever there are changes, updates, revisions or errata related to a specified product family or development tool of interest.

To register, access the Microchip web site at www.microchip.com, click on Customer Change Notification and follow the registration instructions.

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. These include the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD 2 and 3 in-circuit debuggers and PICkit™ 2 and 3 debug express.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the device (production) programmers MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger, MPLAB PM3, and PRO MATE II and development (nonproduction) programmers MPLAB ICD 2 in-circuit debugger, PICSTART® Plus and PICkit 1, 2 and 3.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://support.microchip.com

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The dsPIC® family of Digital Signal Controllers (dsPIC30F and dsPIC33F DSCs) combines the high performance required in DSP applications with standard microcontroller features needed for embedded applications. PIC24 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 microcontrollers intended for applications that do not require the power of the DSC capabilities.

All of these devices are fully supported by a complete set of software development tools, including an optimizing C compiler, an assembler, a linker and an archiver/librarian.

This chapter provides an overview of these tools and introduces the features of the optimizing C compiler, including how it works with the assembler and linker. The assembler and linker are discussed in detail in the "*MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317).

1.2 HIGHLIGHTS

Items discussed in this chapter are:

- Compiler Description and Documentation
- Compiler and Other Development Tools
- Compiler Feature Set

1.3 COMPILER DESCRIPTION AND DOCUMENTATION

There are three Microchip compilers that support various Microchip 16-bit devices. Also, each one of these compilers comes in different editions, which support different levels of optimization.

	MPLAB® C Compiler for	Device Support	Edition Support
1	PIC24 MCUs and dsPIC® DSCs	All 16-bit devices	Std, Std Eval
2	dsPIC DSCs	dsPIC30F/33F DSCs	Std, Std Eval, Lite
3	PIC24 MCUs	PIC24F/H MCUs	Std, Std Eval, Lite

Each compiler is an ANSI x3.159-1989-compliant, optimizing C compiler. Each compiler is a Windows® console application that provides a platform for developing C code. Each compiler is a port of the GCC compiler from the Free Software Foundation.

The first and second compilers include language extensions for dsPIC DSC embedded-control applications.

1.3.1 Compiler Editions

Each of the three compilers in **Section 1.3 "Compiler Description and Documentation"** come in one or more of the following editions:

- Standard (Purchased Compiler) – All optimization levels enabled.
- Standard Evaluation (Free) – All optimization levels enabled for 60 days, but then reverts to optimization level 1 only.
- Lite (Free) – Optimization level 1 only.

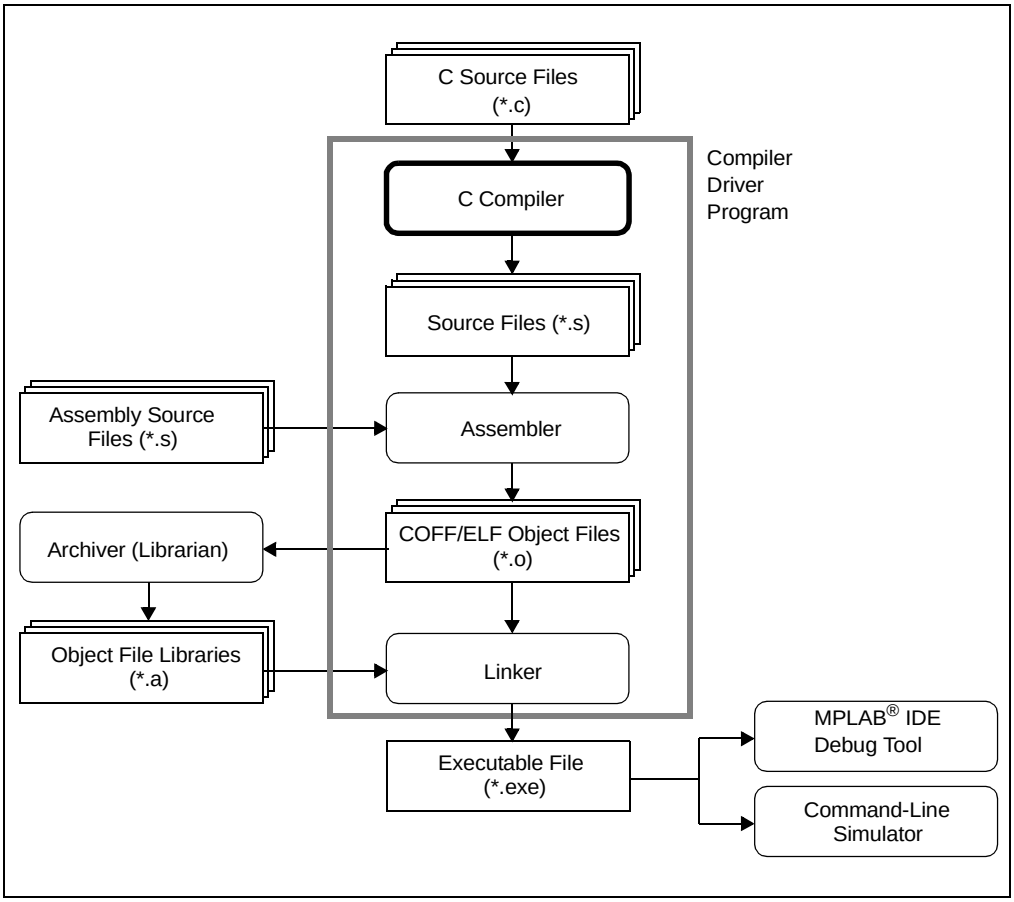
1.3.2 Compiler Documented in this Manual

This manual describes the standard edition of the Standard (purchased) compiler, since the Standard Evaluation and Lite compilers are subsets of the first. Features that are unique to specific devices, and therefore specific compilers, are noted with "DD" text the column (see the **Preface**) and text identifying the devices to which the information applies.

1.4 COMPILER AND OTHER DEVELOPMENT TOOLS

The MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs 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 COFF or ELF file format. The COFF or ELF file can be loaded into the MPLAB IDE, where it can be tested and debugged, or the conversion utility can be used to convert the COFF or ELF file to Intel® hex format, suitable for loading into the command-line simulator or a device programmer. See Figure 1-1 for an overview of the software development data flow.

FIGURE 1-1: SOFTWARE DEVELOPMENT TOOLS DATA FLOW



1.5 COMPILER FEATURE SET

The 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. This section describes key features of the compiler.

1.5.1 ANSI C Standard

The compiler is a fully validated compiler that conforms to the ANSI C standard as defined by the ANSI specification 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.

1.5.2 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.

1.5.3 ANSI Standard Library Support

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

1.5.4 Flexible Memory Models

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

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

1.5.5 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 (see Figure 1-1).

Chapter 2. Differences Between 16-Bit Device C and ANSI C

2.1 INTRODUCTION

This section discusses the differences between the C language supported by MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB C30) syntax and the 1989 standard ANSI C.

2.2 HIGHLIGHTS

Items discussed in this chapter are:

- Keyword Differences
- Statement Differences
- Expression Differences

2.3 KEYWORD DIFFERENCES

This section describes the keyword differences between plain ANSI C and the C accepted by the 16-bit device compiler. The new keywords are part of the base GCC implementation, and the discussion in this section is 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
- Specifying Attributes of Functions
- Inline Functions
- Variables in Specified Registers
- Complex Numbers
- Double-Word Integers
- Referring to a Type with `typeof`

2.3.1 Specifying Attributes of Variables

The compiler keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently supported for variables:

- `address (addr)`
- `aligned (alignment)`
- `boot`
- `deprecated`
- `eds`
- `fillupper`
- `far`
- `mode (mode)`
- `near`
- `noload`
- `page`
- `packed`
- `persistent`
- `reverse (alignment)`
- `section ("section-name")`
- `secure`
- `sfr (address)`
- `space (space)`
- `transparent_union`
- `unordered`
- `unused`
- `weak`

You may also specify attributes with `__` (double underscore) preceding and following each keyword (e.g., `__aligned__` instead of `aligned`). This allows you to use them in header files without being concerned about a possible macro of the same name.

To specify multiple attributes, separate them by commas within the double parentheses, for example:

```
__attribute__ ((aligned (16), packed)).
```

> **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.

```
int var __attribute__ ((address(0x800)));
```

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; but 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, for extended data space, attribute indicates to the compiler that the variable will may be allocated anywhere within data memory. Variables with this attribute will likely also need the __eds__ type qualifier (see **Chapter 6. "Additional C Pointer Types"**) in order for the compiler to properly generate the correct access sequence. Note 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 };
```

page

The `page` attribute places variable definitions into a specific page of memory. The page size depends on the type of memory selected by a `space` attribute. Objects residing in RAM will be constrained to a 32K page while objects residing in Flash will be constrained to a 64K page (upper byte not included).

```
unsigned int var[10] __attribute__ ((space(auto_psv)));
```

The `space(auto_psv)` or `space(psv)` attribute will use a single memory page by default.

```
__eds__ unsigned int var[10] __attribute__ ((space(eds), page));
```

When dealing with `space(eds)`, please refer to **Chapter 6. "Additional C Pointer Types"** for more information.

packed

The `packed` attribute specifies that a structure member should have the smallest possible alignment unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the member `x` is packed, so that it immediately follows `a`, with no padding for alignment:

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

> **Note:** The device architecture requires that words be aligned on even byte boundaries, so care must be taken when using the packed attribute to avoid run-time addressing errors.

persistent

The `persistent` attribute specifies that the variable should not be initialized or cleared at startup. A variable with the persistent attribute could be used to store state information that will remain valid after a device reset.

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

Persistent data is not normally initialized by the C run-time. However, from a cold-restart, persistent data may not have any meaningful value. This code example shows how to safely initialize such data:

```
#include "p24Fxxxx.h"

int last_mode __attribute__((persistent));

int main()
{
    if ((RCONbits.POR == 0) &&
        (RCONbits.BOR == 0)) {
      /* last_mode is valid */
    } else {
      /* initialize persistent data */
      last_mode = 0;
    }
}
```

reverse (*alignment*)

The `reverse` attribute specifies a minimum alignment for the ending address of a variable, plus one. The alignment is specified in bytes and must be a power of two. Reverse-aligned variables can be used for decrementing modulo buffers in dsPIC DSC assembly language. This attribute could be useful if an application defines variables in C that will be accessed from assembly language.

```
int buf1[128] __attribute__ ((reverse(256)));
```

The `reverse` attribute conflicts with the `aligned` and `section` attributes. An attempt to name a section for a reverse-aligned variable will be ignored with a warning. It is an error condition to specify both `reverse` and `aligned` for the same variable. A variable with the `reverse` attribute cannot be placed into the `auto_psv` space (see the `space()` attribute or the `-mconst-in-code` option); attempts to do so will cause a warning and the compiler will place the variable into the PSV space.

section (*"section-name"*)

By default, the compiler places the objects it generates in sections such as `.data` and `.bss`. The `section` attribute allows you to override this behavior by specifying that a variable (or function) lives in a particular section.

```
struct a { int i[32]; };
struct a buf __attribute__((section("userdata"))) = {{0}};
```

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 also specifies the run-time address of the variable, using the *address* parameter.

```
extern volatile int __attribute__ ((sfr(0x200)))u1mod;
```

The use of the extern specifier is required in order to not produce an error.

> **Note:** By convention, the `sfr` attribute is used only in processor header files. To define a general user variable at a specific address use the `address` attribute in conjunction with `near` or `far` to specify the correct addressing mode.

space (*space*)

Normally, the compiler allocates variables in general data space. The `space` attribute can be used to direct the compiler to allocate a variable in specific memory spaces. Memory spaces are discussed further in **Section 4.5 “Memory 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.

`eds`

Allocate the variable in the extended data space. For devices that do not have extended data space, this is equivalent to `space(data)`. Variables in `space(eds)` will generally require special handling to access. Refer to **Chapter 6. “Additional C Pointer Types”** for more information.
`space(eds)` has been deprecated in favour of the `eds` attribute.

DD **`xmemory`** - dsPIC30F/33F DSCs only

Allocate the variable in X data space. Variables in X data space can be accessed using ordinary C statements. An example of `xmemory` space allocation is:

```
int x[32] __attribute__ ((space(xmemory)));
```

DD **`ymemory`** - dsPIC30F/33F 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, or using the program space visibility window.

auto_psv

Allocate the variable in program space, in a compiler-managed section designated for automatic program space visibility window access. Variables in `auto_psv` space can be read (but not written) using ordinary C statements, and are subject to a maximum of 32K total space allocated. When specifying `space(auto_psv)`, it is not possible to assign a section name using the `section` attribute; any section name will be ignored with a warning. A variable in the `auto_psv` space cannot be placed at a specific address or given a reverse alignment.

> **Note:** Variables placed in the `auto_psv` section are not loaded into data memory at startup. This attribute may be useful for reducing RAM usage.

DD **dma** - PIC24H MCUs, dsPIC33F 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()` (see **Appendix B. "Built-in Functions"**) can be used to find the correct offset for configuring the DMA peripheral.

```
#include <p24Hxxxx.h>
unsigned int BufferA[8] __attribute__((space(dma)));
unsigned int BufferB[8] __attribute__((space(dma)));

int main()
{
  DMA1STA = __builtin_dmaoffset(BufferA);
  DMA1STB = __builtin_dmaoffset(BufferB);
  /* ... */
}
```

psv

Allocate the variable in program space, in a section designated for program space visibility window access. The linker will locate the section so that the entire variable can be accessed using a single setting of the PSVPAG register. Variables in PSV space are not managed by the compiler and can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

DD **eedata** - PIC24F, dsPIC30F/33F DSCs only

Allocate the variable in EEData space. Variables in EEData space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

pmp

Allocate the variable in off chip memory associated with the PMP peripheral. For complete details please see **Section 6.3 "PMP Pointers"**.

`external`

Allocate the variable in a user defined memory space. For complete details please see **Section 6.4 “External Pointers”**.

`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;
```

`unused`

This attribute, attached to a variable, means that the variable is meant to be possibly unused. The compiler will not produce an unused variable warning for this variable.

`weak`

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol may be superseded by a global definition. When `weak` is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((__weak__)) s;
int foo() {
  if (&s) return s;
  return 0; /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that `s` has been defined (and returns its value if it has). Otherwise ‘`0`’ is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

The `weak` attribute may be applied to functions as well as variables:

```
extern int __attribute__((__weak__)) compress_data(void *buf);
int process(void *buf) {
  if (compress_data) {
    if (compress_data(buf) == -1) /* error */
  }
  /* process buf */
}
```

In the above code, the function `compress_data` will be used only if it is linked in from some other module. Deciding whether or not to use the feature becomes a link-time decision, not a compile time decision.

The affect of the weak attribute on a definition is more complicated and requires multiple files to describe:

```
/* weak1.c */
int __attribute__((__weak__)) i;

void foo() {
  i = 1;
}

/* weak2.c */
int i;
extern void foo(void);

void bar() {
  i = 2;
}

main() {
  foo();
  bar();
}
```

Here the definition in weak2.c of i causes the symbol to become a strong definition. No link error is emitted and both i's refer to the same storage location. Storage is allocated for weak1.c's version of i, but this space is not accessible.

There is no check to ensure that both versions of i have the same type; changing i in weak2.c to be of type float will still allow a link, but the behavior of function foo will be unexpected. foo will write a value into the least significant portion of our 32-bit float value. Conversely, changing the type of the weak definition of i in weak1.c to type float may cause disastrous results. We will be writing a 32-bit floating point value into a 16-bit integer allocation, overwriting any variable stored immediately after our i.

In the cases where only weak definitions exist, the linker will choose the storage of the first such definition. The remaining definitions become in-accessible.

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

2.3.2 Specifying Attributes of Functions

In the compiler, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently supported for functions:

- `address (addr)`
- `alias ("target")`
- `auto_psv, no_auto_psv`
- `boot`
- `const`
- `deprecated`
- `far`
- `format (archetype, string-index, first-to-check)`
- `format_arg (string-index)`
- `interrupt [ ( [ save(list) ] [, irq(irqid) ] [, altirq(altirqid)] [, preprologue(asm) ] ) ]`
- `near`
- `no_instrument_function`
- `noload`
- `noreturn`
- `section ("section-name")`
- `secure`
- `shadow`
- `unused`
- `user_init`
- `weak`

You may also specify attributes with __ (double underscore) preceding and following each keyword (e.g., `__shadow__` instead of `shadow`). This allows you to use them in header files without being concerned about a possible macro of the same name.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

`address (addr)`

The `address` attribute specifies an absolute address for the function. This attribute cannot be used in conjunction with a `section` attribute; the `address` attribute will take precedence.

```
void __attribute__ ((address(0x100))) foo() {
...
}
```

Alternatively, you may define the address in the function prototype:

```
void foo() __attribute__ ((address(0x100)));
```

`alias ("target")`

The alias attribute causes the declaration to be emitted as an alias for another symbol, which must be specified.

Use of this attribute results in an external reference to target, which must be resolved during the link phase.

auto_psv, no_auto_psv

The auto_psv attribute, when combined with the interrupt attribute, will cause the compiler to generate additional code in the function prologue to set the PSVPAG 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 PSVPAG has been modified and the interrupt routine, or a function it calls, uses an auto_psv variable. Compare this with no_auto_psv. If neither auto_psv nor no_auto_psv option is specified for an interrupt routine, the compiler will issue a warning and select this option.

The no_auto_psv attribute, when combined with the interrupt attribute, will cause the compiler to not generate additional code for accessing space(auto_psv) (or constants in the constants-in-code memory model) variables. Use this option if none of the conditions under auto_psv hold true. If neither auto_psv nor no_auto_psv option is specified for an interrupt routine, the compiler will issue a warning and assume auto_psv.

boot

This attribute directs the compiler to allocate a function in the boot segment of program Flash.

For example, to declare a protected function:

```
void __attribute__((boot)) func();
```

An optional argument can be used to specify a protected access entry point within the boot segment. The argument may be a literal integer in the range 0 to 31 (except 16), or the word unused. Integer arguments correspond to 32 instruction slots in the segment access area, which occupies the lowest address range of each secure segment. The value 16 is excluded because access entry 16 is reserved for the secure segment interrupt vector. The value unused is used to specify a function for all of the unused slots in the access area.

Access entry points facilitate the creation of application segments from different vendors that are combined at run time. They can be specified for external functions as well as locally defined functions. For example:

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
  boot_service3();
}
```

Note: In order to allocate functions with the boot or secure attribute, memory for the boot and/or secure segment must be reserved. This can be accomplished by setting configuration words in source code, or by specifying linker command options. For more information, see Chapter 8.8, "Options that Specify CodeGuard Security Features", in the linker manual (DS51317).

If attributes boot or secure are used, and memory is not reserved, then a link error will result.

To specify a secure interrupt handler, use the boot attribute in combination with the interrupt attribute:

```
void __attribute__((boot,interrupt)) boot_interrupts();
```

When an access entry point is specified for an external secure function, that function need not be included in the project for a successful link. All references to that function will be resolved to a fixed location in Flash, depending on the security model selected at link time.

When an access entry point is specified for a locally defined function, the linker will insert a branch instruction into the secure segment access area. The exception is for access entry 16, which is represented as a vector (i.e, an instruction address) rather than an instruction. The actual function definition will be located beyond the access area; therefore the access area will contain a jump table through which control can be transferred from another security segment to functions with defined entry points.

Automatic variables are owned by the enclosing function and do not need the `boot` attribute. They may be assigned initial values, as shown:

```
void __attribute__((boot)) chuck_cookies()
{
  int hurl;
  int them = 55;
  char *where = "far";
  splat(where);
  /* ... */
}
```

Note that the initial value of `where` is based on a string literal which is allocated in the PSV constant section `.boot_const`. The compiler will set PSVPAG to the correct value upon entrance to the function. If necessary, the compiler will also restore PSVPAG after the call to `splat()`.

`const`

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `const`. For example:

```
int square (int) __attribute__ ((const int));
```

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

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to have a `void` return type.

`deprecated`

See **Section 2.3.1 “Specifying Attributes of Variables”** for information on the `deprecated` attribute.

`far`

The `far` attribute tells the compiler that the function should not be called using a more efficient form of the call instruction.

`format (`*`archetype, string-index, first-to-check`*`)`

The `format` attribute specifies that a function takes `printf`, `scanf` or `strftime` style arguments which should be type-checked against a format string. For example, consider the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
     __attribute__ ((format (printf, 2, 3)));
```

This causes the compiler to check the arguments in calls to my_printf for consistency with the printf style format string argument my_format.

The parameter *archetype* determines how the format string is interpreted, and should be one of printf, scanf or strftime. The parameter *string-index* specifies which argument is the format string argument (arguments are numbered from the left, starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as vprintf), specify the third parameter as zero. In this case, the compiler only checks the format string for consistency.

In the example above, the format string (my_format) is the second argument of the function my_print, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The format attribute allows you to identify your own functions that take format strings as arguments, so that the compiler can check the calls to these functions for errors. The compiler always checks formats for the ANSI library functions printf, fprintf, sprintf, scanf, fscanf, sscanf, strftime, vprintf, vfprintf and vsprintf, whenever such warnings are requested (using -Wformat), so there is no need to modify the header file stdio.h.

format_arg (*string-index*)

The format_arg attribute specifies that a function takes printf or scanf style arguments, modifies it (for example, to translate it into another language), and passes it to a printf or scanf style function. For example, consider the declaration:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

This causes the compiler to check the arguments in calls to my_dgettext, whose result is passed to a printf, scanf or strftime type function for consistency with the printf style format string argument my_format.

The parameter *string-index* specifies which argument is the format string argument (starting from 1).

The format-arg attribute allows you to identify your own functions which modify format strings, so that the compiler can check the calls to printf, scanf or strftime function, whose operands are a call to one of your own functions.

interrupt [([save(*list*)] [, irq(*irqid*)]
[, altirq(*altirqid*)] [, preprologue(*asm*)])]

Use this option to indicate that the specified function is an interrupt handler. The compiler will generate function prologue and epilogue sequences suitable for use in an interrupt handler when this attribute is present. The optional parameter save specifies a list of variables to be saved and restored in the function prologue and epilogue, respectively. The optional parameters irq and altirq specify interrupt vector table ID's to be used. The optional parameter preprologue specifies assembly code that is to be emitted before the compiler-generated prologue code. See **Chapter 8. "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.

near

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

no_instrument_function

If the command line option `-finstrument-function` is given, profiling function calls will be generated at entry and exit of most user-compiled functions. Functions with this attribute will not be so instrumented.

noload

The `noload` attribute indicates that space should be allocated for the function, but that the actual code should not be loaded into memory. This attribute could be useful if an application is designed to load a function into memory at run time, such as from a serial EEPROM.

```
void bar() __attribute__ ((noload)) {
...
}
```

noreturn

A few standard library functions, such as `abort` and `exit`, cannot return. The compiler knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example:

```
void fatal (int i) __attribute__ ((noreturn));

void
fatal (int i)
{
  /* Print error message. */
  exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. Also, it helps avoid spurious warnings of uninitialized variables.

It does not make sense for a `noreturn` function to have a return type other than `void`.

section (*"section-name"*)

Normally, the compiler places the code it generates in the `.text` section. Sometimes, however, you need additional sections, or you need certain functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, consider the declaration:

```
extern void foobar (void) __attribute__ ((section (".libtext")));
```

This puts the function `foobar` in the `.libtext` section.

The `section` attribute conflicts with the `address` attribute. The section name will be ignored with a warning.

secure

This attribute directs the compiler to allocate a function in the `secure` segment of program Flash.

For example, to declare a protected function:

```
void __attribute__((secure)) func();
```

An optional argument can be used to specify a protected access entry point within the `secure` segment. The argument may be a literal integer in the range 0 to 31 (except 16), or the word `unused`. Integer arguments correspond to 32 instruction slots in the segment access area, which occupies the lowest address range of each secure segment. The value 16 is excluded because access entry 16 is reserved for the secure segment interrupt vector. The value `unused` is used to specify a function for all of the unused slots in the access area.

Access entry points facilitate the creation of application segments from different vendors that are combined at run time. They can be specified for external functions as well as locally defined functions. For example:

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
  boot_service3();
}
```

> **Note:** In order to allocate functions with the `boot` or `secure` attribute, memory for the boot and/or secure segment must be reserved. This can be accomplished by setting configuration words in source code, or by specifying linker command options. For more information, see Chapter 8.8, "Options that Specify CodeGuard Security Features", in the linker manual (DS51317).
>
> If attributes `boot` or `secure` are used, and memory is not reserved, then a link error will result.

To specify a secure interrupt handler, use the `secure` attribute in combination with the interrupt attribute:

```
void __attribute__((secure,interrupt)) secure_interrupts();
```

When an access entry point is specified for an external secure function, that function need not be included in the project for a successful link. All references to that function will be resolved to a fixed location in Flash, depending on the security model selected at link time.

When an access entry point is specified for a locally defined function, the linker will insert a branch instruction into the secure segment access area. The exception is for access entry 16, which is represented as a vector (i.e, an instruction address) rather than an instruction. The actual function definition will be located beyond the access area; therefore the access area will contain a jump table through which control can be transferred from another security segment to functions with defined entry points.

Automatic variables are owned by the enclosing function and do not need the `secure` attribute. They may be assigned initial values, as shown:

```
void __attribute__((secure)) chuck_cookies()
{
  int hurl;
  int them = 55;
  char *where = "far";
  splat(where);
  /* ... */
}
```

Note that the initial value of `where` is based on a string literal which is allocated in the PSV constant section `.secure_const`. The compiler will set PSVPAG to the correct value upon entrance to the function. If necessary, the compiler will also restore PSVPAG after the call to `splat()`.

shadow

The shadow attribute causes the compiler to use the shadow registers rather than the software stack for saving registers. This attribute is usually used in conjunction with the interrupt attribute.

```
void __attribute__ ((interrupt, shadow)) _T1Interrupt (void);
```

unused

This attribute, attached to a function, means that the function is meant to be possibly unused. The compiler will not produce an unused function warning for this function.

user_init

The user_init attribute may be applied to any non-interrupt function with void parameter and return types. Applying this attribute will cause default C start-up modules to call this function before the user main is executed. There is no guarantee of ordering, so these functions cannot rely on other user_init functions having been previously run; these functions will be called after PSV and data initialization. A user_init may still be called by the executing program. For example:

```
void __attribute__((user_init)) initialize_me(void) {
   // perform initalization sequence alpha alpha beta
}
```

weak

See **Section 2.3.1 “Specifying Attributes of Variables”** for information on the weak attribute.

2.3.3 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 (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 nonintegrated 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.

2.3.4 Variables in Specified 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.

Accumulator registers are not allocated by the compiler so it is safe to allocate them using the following syntax:

```
register int Accum_A asm("A");
```

No other register should be allocated.

2.3.5 Complex Numbers

The compiler supports complex data types. You can declare both complex integer types and complex floating types, using the keyword `__complex__`.

For example, `__complex__ float x`; declares `x` as a variable whose real part and imaginary part are both of type float. `__complex__ short int y;` declares `y` to have real and imaginary parts of type `short int`.

To write a constant with a complex data type, use the suffix '`i`' or '`j`' (either one; they are equivalent). For example, `2.5fi` has type `__complex__ float` and `3i` has type `__complex__ int`. Such a constant is a purely imaginary value, but you can form any complex value you like by adding one to a real constant.

To extract the real part of a complex-valued expression *exp*, write `__real__` *exp*. Similarly, use `__imag__` to extract the imaginary part. For example;

```
__complex__ float z;
float r;
float i;

r = __real__ z;
i = __imag__ z;
```

The operator '~' performs complex conjugation when used on a value with a complex type.

The compiler can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice-versa). The debugging information format has no way to represent noncontiguous allocations like these, so the compiler describes noncontiguous complex variables as two separate variables of noncomplex type. If the variable's actual name is `foo`, the two fictitious variables are named `foo$real` and `foo$imag`.

2.3.6 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. Addition, subtraction and bitwise boolean operations on these types are open-coded, but division and shifts are not open-coded. The operations that are not open-coded use special library routines that come with the compiler.

2.3.7 Referring to a Type with `typeof`

Another way to refer to the type of an expression is with the `typeof` keyword. The syntax for using this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is a pointer to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`.

A `typeof` construct can be used anywhere a `typedef` name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to:
  ```
  typeof (*x) y;
  ```
- This declares `y` as an array of such values:
  ```
  typeof (*x) y[4];
  ```
- This declares `y` as an array of pointers to characters:
  ```
  typeof (typeof (char *)[4]) y;
  ```
  It is equivalent to the following traditional C declaration:
  ```
  char *y[4];
  ```

To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T) typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of four pointers to `char`.

2.4 STATEMENT DIFFERENCES

This section describes the statement differences between plain ANSI C and the C accepted by the compiler. The statement differences are part of the base GCC implementation, and the discussion in the section is based on the standard GCC documentation, tailored for the specific syntax and semantics of the 16-bit compiler port of GCC.

- Labels as Values
- Conditionals with Omitted Operands
- Case Ranges

2.4.1 Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '&&'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement, `goto *exp;`. For example:

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

> **Note:** This does not check whether the subscript is in bounds. (Array indexing in C never does.)

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner and therefore preferable to an array.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for fast dispatching.

This mechanism can be misused to jump to code in a different function. The compiler cannot prevent this from happening, so care must be taken to ensure that target addresses are valid for the current function.

2.4.2 Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression:

```
x ? : y
```

has the value of x if that is nonzero; otherwise, the value of y.

This example is perfectly equivalent to:

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

2.4.3 Case Ranges

You can specify a range of consecutive values in a single case label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual case labels, one for each integer value from *low* to *high*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the ..., otherwise it may be parsed incorrectly when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

2.5 EXPRESSION DIFFERENCES

This section describes the expression differences between plain ANSI C and the C accepted by the compiler.

2.5.1 Binary Literals

A sequence of binary digits preceded by 0b or 0B (the numeral '0' followed by the letter 'b' or 'B') is taken to be a binary integer. The binary digits consist of the numerals '0' and '1'. For example, the (decimal) number 255 can be written as 0b11111111.

Like other integer literals, a binary literal may be suffixed by the letter 'u' or 'U', to specify that it is unsigned. A binary literal may also be suffixed by the letter 'l' or 'L', to specify that it is long. Similarly, the suffix 'll' or 'LL' denotes a long, long binary literal.

Chapter 3. Using the Compiler on the Command Line

3.1 INTRODUCTION

This chapter discusses using the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB C30) on the command line. For information on using the compiler with MPLAB IDE, please refer to the "*16-bit Language Tools Getting Started*" (DS70094).

3.2 HIGHLIGHTS

Items discussed in this chapter are:

- Overview
- File Naming Conventions
- Options
- Environment Variables
- Predefined Macro Names
- Compiling a Single File on the Command Line
- Compiling Multiple Files on the Command Line
- Notable Symbols

3.3 OVERVIEW

The compilation driver program (`pic30-gcc`) 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.

The basic form of the compiler command line is:

```
pic30-gcc [options] files
```

> **Note:** This executable name applies for all 16-bit compilers, i.e., MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs, MPLAB C Compiler for dsPIC DSCs, and MPLAB C Compiler for PIC24 MCUs.

The available options are described in **Section 3.5 "Options"**.

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

For example, to compile, assemble and link the C source file `hello.c`, creating the absolute executable `hello.exe`.

```
pic30-gcc -o hello.exe hello.c
```

3.4 FILE NAMING CONVENTIONS

The compilation driver recognizes the following file extensions, which are case-sensitive.

TABLE 3-1: 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).
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.

3.5 OPTIONS

The compiler has many options for controlling compilation, all of which are case-sensitive.

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

3.5.1 Options Specific to 16-Bit Devices

For more information on the memory models, see **Section 4.6 "Memory Models"**.

TABLE 3-2: dsPIC® DSC DEVICE-SPECIFIC OPTIONS

Option	Definition
`-mconst-in-code`	Put constants in the auto_psv space. The compiler will access these constants using the PSV window. **(This is the default.)**
`-mconst-in-data`	Put constants in the data memory space.
`-mconst-in-auxflash`	When combined with `-mconst-in-code`, put call `const` qualified file scope variables into auxiliary FLASH. All modules with auxiliary FLASH should be compiled with this option; otherwise a link error may occur.
`-merrata=` *`id`*`[,`*`id`*`]*`	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. An *`id`* of `list` will display the currently supported errata identifiers along with a brief description of the errata. An *`id`* of `all` will enable all currently supported errata work arounds.
`-mfillupper`	Specify the upper byte of variables stored into `space(prog)` sections. The `fillupper` attribute will perform the same function on individual variables.
`-mlarge-arrays`	Specifies that arrays may be larger than the default maximum size of 32K. See **Section 4.15 "Using Large Arrays"** for more information.
`-mlarge-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.
`-mlarge-data`	Compile using the large data model. No assumptions are made about the location of static and external variables.
`-mcpu=` *`target`*	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 3.7 "Predefined Macro Names"** for more information. A full list of accepted targets can be seen in the `Readme.htm` file that came with the release.
STD `-mpa`[(1)]	Enable the procedure abstraction optimization. There is no limit on the nesting level.

Note 1: The procedure abstractor behaves as the inverse of inlining functions. The pass is designed to extract common code sequences from multiple sites throughout a translation unit and place them into a common area of code. Although this option generally does not improve the run-time performance of the generated code, it can reduce the code size significantly. Programs compiled with `-mpa` can be harder to debug; it is not recommended that this option be used while debugging using the COFF object format.
The procedure abstractor is invoked as a separate phase of compilation, after the production of an assembly file. This phase does not optimize across translation units. When the procedure-optimizing phase is enabled, inline assembly code must be limited to valid machine instructions. Invalid machine instructions or instruction sequences, or assembler directives (sectioning directives, macros, include files, etc.) must not be used, or the procedure abstraction phase will fail, inhibiting the creation of an output file.

TABLE 3-2: dsPIC® DSC DEVICE-SPECIFIC OPTIONS (CONTINUED)

Option	Definition
STD `-mpa=n`(1)	Enable the procedure abstraction optimization up to level *n*. If *n* is zero, the optimization is disabled. If *n* is 1, first level of abstraction is allowed; that is, instruction sequences in the source code may be abstracted into a subroutine. If *n* 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 *n*. The net effect is to limit the subroutine call nesting depth to a maximum of *n*.
STD `-mno-pa`(1)	Do not enable the procedure abstraction optimization. **(This is the default.)**
`-mno-isr-warn`	By default the compiler will produce a warning if the `__interrupt__` 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 *omf* specifier can be one of the following: `coff` Produce COFF object files. **(This is the default.)** `elf` Produce ELF object files. The debugging format used for ELF object files is DWARF 2.0.
`-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.)**
`-msmall-data`	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.)**
`-msmall-scalar`	Like `-msmall-data`, except that only static and external scalars are assumed to be in the lower 8 KB of data memory space. **(This is the default.)**
`-mtext=`*name*	Specifying `-mtext=`*name* will cause text (program code) to be placed in a section named *name* rather than the default `.text` section. No white spaces should appear around the `=`.
`-mauxflash`	Place all code from the current translation unit into auxiliary FLASH. This option is only available on devices that have auxiliary FLASH.
`-msmart-io [=0|1|2]`	This option attempts to statically analyze format strings passed to `printf`, `scanf` 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. `-msmart-io=0` disables this option, while `-msmart-io=2` causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. **`-msmart-io=1` is the default** and will only convert the literal values it can prove.

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

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

3.5.2 Options for Controlling the Kind of Output

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

TABLE 3-3: KIND-OF-OUTPUT CONTROL OPTIONS

Option	Definition
`-c`	Compile or assemble the source files, but do not link. The default file extension is `.o`.
`-E`	Stop after the preprocessing stage, i.e., before running the compiler proper. The default output file is `stdout`.
`-o` *`file`*	Place the output in *`file`*.
`-S`	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is `.s`.
`-v`	Print the commands executed during each stage of compilation.

`-x`	You can specify the input language explicitly with the `-x` option:
	-x language
	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 `-x` option. The following values are supported by the compiler:
	`c c-header cpp-output`
	`assembler assembler-with-cpp`
	
	-x none
	Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another `-x` option has been used.
	For example:
	`pic30-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s`
	
	Without the `-x none`, the compiler will assume all the input files are for the assembler.

`--help`	Print a description of the command line options.

3.5.3 Options for Controlling the C Dialect

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

TABLE 3-4: C DIALECT CONTROL OPTIONS

Option	Definition
`-ansi`	Support all (and only) ANSI-standard C programs.
`-aux-info filename`	Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C. Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (`I`, `N` for new or `O` for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (`C` or `F`, 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.
`-ffreestanding`	Assert that compilation takes place in a freestanding environment. This implies `-fno-builtin`. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to `-fno-hosted`.
`-fno-asm`	Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. `-ansi` implies `-fno-asm`.
`-fno-builtin` `-fno-builtin-`*`function`*	Don't recognize built-in functions that do not begin with `__builtin_` as prefix.
`-fsigned-char`	Let the type `char` be signed, like `signed char`. **(This is the default.)**

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`
	These options control whether a bit field is signed or unsigned, when the declaration does not use either signed or unsigned. By default, such a bit field is signed, unless `-traditional` is used, in which case bit fields are always unsigned.

`-funsigned-char`	Let the type `char` be unsigned, like `unsigned char`.

3.5.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

You can request many specific warnings with options beginning `-W`, for example, `-Wimplicit`, to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning `-Wno-` to turn off warnings, for example, `-Wno-implicit`. This manual lists only one of the two forms, whichever is not the default.

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

TABLE 3-5: WARNING/ERROR OPTIONS IMPLIED BY `-WALL`

Option	Definition
`-fsyntax-only`	Check the code for syntax, but don't do anything beyond that.
`-pedantic`	Issue all the warnings demanded by strict ANSI C; reject all programs that use forbidden extensions.
`-pedantic-errors`	Like `-pedantic`, except that errors are produced rather than warnings.
`-w`	Inhibit all warning messages.
`-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 `char`.
`-Wcomment` `-Wcomments`	Warn whenever a comment-start sequence `/*` appears in a `/*` comment, or whenever a Backslash-Newline appears in a `//` comment.
`-Wdiv-by-zero`	Warn about compile-time integer division by zero. To inhibit the warning messages, use `-Wno-div-by-zero`. 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 `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.
`-Wimplicit`	Equivalent to specifying both `-Wimplicit-int` and `-Wimplicit-function-declaration`.
`-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 `main` is suspicious. `main` should be a function with external linkage, returning `int`, 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 a is not fully bracketed, but that for b is fully bracketed.
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

TABLE 3-5: WARNING/ERROR OPTIONS IMPLIED BY -WALL (CONTINUED)

Option	Definition

-Wmultichar
-Wno-multichar
	Warn if a multi-character *character* 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 *character* constant:
	`char`
	`xx(void)`
	`{`
	`return('xx');`
	`}`

-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 `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.
-Wsequence-point	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap. It is not specified, when, between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that "Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored." If a program breaks these rules, the results on any particular implementation are entirely unpredictable. Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. 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.

TABLE 3-5: WARNING/ERROR OPTIONS IMPLIED BY -WALL (CONTINUED)

Option	Definition
-Wswitch	Warn whenever a switch statement has an index of enumeral 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.) case labels outside the enumeration range also provoke warnings when this option is used.
-Wsystem-headers	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 -Wall in conjunction with this option will not warn about unknown pragmas in system headers; for that, -Wunknown-pragmas must also be used.
-Wtrigraphs	Warn if any trigraphs are encountered (assuming they are enabled).
-Wuninitialized	Warn if an automatic variable is used without first being initialized. These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing. These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared volatile, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.
-Wunknown-pragmas	Warn when a #pragma 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 -Wall 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 -W and -Wunused must be specified. Casting an expression to void suppresses this warning for an expression. Similarly, the unused 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 unused attribute (see **Section 2.3.1 “Specifying Attributes of Variables”**).

TABLE 3-5: WARNING/ERROR OPTIONS IMPLIED BY -WALL (CONTINUED)

Option	Definition
-Wunused-parameter	Warn whenever a function parameter is unused aside from its declaration. To suppress this warning, use the unused attribute (see **Section 2.3.1 "Specifying Attributes of Variables"**).
-Wunused-variable	Warn whenever a local variable or non-constant static variable is unused aside from its declaration. To suppress this warning, use the unused attribute (see **Section 2.3.1 "Specifying Attributes of Variables"**).
-Wunused-value	Warn whenever a statement computes a result that is explicitly not used. To suppress this warning, cast the expression to void.

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 3-6: WARNING/ERROR OPTIONS NOT IMPLIED BY -WALL

Option	Definition

-W	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 };`

-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 int foof() is cast to anything *.
-Wcast-align	Warn whenever a pointer is cast, such that the required alignment of the target is increased. For example, warn if a char * is cast to an int *.
-Wcast-qual	Warn whenever a pointer is cast, so as to remove a type qualifier from the target type. For example, warn if a const char * is cast to an ordinary char *.

TABLE 3-6: WARNING/ERROR OPTIONS NOT IMPLIED BY -WALL

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 x = -1 if x is unsigned. But do not warn about explicit casts like (unsigned) -1.
-Werror	Make all warnings into errors.
-Winline	Warn if a function can not be inlined, and either it was declared as inline, or else the -finline-functions option was given.
-Wlarger-than-*len*	Warn whenever an object of larger than *len* bytes is defined.
-Wlong-long -Wno-long-long	Warn if long long type is used. This is default. To inhibit the warning messages, use -Wno-long-long. Flags -Wlong-long and -Wno-long-long are taken into account only when -pedantic 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 -Wformat 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 -Wformat is enabled.
-Wmissing-noreturn	Warn about functions that might be candidates for attribute noreturn. These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. Actually, do not ever return before adding the noreturn 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 extern 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 deprecated 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 void. The compiler assigns these types a size of 1, for convenience in calculations with void * 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.

TABLE 3-6: WARNING/ERROR OPTIONS NOT IMPLIED BY -WALL

Option	Definition
-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 -W; to get the other warnings of -W without this warning, use -W -Wno-sign-compare.
-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. • 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 long. • 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 #if directive.
-Wunreachable-code	Warn if the compiler detects that code will never be executed. It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently-unreachable code. For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.
-Wwrite-strings	Give string constants the type const char[length] so that copying the address of one into a non-const char * 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 const in declarations and prototypes. Otherwise, it will just be a nuisance, which is why -Wall does not request these warnings.

3.5.5 Options for Debugging

The following options are used for debugging.

TABLE 3-7: DEBUGGING OPTIONS

Option	Definition
`-g`	Produce debugging information. The compiler supports the use of `-g` with `-O` making it possible to debug optimized code. The shortcuts taken by optimized code may occasionally produce surprising results: • 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.
`-Q`	Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.
`-save-temps`	Don't delete intermediate files. Place them in the current directory and name them based on the source file. Thus, compiling `foo.c` with `-c -save-temps` would produce the following files: `foo.i` (preprocessed file) `foo.p` (pre procedure abstraction assembly language file) `foo.s` (assembly language file) `foo.o` (object file)

3.5.6 Options for Controlling Optimization

The following options control compiler optimizations.

TABLE 3-8: GENERAL OPTIMIZATION OPTIONS

Option	Definition
`-O0`	Do not optimize. **(This is the default.)** Without `-O`, 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 `register` in registers.
`-O` `-O1`	Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. With `-O`, the compiler tries to reduce code size and execution time. When `-O` is specified, the compiler turns on `-fthread-jumps` and `-fdefer-pop`. The compiler turns on `-fomit-frame-pointer`.

TABLE 3-8: GENERAL OPTIMIZATION OPTIONS (CONTINUED)

	Option	Definition
STD	`-O2`	Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. `-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 force copy of memory operands (`-fforce-mem`) and Frame Pointer elimination (`-fomit-frame-pointer`). As compared to `-O`, this option increases both compilation time and the performance of the generated code.
STD	`-O3`	Optimize yet more. `-O3` turns on all optimizations specified by `-O2` and also turns on the `inline-functions` option.
STD	`-Os`	Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

The following options control specific optimizations. The `-O2` option turns on all of these optimizations except `-funroll-loops`, `-funroll-all-loops` and `-fstrict-aliasing`.

You can use the following flags in the rare cases when “fine-tuning” of optimizations to be performed is desired.

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS

	Option	Definition
STD	`-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.
STD	`-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.
STD	`-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.

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

	Option	Definition
STD	`-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.
STD	`-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`.
STD	`-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.
STD	`-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.
STD	`-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.
STD	`-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.
STD	`-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".

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

	Option	Definition
STD	`-frerun-cse-after-loop`	Rerun common subexpression elimination after loop optimizations has been performed.
	`-frerun-loop-opt`	Run the loop optimizer twice.
STD	`-fschedule-insns`	Attempt to reorder instructions to eliminate dsPIC® DSC Read-After-Write stalls (see the "*dsPIC30F Family Reference Manual*" (DS70046) for more details). Typically improves performance with no impact on code size.
STD	`-fschedule-insns2`	Similar to `-fschedule-insns`, but requests an additional pass of instruction scheduling after register allocation has been done.
STD	`-fstrength-reduce`	Perform the optimizations of loop strength reduction and elimination of iteration variables.

STD	`-fstrict-aliasing`	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 `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.
		Pay special attention to code like this:
		`union a_union {`
		`  int i;`
		`  double d;`
		`};`
		
		`int f() {`
		`  union a_union t;`
		`  t.d = 3.0;`
		`  return t.i;`
		`}`
		The practice of reading from a different union member than the one most recently written to (called "type-punning") is common. Even with `-fstrict-aliasing`, type-punning is allowed, provided the memory is accessed through the union type. So, the code above will work as expected. However, this code might not:
		`int f() {`
		`  a_union t;`
		`  int* ip;`
		`  t.d = 3.0;`
		`  ip = &t.i;`
		`  return *ip;`
		`}`

	`-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. `-funroll-loops` implies both `-fstrength-reduce` and `-frerun-cse-after-loop`.

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
`-funroll-all-loops`	Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. `-funroll-all-loops` implies `-fstrength-reduce`, as well as `-frerun-cse-after-loop`.

Options of the form `-f`*`flag`* 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 3-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
`-fforce-mem`	Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. The `-O2` option turns on this option.
`-finline-functions`	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 `static`, then the function is normally not output as assembler code in its own right.
STD `-finline-limit=`*n*	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 `inline` keyword). *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of n 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 an another.
`-fkeep-inline-functions`	Even if all calls to a given function are integrated, and the function is declared `static`, output a separate run time callable version of the function. This switch does not affect `extern` inline functions.
`-fkeep-static-consts`	Emit variables declared static const 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 `-fno-keep-static-consts` option.

TABLE 3-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
`-fno-function-cse`	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.
`-fno-inline`	Do not pay attention to the `inline` 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.

3.5.7 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

TABLE 3-11: PREPROCESSOR OPTIONS

Option	Definition

`-A`*`question`* `(`*`answer`*`)`	Assert the answer *`answer`* for question *`question`*, in case it is tested with a preprocessing conditional such as `#if` *`#question`*`(`*`answer`*`)`. `-A-` disables the standard assertions that normally describe the target machine.
	For example, the function prototype for main might be declared as follows:
	`#if #environ(freestanding)`
	`int main(void);`
	`#else`
	`int main(int argc, char *argv[]);`
	`#endif`
	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 *`predicate`* and answer *`answer`*.
`-A` *`predicate`* `=`*`answer`*	Make an assertion with the predicate *`predicate`* and answer *`answer`*. This form is preferred to the older form `-A` *`predicate`*`(`*`answer`*`)`, 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.
`-D`*`macro`*	Define macro *`macro`* with the string 1 as its definition.
`-D`*`macro=defn`*	Define macro *`macro`* as *`defn`*. 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.

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-dN	Like -dD except that the macro arguments and contents are omitted. Only #define 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.
-H	Print the name of each header file used, in addition to other normal activities.
-I-	Any directories you specify with -I options before the -I- options are searched only for the case of #include "*file*"; they are not searched for #include <*file*>. If additional directories are specified with -I options after the -I-, these directories are searched for all #include directives. (Ordinarily all -I directories are used this way.) In addition, the -I- 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 -I-. 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. -I- does not inhibit the use of the standard system directories for header files. Thus, -I- and -nostdinc are independent.
-I*dir*	Add the directory *dir* 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 *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that -I adds to).
-imacros *file*	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. 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.
-include *file*	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.
-iprefix *prefix*	Specify *prefix* 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.

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-iwithprefix` *dir*	Add a directory to the second include path. The directory's name is made by concatenating prefix and *dir*, 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 *dir*, as in the case of `-iwithprefix`.
`-M`	Tell the preprocessor to output a rule suitable for `make` 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 `#include` header files it uses. This rule may be a single line or may be continued with `\`-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program. `-M` implies `-E` (see **Section 3.5.2 "Options for Controlling the Kind of Output"**).
`-MD`	Like `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` 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 `#include` "*file*". System header files included with `#include <`*file*`>` 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 `make` gives if you remove header files without updating the make-file to match.
	This is typical output:
	`test.o: test.c test.h`
	`test.h:`

`-MQ`	Same as `-MT`, but it quotes any characters which are special to `make`. `-MQ '$(objpfx)foo.o'` gives `$$(objpfx)foo.o: foo.c` The default target is automatically quoted, as if it were given with `-MQ`.

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-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 `.c`, 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 '$(objpfx)foo.o'` might give `$(objpfx)foo.o: foo.c`
`-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 **Section 3.5.10 "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.
`-P`	Tell the preprocessor not to generate `#line` directives. Used with the `-E` option (see **Section 3.5.2 "Options for Controlling the Kind of Output"**).
`-trigraphs`	Support ANSI C trigraphs. The `-ansi` option also has this effect.
`-U`*macro*	Undefine macro *macro*. `-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).

3.5.8 Options for Assembling

The following options control assembler operations.

TABLE 3-12: ASSEMBLY OPTIONS

Option	Definition
`-Wa,`*option*	Pass *option* as an option to the assembler. If *option* contains commas, it is split into multiple options at the commas.

3.5.9 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.

TABLE 3-13: LINKING OPTIONS

Option	Definition
`--gc-sections`	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 `-ffunction-sections` option to the compiler command line.
`-L`*dir*	Add directory *dir* to the list of directories to be searched for libraries specified by the command-line option `-l`.
`-legacy-libc`	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.

TABLE 3-13: LINKING OPTIONS (CONTINUED)

Option	Definition

`-l`*`library`*	Search the library named *`library`* when linking.
	The linker searches a standard list of directories for the library, which is actually a file named `lib`*`library`*`.a`. 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, `foo.o -lz bar.o` searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `libz.a`, those functions may not be loaded.
	The directories searched include several standard system directories, plus any that you specify with `-L`.
	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 `-l` option (e.g., `-lmylib`) and specifying a file name (e.g., `libmylib.a`) is that `-l` searches several directories, as specified.
	By default the linker is directed to search:
	`<install-path>\lib`
	for libraries specified with the `-l` option. For a compiler installed into the default location, this would be:
	`c:\Program Files\Microchip\MPLAB C30\lib`
	This behavior can be overridden using the environment variables defined in **Section 3.6 “Environment Variables”**.

`-nodefaultlibs`	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 `memcmp`, `memset` and `memcpy`. 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.
`-nostdlib`	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 `memcmp`, `memset` and `memcpy`. 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.
`-s`	Remove all symbol table and relocation information from the executable.
`-u` *`symbol`*	Pretend *`symbol`* is undefined to force linking of library modules to define the symbol. It is legitimate to use `-u` multiple times with different symbols to force loading of additional library modules.
`-Wl,`*`option`*	Pass *`option`* as an option to the linker. If *`option`* contains commas, it is split into multiple options at the commas.
`-Xlinker` *`option`*	Pass *`option`* 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.

3.5.10 Options for Directory Search

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

TABLE 3-14: DIRECTORY SEARCH OPTIONS

Option	Definition
`-B`*`prefix`*	This option specifies where to find the executables, libraries, include files and data files of the compiler itself. The compiler driver program runs one or more of the sub-programs `pic30-cpp`, `pic30-cc1`, `pic30-as` and `pic30-ld`. It tries *`prefix`* as a prefix for each program it tries to run. For each sub-program to be run, the compiler driver first tries the `-B` prefix, if any. If the sub-program is not found, or if `-B` was not specified, the driver uses the value held in the `PIC30_EXEC_PREFIX` environment variable, if set. See **Section 3.6 "Environment Variables"**, for more information. Lastly, the driver will search the current `PATH` environment variable for the subprogram. `-B` prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `-L` options for the linker. They also apply to include files in the preprocessor, because the compiler translates these options into `-isystem` options for the preprocessor. In this case, the compiler appends `include` to the prefix. Another way to specify a prefix much like the `-B` prefix is to use the environment variable `PIC30_EXEC_PREFIX`.
`-specs=`*`file`*	Process file after the compiler reads in the standard `specs` file, in order to override the defaults that the `pic30-gcc` driver program uses when determining what switches to pass to `pic30-cc1`, `pic30-as`, `pic30-ld`, etc. More than one `-specs=`*`file`* can be specified on the command line, and they are processed in order, from left to right.

3.5.11 Options for Code Generation Conventions

Options of the form `-f`*`flag`* 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 3-15: CODE GENERATION CONVENTION OPTIONS

Option	Definition

`-fargument-alias`
`-fargument-noalias`
`-fargument-noalias-global`
	Specify the possible relationships among parameters and between parameters and global data.
	`-fargument-alias` specifies that arguments (parameters) may alias each other and may alias global storage.
	`-fargument-noalias` specifies that arguments do not alias each other, but may alias global storage.
	`-fargument-noalias-global` 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.

TABLE 3-15: CODE GENERATION CONVENTION OPTIONS (CONTINUED)

Option	Definition
`-fcall-saved-`*`reg`*	Treat the register named *`reg`* 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 *`reg`* 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.
`-fcall-used-`*`reg`*	Treat the register named *`reg`* 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 *`reg`*. 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 *`reg`* 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). *`reg`* 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 `struct` and `union` 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 `double` type equivalent to `float`. This option makes `double` equivalent to `long double`. 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 `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` 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.
`-fvolatile`	Consider all memory references through pointers to be volatile.
`-fvolatile-global`	Consider all memory references to external and global data items to be volatile. The use of this switch has no effect on static data.
`-fvolatile-static`	Consider all memory references to static data to be volatile.

3.6 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 3-16: COMPILER-RELATED ENVIRONMENTAL VARIABLES

Option	Definition
PIC30_C_INCLUDE_PATH	This variable's value is a semicolon-separated list of directories, much like PATH. When the compiler searches for header files, it tries the directories listed in the variable, after the directories specified with -I but before the standard header file directories. If the environment variable is undefined, the preprocessor chooses an appropriate value based on the standard installation. By default, the following directories are searched for include files: <install-path>\include and <install-path>\support\h
PIC30_COMPILER_PATH	The value of PIC30_COMPILER_PATH is a semicolon-separated list of directories, much like PATH. The compiler tries the directories thus specified when searching for subprograms, if it can't find the subprograms using PIC30_EXEC_PREFIX.
PIC30_EXEC_PREFIX	If PIC30_EXEC_PREFIX 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 PATH environment variable. If the PIC30_EXEC_PREFIX environment variable is unset or set to an empty value, the compiler driver chooses an appropriate value based on the standard installation. If the installation has not been modified, this will result in the driver being able to locate the required subprograms. Other prefixes specified with the -B command line option take precedence over the user- or driver-defined value of PIC30_EXEC_PREFIX. Under normal circumstances it is best to leave this value undefined and let the driver locate subprograms itself.
PIC30_LIBRARY_PATH	This variable's value is a semicolon-separated list of directories, much like PATH. This variable specifies a list of directories to be passed to the linker. The driver's default evaluation of this variable is: <install-path>\lib; <install-path>\support\gld.
PIC30_OMF	Specifies the OMF (Object Module Format) to be used by the compiler. By default, the tools create COFF object files. If the environment variable PIC30_OMF has the value elf, the tools will create ELF object files.
TMPDIR	If TMPDIR 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.

3.7 PREDEFINED MACRO NAMES

The compiler predefines several macros which can be tested by conditional directives in source code.

The following preprocessing symbols are defined by the compiler being used.

Compiler	Symbol	Defined with -ansi command-line option?
16-Bit Compiler	`C30`	No
	`__C30`	Yes
	`__C30__`	Yes
ELF-specific	`C30ELF`	No
	`__C30ELF`	Yes
	`__C30ELF__`	Yes
COFF-specific	`C30COFF`	No
	`__C30COFF`	Yes
	`__C30COFF__`	Yes

The following symbols define the target family.

Symbol	Defined with -ansi command-line option?
`__dsPIC30F__`	dsPIC30F target device family
`__dsPIC33E__`	dsPIC33E target device family
`__dsPIC33F__`	dsPIC33F target device family
`__PIC24E__`	PIC24E target device family
`__PIC24F__`	PIC24FJ target device family
`__PIC24FK__`	PIC24FK target device family
`__PIC24H__`	PIC24H target device family

The following symbols define device features.

Symbol	Defined with -ansi command-line option?
`__HAS_DSP__`	Device has a DSP engine
`__HAS_EEDATA__`	Device has EEDATA memory
`__HAS_DMA__`	Device has DMA memory
`__HAS_CODEGUARD__`	Device has CodeGuard™ Security
`__HAS_PMP__`	Device has Parallel Master Port
`__HAS_PMP_ENHANCED__`	Device has Enhanced Parallel Master Port
`__HAS_EDS__`	Device has Extended Data Space

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

The compiler will define the constant `__C30_VERSION__`, giving a numeric value to the version identifier. This can be used to take advantage of new compiler features while still remaining backward compatible with older versions.

The value is based upon the major and minor version numbers of the current release. For example, release version 2.00 will have a `__C30_VERSION__` definition of 200. This macro can be used, in conjunction with standard preprocessor comparison statements, to conditionally include/exclude various code constructs.

The current definition of `__C30_VERSION__` can be discovered by adding `--version` to the command line, or by inspecting the `README.html` file that came with the release.

Constants that have been deprecated may be found in **Appendix E. “Deprecated Features”**.

3.8 COMPILING A SINGLE FILE ON THE COMMAND LINE

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

- `c:\Program Files\Microchip\MPLAB C30\include` - Include directory for ANSI C header file. This directory is where the compiler stores the standard C library system header files. The PIC30_C_INCLUDE_PATH environment variable can point to that directory. (From the DOS command prompt, type set to check this.)
- `c:\Program Files\Microchip\MPLAB C30\support\dsPIC30F\h` - Include directory for dsPIC® DSC device-specific header files. This directory is where the compiler stores the dsPIC DSC device-specific header files.
- `c:\Program Files\Microchip\MPLAB C30\lib` - Library directory: this directory is where the libraries and precompiled object files reside.
- `c:\Program Files\Microchip\MPLAB C30\support\dsPIC30F\gld` - Linker script directory: this directory is where device-specific linker script files may be found.
- `c:\Program Files\Microchip\MPLAB C30\bin` - Executables directory: this directory is where the compiler programs are located. Your `PATH` environment variable should include this directory.

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 <p30f2010.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 `p30f2010.h,` which provides definitions for all special function registers on that part. For more information on header files, see **Chapter 7. "Device Support Files"**.

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

```
C:\> pic30-gcc -mcpu=30f2010 -o ex1.o ex1.c
```

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

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

```
C:\> pic30-bin2hex ex1.o
```

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

3.9 COMPILING MULTIPLE FILES ON THE COMMAND LINE

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

File 1

```
/* ex1.c */
#include <p30f2010.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 <p30f2010.h>
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

Compile both files by typing the following at a DOS prompt:

```
C:\> pic30-gcc -mcpu=30f2010 -o ex1.o 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.o is created.

3.10 NOTABLE SYMBOLS

The 16-bit linker defines several symbols that may be used in your C code development. Please see the *MPLAB Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide* (DS51317) 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:

```
__builtin_tblpage(&_PROGRAM_END)
__builtin_tbloffset(&_PROGRAM_END)

_prog_addressT big_addr;
_init_prog_address(big_addr, _PROGRAM_END)
```

NOTES:

Chapter 4. Run Time Environment

4.1 INTRODUCTION

This section discusses the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30) run-time environment.

4.2 HIGHLIGHTS

Items discussed in this chapter are:

- Address Spaces
- Startup and Initialization
- Memory Spaces
- Memory Models
- Locating Code and Data
- Software Stack
- The C Stack Usage
- The C Heap Usage
- Function Call Conventions
- Register Conventions
- Bit Reversed and Modulo Addressing
- Program Space Visibility (PSV) Usage

4.3 ADDRESS SPACES

The dsPIC Digital Signal Controller (DSC) devices are a combination of traditional PIC® Microcontroller (MCU) features (peripherals, Harvard architecture, RISC) and new DSP capabilities. The dsPIC DSC 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 compiler can access constant data in program memory through the program space visibility window.

4.4 STARTUP AND INITIALIZATION

Two C run-time startup modules are included in the `libpic30.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 (`crt0.o`) is linked by default and performs the following:

1. The Stack Pointer (W15) and Stack Pointer Limit register (SPLIM) are initialized, using values provided by the linker or a custom linker script. For more information, see **Section 4.8 "Software Stack"**.
2. If a `.const` section is defined, it is mapped into the program space visibility window by initializing the PSVPAG and CORCON registers. Note that a `.const` section is defined when the "Constants in code space" option is selected in MPLAB IDE, or the default `-mconst-in-code` option is specified on the compiler command line.
3. The data initialization template in section `.dinit` is read, causing all uninitialized sections to be cleared, and all initialized sections to be initialized with values read from program memory. The data initialization template is created by the linker, and supports the standard sections listed in **Section 4.3 "Address Spaces"**, as well as the user-defined sections.

> **Note:** The persistent data section `.pbss` is never cleared or initialized.

4. If the application has defined `user_init` functions, 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 (`crt1.o`) 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.

Source code (in dsPIC DSC assembly language) for both modules is provided in the `c:\Program Files\Microchip\MPLAB C30\src` directory. 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.

4.5 MEMORY SPACES

Static and external variables are normally allocated in general purpose data memory. Const-qualified variables will be allocated in general purpose data memory if the constants-in-data memory model is selected, or in program memory if the constants-in-code memory model is selected.

The compiler defines several special purpose memory spaces to match architectural features of 16-bit devices. Static and external variables may be allocated in the special purpose memory spaces through use of the `space` attribute, described in **Section 2.3.1 "Specifying Attributes of Variables"**.

`data`

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

`eds`

Allocate the variable in the extended data space. For devices that do not have extended data space, this is equivalent to `space(data)`. Variables in `space(eds)` will generally require special handling to access. Refer to **Chapter 6. "Additional C Pointer Types"** for more information.

`space(eds)` has been deprecated in favour of the `eds` attribute.

DD **`xmemory`** - dsPIC30F/dsPIC33F devices only

X data address space. Variables in X data space can be accessed using ordinary C statements. X data address space has special relevance for DSP-oriented libraries and/or assembly language instructions.

DD **`ymemory`** - dsPIC30F/dsPIC33F 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 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, or using the program space visibility window.

`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.

DD **eedata** - PIC24F MCUs, dsPIC30F/33F devices only

Data EEPROM space, a region of 16-bit wide non-volatile memory located at high addresses in program memory. Variables in eedata space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

DD **dma** - PIC24H MCUs, dsPIC33F DSCs only

DMA memory. Variables in DMA memory can be accessed using ordinary C statements and by the DMA peripheral.

4.6 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 dsPIC DSC device part that you are using and the type of memory usage.

TABLE 4-1: MEMORY MODEL COMMAND LINE OPTIONS

Option	Memory Definition	Description
`-msmall-data`	Up to 8 KB of data memory. **This is the default.**	Permits use of PIC18 like instructions for accessing data memory.
`-msmall-scalar`	Up to 8 KB of data memory. **This is the default.**	Permits use of PIC18 like instructions for accessing scalars in data memory.
`-mlarge-data`	Greater than 8 KB of data memory.	Uses indirection for data references.
`-msmall-code`	Up to 32 Kwords of program memory. **This is the default.**	Function pointers will not go through a jump table. Function calls use `RCALL` instruction.
`-mlarge-code`	Greater than 32 Kwords of program memory.	Function pointers might go through a jump table. Function calls use `CALL` instruction.
`-mconst-in-data`	Constants located in data memory.	Values copied from program memory by startup code.
`-mconst-in-code`	Constants located in program memory. **This is the default.**	Values are accessed via Program Space Visibility (PSV) data window.
`-mconst-in-aux-flash`	Constants in auxiliary FLASH	Values are accessed via Program Space visibility window.

The command-line options apply globally to the modules being compiled. Individual variables and functions can be declared as `near`, `far` or `eds` to better control the code generation. For information on setting individual variable or function attributes, see **Section 2.3.1 "Specifying Attributes of Variables"** and **Section 2.3.2 "Specifying Attributes of Functions"**.

4.6.1 Near and Far Data

If variables are allocated in the near data section, the compiler is often able to generate better (more compact) code than if the variables are not allocated in the near data section. If all variables for an application can fit within the 8 KB of near data, then the compiler can be requested to place them there by using the default `-msmall-data` command line option when compiling each module. If the amount of data consumed by scalar types (no arrays or structures) totals less than 8 KB, the default `-msmall-scalar` may be used. This requests that the compiler arrange to have just the scalars for an application allocated in the near data section.

If neither of these global options is suitable, then the following alternatives are available.

1. It is possible to compile some modules of an application using the `-mlarge-data` or `-mlarge-scalar` command line options. In this case, only the variables used by those modules will be allocated in the far data section. If this alternative is used, then care must be taken when using externally defined variables. If a variable that is used by modules compiled using one of these options is defined externally, then the module in which it is defined must also be compiled using the same option, or the variable declaration and definition must be tagged with the far attribute.
2. If the command line options `-mlarge-data` or `-mlarge-scalar` have been used, then an individual variable may be excluded from the `far` data space by tagging it with the near attribute.
3. Instead of using command-line options, which have module scope, individual variables may be placed in the far data section by tagging them with the `far` attribute.

The linker will produce an error message if all near variables for an application cannot fit in the 8K near data space.

4.6.2 Near and Far Code

Functions that are near (within a radius of 32 Kwords of each other) may call each other more efficiently that those which are not. If it is known that all functions in an application are near, then the default `-msmall-code` command line option can be used when compiling each module to direct the compiler to use a more efficient form of the function call.

If this default option is not suitable, then the following alternatives are available:

1. It is possible to compile some modules of an application using the `-msmall-code` command line option. In this case, only function calls in those modules will use a more efficient form of the function call.
2. If the `-msmall-code` command-line option has been used, then the compiler may be directed to use the long form of the function call for an individual function by tagging it with the `far` attribute.
3. Instead of using command-line options, which have module scope, the compiler may be directed to call individual functions using a more efficient form of the function call by tagging their declaration and definition with the `near` attribute.

The `-msmall-code` command-line option differs from the `-msmall-data` command-line option in that in the former case, the compiler does nothing special to ensure that functions are allocated near one another, whereas in the latter case, the compiler will allocate variables in a special section.

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.

4.7 LOCATING CODE AND DATA

As described in **Section 4.3 "Address Spaces"**, the compiler arranges for code to be placed in the `.text` section, and data to be placed in one of several named sections, depending on the memory model used and whether or not the data is initialized. When modules are combined at link time, the linker determines the starting addresses of the various sections based on their attributes.

Cases may arise when a specific function or 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 2.3 "Keyword Differences"**. For example, to locate function `PrintString` at address 0x8000 in program memory:

```
int __attribute__ ((address(0x8000))) PrintString (const char *s);
```

Likewise, to locate variable `Mabonga` at address 0x1000 in data memory:

```
int __attribute__ ((address(0x1000))) Mabonga = 1;
```

Another way to locate code or data is by placing the function or 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 code or 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 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.

Similarly, 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 function 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.

4.8 SOFTWARE STACK

The dsPIC DSC device dedicates register W15 for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. The stack grows upward, towards higher memory addresses.

The dsPIC DSC device also supports stack overflow detection. If the Stack Pointer Limit register, SPLIM, is initialized, the device will test for overflow on all stack operations. If an overflow should occur, the processor will initiate a stack error exception. By default, this will result in a processor reset. Applications may also install a stack error exception handler by defining an interrupt function named `_StackError`. See **Chapter 8. "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® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317) 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:

<verbose>

```
  .section *,data,stack
  .space 0x100
```

<end>

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® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317) for more information.

4.9 THE C STACK USAGE

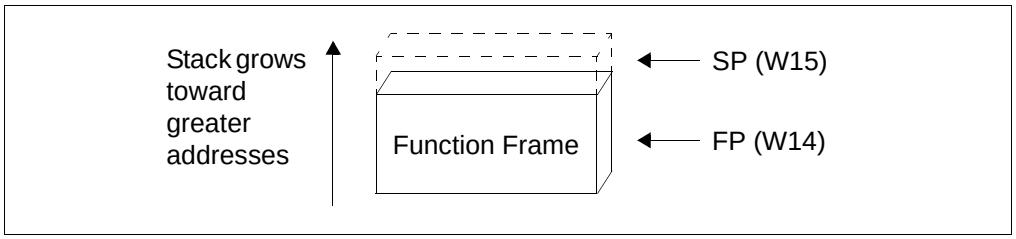
The C compiler uses the software stack to:

- 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.

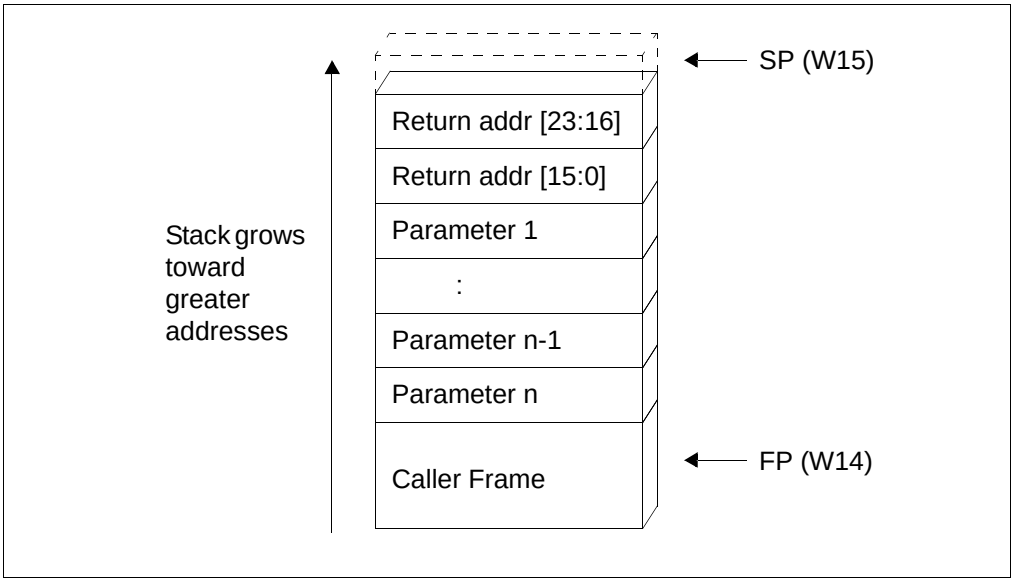
FIGURE 4-1: STACK AND FRAME POINTERS



The C run-time startup modules (`crt0.o` and `crt1.o` in `libpic30.a`) initialize the Stack Pointer W15 to point to the bottom of the stack and initialize the Stack Pointer Limit register to point to the top of the stack. The stack grows up and if it should grow beyond the value in the Stack Pointer Limit register, then a stack error trap will be taken. The user may initialize the Stack Pointer Limit register to further restrict stack growth.

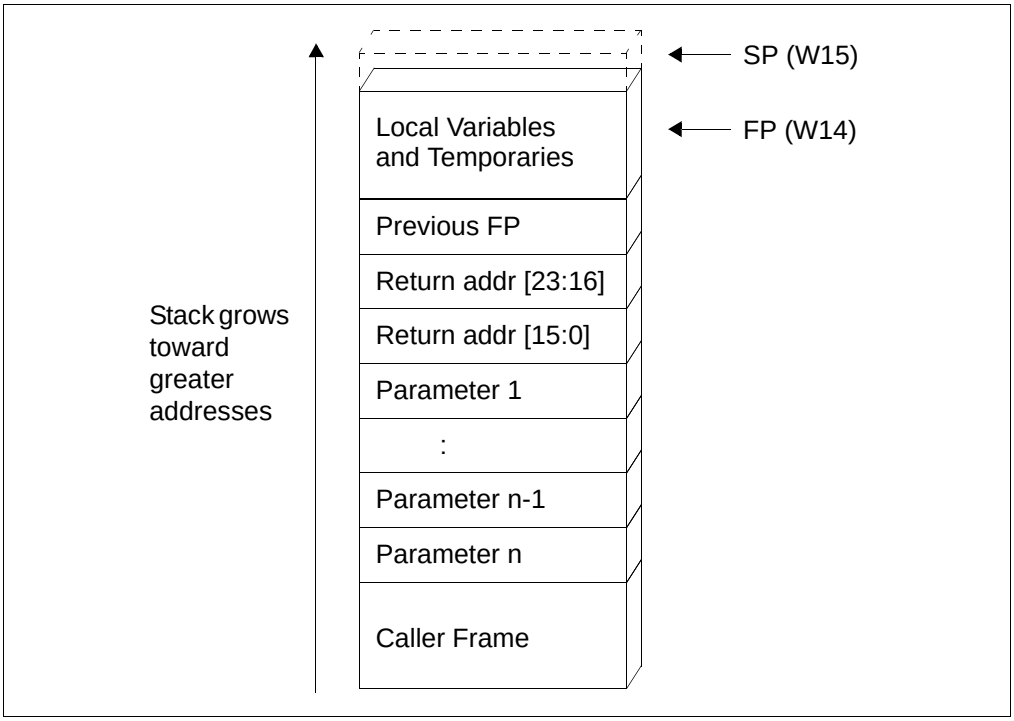
The following diagrams illustrate the steps involved in calling a function. Executing a `CALL` or `RCALL` instruction pushes the return address onto the software stack. See Figure 4-2.

FIGURE 4-2: CALL OR RCALL



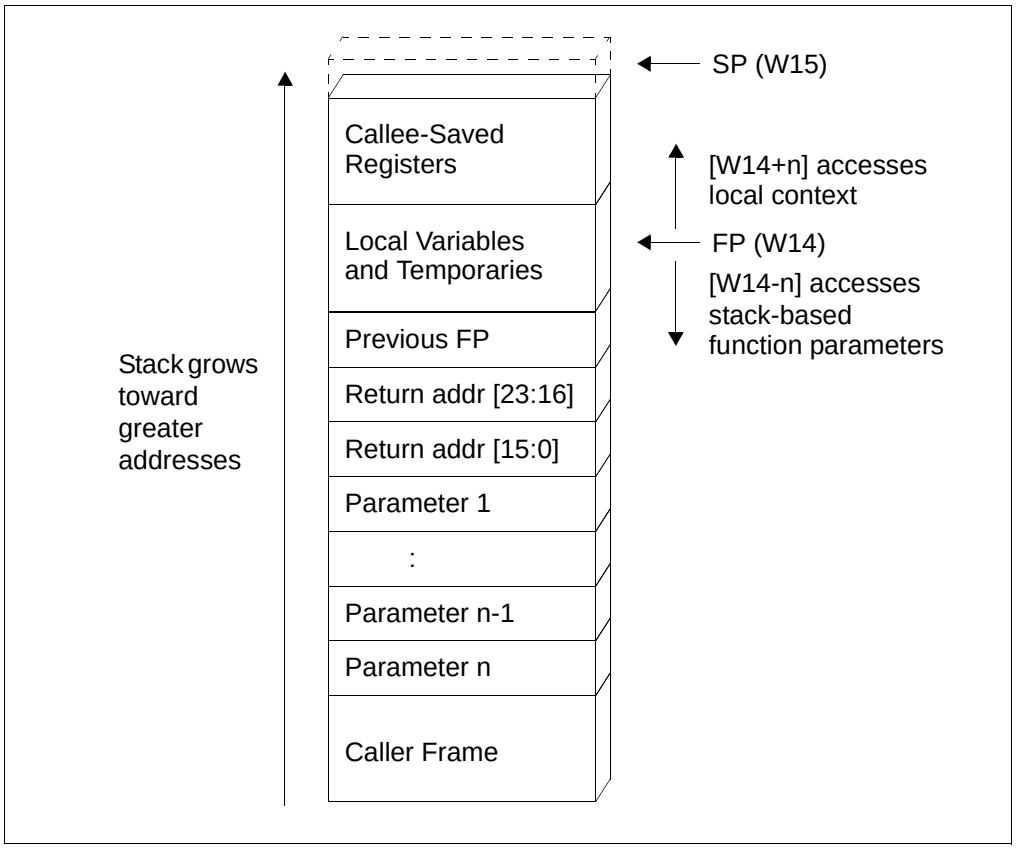
The called function (callee) can now allocate space for its local context (Figure 4-3).

FIGURE 4-3: CALLEE SPACE ALLOCATION



Finally, any callee-saved registers that are used in the function are pushed (Figure 4-4).

FIGURE 4-4: PUSH CALLEE-SAVED REGISTERS



4.10 THE C HEAP USAGE

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:

```
pic30-gcc foo.c -Wl,--heap=512
```

The linker allocates the heap immediately below the stack.

If you use a standard C library input/output function, then a heap must be allocated. If `stdout` is the only file that you use, then the heap size can be zero, that is, use the command-line option:

```
-Wl,--heap=0
```

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, 514 bytes of heap space is required. If there is insufficient heap memory for the buffer, then the file will be opened in unbuffered mode.

4.11 FUNCTION CALL CONVENTIONS

When calling a function:

- Registers W0-W7 are caller saved. The calling function must push these values onto the stack for the register values to be preserved.
- 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.

TABLE 4-2: REGISTERS REQUIRED

Data Type	Number of Registers Required
`char`	1
`int`	1
`short`	1
`pointer`	1
`long`	2 (contiguous – aligned to even numbered register)
`float`	2 (contiguous – aligned to even numbered register)
`double*`	2 (contiguous – aligned to even numbered register)
`long double`	4 (contiguous – aligned to quad numbered register)
`structure`	1 register per 2 bytes in structure

* `double` is equivalent to `long double` if `-fno-short-double` is used.

Parameters are placed in the first aligned contiguous register(s) that are available. The calling function must preserve the parameters, if required. Structures do not have any alignment restrictions; a structure parameter will occupy registers if there are enough registers to hold the entire structure. Function results are stored in consecutive registers, beginning with W0.

4.11.1 Function Parameters

The first eight working registers (W0-W7) are used for function parameters.Parameters are allocated to registers in left-to-right order, and a parameter is assigned to the first available register that is suitably aligned.

In the following example, all parameters are passed in registers, although not in the order that they appear in the declaration. This format allows the compiler to make the most efficient use of the available parameter registers.

EXAMPLE 4-1: 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 4-2: 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 4-3: 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 otherwise told not to do so), and stack-based parameters will be accessed via the Frame Pointer register (W14). The above example, b will be accessed from W14-22. The Frame Pointer offset of negative 22 has been calculated (refer to Figure 4-4) by removing 2 bytes for the previous FP, 4 bytes for the return address, followed by 16 bytes for b.

When no Frame Pointer is used, the assembly programmer must know how much stack space has been used since entry to the procedure. If no further stack space is used, the calculation is similar to the above. b would be accessed via W15-20; 4 bytes for the return address and 16 bytes to access the start of b.

4.11.2 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.

4.11.3 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.

4.12 REGISTER CONVENTIONS

Specific registers play specific roles in the C run-time environment. Register variables use one or more working registers, as shown in Table 4-3.

TABLE 4-3: 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}. The lower-numbered register contains the least significant 16 bits of the value.
`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}. The lower-numbered register contains the least significant 16 bits of the value. 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}. The lower-numbered register contains the least significant 16 bits of the significant.
`double*`	A pair of contiguous registers, the first of which is a register from the set {W0, W2, W4, W6, W8, W10, W12}. The lower-numbered register contains the least significant 16 bits of the significant.
`long double`	A quadruplet of contiguous registers, the first of which is a register from the set {W0, W4, W8}. The lower-numbered register contains the least significant 16 bits of the significant.
`structure`	1 contiguous register per 2 bytes in the structure.

* `double` is equivalent to `long double` if `-fno-short-double` is used.

4.13 BIT REVERSED AND MODULO ADDRESSING

The compiler does not directly support the use of bit reversed and modulo addressing. If either of these addressing modes is enabled for a register, then it is the programmer's responsibility to ensure that the compiler does not use that register as a pointer. 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 assembly language functions. The `aligned` attribute may be used to define arrays that are positioned for use as incrementing modulo buffers. The `reverse` attribute may be used to define arrays that are positioned for use as decrementing modulo buffers. For more information on these attributes, see **Section 2.3 "Keyword Differences"**. For more information on modulo addressing, see chapter 3 of the "*dsPIC30F Family Reference Manual*" (DS70046).

4.14 PROGRAM SPACE VISIBILITY (PSV) USAGE

By default, the compiler will automatically arrange for strings and `const`-qualified initialized variables to be allocated in the `.const` section, which is mapped into the PSV window. Then PSV management is left up to compiler management, which does not move it, limiting the size of accessible program memory to the size of the PSV window itself.

Alternatively, an application may take control of the PSV window for its own purposes. The advantage of directly controlling the PSV usage in an application is that it affords greater flexibility than having a single `.const` section permanently mapped into the PSV window. The disadvantage is that the application must manage the PSV control registers and bits. Specify the `-mconst-in-data` option to direct the compiler not to use the PSV window.

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 section `.const`, 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 **Section 2.3 "Keyword Differences"**.

For more on PSV usage, see the "*MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317).

4.14.1 Boot and Secure Constants

Two new psv constant sections will be defined: `.boot_const` and `.secure_const`. These sections are analogous to the generic section `.const`, except that the compiler uses them independently of the user-selectable constants memory model.

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. Consequently, PSVPAG and CORCONbits.PSV must become compiler managed resources. 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.

4.14.2 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 can not 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.

4.14.3 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`. The logic for making that association is already supported in the compiler for named PSV sections.

4.14.4 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)`.

Compiler v3.0 will provide automatic support for multiple values of PSVPAG. Variables declared with `space(auto_psv)` may be combined with secure segment constant variables and/or managed psv pointer variables in the same source file. Precompiled objects that assume a single, pre-set value of PSVPAG will be link-compatible with objects that define secure segment psv constants or managed psv variables.

Even though PSVPAG is now considered to be a compiler-managed resource, there is no change to the function calling conventions. Objects and libraries created with earlier versions are compatible with 3.0 objects, with the exception of some Interrupt Service Routines as noted in **Section 8.10 “PSV Usage with Interrupt Service Routines”**.

4.15 USING LARGE ARRAYS

The compiler option `-mlarge-arrays` allows you to define and access arrays larger than 32K. You must ensure that there is enough space to allocate such an array by nominating a memory space large enough to contain such an object.

Using this option will have some effect on how code is generated as it effects the definition of the `size_t` type, increasing it to an `unsigned long int`. If used as a global option, this will affect many operations used in indexing (making the operation more complex). Using this option locally may effect how variables can be accessed. With these considerations in mind, using large arrays is requires careful planning. This section discusses some techniques for its use.

Two things occur when the `-mlarge-arrays` option is selected:

1. The compiler generates code in a different way for accessing arrays.
2. The compiler defines the `size_t` type to be `unsigned long int`.

Item 1 can have a negative effect on code size, if used throughout the whole program. It is possible to only compile a single module with this option and have it work, but there are limitations which will be discussed shortly.

Item 2 affects the calling convention when external functions receive or return objects of type `size_t`. The compiler provides libraries built to handle a larger `size_t` and these objects will be selected automatically by the linker (provided they exist).

Mixing `-mlarge-arrays` and normal-sized arrays together is relatively straightforward and might be the best way to make use of this new 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 accessory 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 accessory 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:

```
/* 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];
}

/* 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));
}
```

Chapter 5. Data Types

5.1 INTRODUCTION

This section discusses the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30) data types.

5.2 HIGHLIGHTS

Items discussed in this chapter are:

- Data Representation
- Integer
- Floating Point
- Pointers

5.3 DATA REPRESENTATION

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

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

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

0x100	0x101	0x102	0X103
0x78	0x56	0x34	0x12

As another example, the long value of `0x12345678` is stored in registers w4 and w5:

w4	w5
0x5678	0x1234

5.4 INTEGER

Table 5-1 shows integer data types are supported in the compiler.

TABLE 5-1: INTEGER DATA TYPES

Type	Bits	Min	Max
`char, signed char`	8	-128	127
`unsigned char`	8	0	255
`short, signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int, signed int`	16	-32768	32767
`unsigned int`	16	0	65535
`long, signed long`	32	-2^{31}	$2^{31} - 1$
`unsigned long`	32	0	$2^{32} - 1$
`long long**, signed long long**`	64	-2^{63}	$2^{63} - 1$
`unsigned long long**`	64	0	$2^{64} - 1$

** ANSI-89 extension

For information on implementation-defined behavior of integers, see **Section A.7 "Integers"**.

5.5 FLOATING POINT

The compiler uses the IEEE-754 format. Table 5-2 shows floating point data types are supported.

TABLE 5-2: FLOATING POINT DATA TYPES

Type	Bits	E Min	E Max	N Min	N Max
`float`	32	-126	127	2^{-126}	2^{128}
`double*`	32	-126	127	2^{-126}	2^{128}
`long double`	64	-1022	1023	2^{-1022}	2^{1024}

E = Exponent
N = Normalized (approximate)
* `double` is equivalent to `long double` if `-fno-short-double` is used.

For information on implementation-defined behavior of floating point numbers, see section **Section A.8 "Floating Point"**.

5.6 POINTERS

All standard pointers are 16 bits wide. This is sufficient for full data space access (64 KB) and the small code model (32 Kwords of code.) In the large code model (>32 Kwords of code), pointers may resolve to "handles"; that is, the pointer is the address of a `GOTO` instruction which is located in the first 32 Kwords of program space.

A set of special purpose, 32-bit data pointers are also available. See **Chapter 6. "Additional C Pointer Types"** for more information.

Chapter 6. Additional C Pointer Types

6.1 INTRODUCTION

MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30) offers some extended pointer modes to help access more of the unique features of Microchip's 16-bit product architecture. Extended pointers and their use will be covered in this chapter.

- Managed PSV Pointers – for reading more data through the PSV
- PMP Pointers – for accessing data via the PMP peripheral (where available)
- External Pointers – for accessing external memory in a user-defined fashion
- Extended Data Space Pointers – for accessing variables declared in a variety of different memory spaces

Although the concentration will be on pointer access, defining variables and ensuring that the data is allocated in the correct region of the 16-bit architectures (bi-polar) memory is also covered.

This chapter will make use of concepts introduced in **Chapter 2. "Differences Between 16-Bit Device C and ANSI C"**.

6.2 MANAGED PSV POINTERS

The dsPIC30F/33F and PIC24F/H families of processors contain hardware support for accessing data from within program Flash using a hardware feature that is commonly called Program Space Visibility (PSV). More detail about how PSV works can be found in device data sheets or family reference manuals. Also, see **Section 4.14 "Program Space Visibility (PSV) Usage"** and **Section 8.10 "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. 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.

6.2.1 Defining Data for Managed PSV Access

Chapter 2. "Differences Between 16-Bit Device C and ANSI C" introduces C extensions which allows the identification of extra information for a variable or function. The compiler provides the `space` attribute to help place variables into different areas (spaces) of memory.

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.

6.2.2 Managed PSV Access

Just placing something into Flash using the space attribute does not mean the compiler will be able to manage the access. The compiler requires that you identify variables in a special way. This is done because the managed PSV can be less efficient than managing the PSVPAG by hand (though far less complicated).

The compiler introduces several new qualifiers (CV-qualifiers for the language lawyers in the audience). Like const-volatile qualifier, the new qualifiers can be applied to pointers or objects. These 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

Typically there is no need to specify __psv__ or __prog__ for an object placed in space(auto_psv).

Moving the FLASH_variable from the previous section into a normal Flash space and requesting that the compiler manage the 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 PSVPAG 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;
```

6.2.3 ISR Considerations

A data access using managed PSV pointers is definitely not atomic, meaning it can take several instructions to complete the access. Care should be taken if an access should not be interrupted.

Furthermore an interrupt service routine (ISR) never really knows what the current state of the PSVPAG register will be. Unfortunately the compiler is not really in any position to determine whether or not this is important in all cases.

The compiler will make the simplifying assumption that the writer of the interrupt service routine will know whether or not the automatic, compiler managed PSVPAG is required by the ISR. This is required to access any constant data in the auto_psv space or any string literals or constants when the default -mconst-in-code option is selected. When defining an interrupt service routine, it is best to specify whether or not it is necessary to assert the default setting of the PSVPAG SFR.

This is achieved by adding a further attribute to the interrupt function definition:

- `auto_psv` - the compiler will set the PSVPAG register to the correct value for accessing the `auto_psv` space, ensuring that it is restored when exiting the ISR
- `no_auto_psv` - the compiler will not set the PSVPAG register

For example:

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {
  IFS0bits.T1IF = 0;
}
```

Current code (that does not assert the `auto_psv` attribute) may not execute properly unless recompiled. When recompiled, if no indication is made, the compiler will generate a warning message and select the `auto_psv` model.

The choice is provided so that, if you are especially conscious of interrupt latency, you may select the best option. Saving and setting the PSVPAG will consume approximately 3 cycles at the entry to the function and one further cycle to restore the setting upon exit from the function.

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

6.3 PMP POINTERS

Some devices contain a Parallel Master Port (PMP) peripheral which allows the connection of various memory and non-memory devices directly to the device. Access to the peripheral is controlled via a selection of peripherals. More information about this peripheral can be found in the Family Reference Manual or device-specific data sheets.

Note: PMP attributes are not supported on devices with EPMP. Use EDS.

PMP pointers are similar to managed PSV pointers as described in the previous section. These pointers make it easier to read or write data using the PMP.

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 in order to access the variable.

To use this feature:

- Initialize PMP - define the initialization function: `void __init_PMP(void)`
- Declare a New Memory Space
- Define Variables within PMP Space

6.3.1 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.

The toolsuite, if PMP is used, 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 */
}
```

6.3.2 Declare a New Memory Space

The compiler toolsuite requires information about each additional memory being attached via the PMP. In order for the 16-bit device linker to be able to properly assign memory, information about the size of memory available and the number of chip-selects needs to be provided.

In **Chapter 2. "Differences Between 16-Bit Device C 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 these declaration appear in a common header file so that the declaration is available to all translation units.

6.3.3 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 previous 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 accessing 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.

6.4 EXTERNAL POINTERS

Not all of Microchip's 16-bit devices have a PMP peripheral, or not all memories are suitable for attaching to a parallel port (serial memories sold by Microchip, for example). The toolsuite provides a more general interface to any external memory, although, as will be seen, the memory does not have to be external.

Like PMP memory space, the tool-chain needs to learn about external memories that are being attached. Unlike PMP, however, the compiler does not know how to access these memories. A mechanism is provided by which an application can specify how to access such memories.

External pointers (and their addresses) consume 32 bits. The largest attachable memory is 64K (16 bits); the other 16 bits is used to uniquely identify the memory. A total of 64K (16 bits) of these may be (theoretically) attached.

To use this feature:

- Declare a New Memory Space
- Define Variables within an External Space
- Define How to Access Memory Spaces

As an example:

- An External Example

6.4.1 Declare a New Memory Space

This is very similar to declaring a new memory space for PMP access.

The 16-bit toolsuite requires information about each external memory. In order for 16-bit device linker to be able to properly assign memory, information about the size of memory available and, optionally the origin of the memory, needs to be provided.

In **Chapter 2. "Differences Between 16-Bit Device C and ANSI C"** the new 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`.

6.4.2 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 pointers, we have qualified the variable with a new type qualifier `__external__`. When attached to a variable or pointer it instructs the compiler to generate the correct sequence for accessing.

Now that a 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.

6.4.3 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 (5) functions may be defined for reading and five (5) for writing. One each of these is a generic function and will be called if any of the other four is not defined but is 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.

6.4.3.1 FUNCTIONS FOR READING

read_external

```
void __read_external(unsigned int address,
  unsigned int memory_space,
  void *buffer,
  unsigned int len)
```

This function is a generic Read function and will be called if one of the next functions are required but not defined. This function should perform the steps necessary to fill `len` bytes of memory in the `buffer` from the external memory named `memory_space` starting at address `address`.

read_external8

```
unsigned char __read_external8(unsigned int address,
  unsigned int memory_space)
```

Read 8 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access an 8-bit sized object.

read_external16

```
unsigned int __read_external16(unsigned int address,
  unsigned int memory_space)
```

Read 16 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access an 16-bit sized object.

read_external32

```
unsigned long __read_external32(unsigned int address,
  unsigned int memory_space)
```

Read 32 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access a 32-bit sized object, such as a `long` or `float` type.

read_external64

```
unsigned long long __read_external64(unsigned int address,
  unsigned int memory_space)
```

Read 64 bits from external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to access a 64-bit sized object, such as a `long long` or `long double` type.

6.4.3.2 FUNCTIONS FOR WRITING

write_external

```
void __write_external(unsigned int address,
  unsigned int memory_space,
  void *buffer,
  unsigned int len)
```

This function is a generic Write function and will be called if one of the next functions are required but not defined. This function should perform the steps necessary to write `len` bytes of memory from the `buffer` to the external memory named `memory_space` starting at address `address`.

write_external8

```
void __write_external8(unsigned int address,
  unsigned int memory_space,
  unsigned char data)
```

Write 8 bits of `data` to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write an 8-bit sized object.

write_external16

```
void __write_external16(unsigned int address,
  unsigned int memory_space,
  unsigned int data)
```

Write 16 bits of `data` to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write an 16-bit sized object.

write_external32

```
void __write_external32(unsigned int address,
  unsigned int memory_space,
  unsigned long data)
```

Write 32 bits of `data` to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write a 32-bit sized object, such as a `long` or `float` type.

write_external64

```
void __write_external64(unsigned int address,
  unsigned int memory_space,
  unsigned long long data)
```

Write 64 bits of `data` to external memory space `memory_space` starting from address `address`. The compiler would like to call this function if trying to write a 64-bit sized object, such as a `long long` or `long double` type.

6.4.4 An External Example

The following snippets of example come from a working example (in the Examples folder.)

This example implements, using *external* memory, addressable bit memory. In this case each bit is stored in real data memory, not off chip. The code will define an `external` memory of 512 units and map accesses using the appropriate read and write function to one of 64 bytes in local data memory.

First the external memory is defined:

```
 extern unsigned int bit_memory
__attribute__((space(external(size(512)))));
```

Next appropriate read and write functions are defined. These functions will make use of an array of memory that is reserved in the normal way.

```
 static unsigned char real_bit_memory[64];
 unsigned char __read_external8(unsigned int address,
                                unsigned int memory_space) {
   if (memory_space == bit_memory) {
     /* an address within our bit memory */
     unsigned int byte_offset, bit_offset;
     byte_offset = address / 8;
     bit_offset = address % 8;
     return (real_bit_memory[byte_offset] >> bit_offset) & 0x1;
   } else {
     fprintf(stderr,"I don't know how to access memory space: %d\n",
             memory_space);
   }
   return 0;
 }
 void __write_external8(unsigned int address,
                        unsigned int memory_space,
                        unsigned char data) {
   if (memory_space == bit_memory) {
     /* an address within our bit memory */
     unsigned int byte_offset, bit_offset;
     byte_offset = address / 8;
     bit_offset = address % 8;
     real_bit_memory[byte_offset] &= (~(1 << bit_offset));
     if (data & 0x1) real_bit_memory[byte_offset] |=
        (1 << bit_offset);
```

```
  } else {
    fprintf(stderr,"I don't know how to access memory space: %d\n",
            memory_space);
  }
}
```

These functions work in a similar fashion:

- if accessing `bit_memory`, then
  - determine the correct byte offset and bit offset
  - read or write the appropriate place in the `real_bit_memory`
- otherwise access another memory (whose access is unknown)

With the two major pieces of the puzzle in place, generate some variables and accesses:

```
__external__ unsigned char bits[NUMBER_OF_BITS]
             __attribute__((space(external(bit_memory))));
// inside main
  __external__ unsigned char *bit;
  bit = bits;
  for (i = 0; i < 512; i++) {
    printf("%d ",*bit++);
  }
```

Apart from the `__external__` CV-qualifiers, ordinary C statements can be used to define and access variables in the external memory space.

6.5 EXTENDED DATA SPACE POINTERS

Extended data space pointers allow you to easily access variables that have been placed in a variety of different memory spaces. These include: `space(data)` (and its subsets), `space(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__((space(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 `int`s, located in `space(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 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)

Chapter 7. Device Support Files

7.1 INTRODUCTION

This section discusses device support files used in support of compilation using the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30).

7.2 HIGHLIGHTS

Items discussed in this chapter are:

- Processor Header Files
- Register Definition Files
- Using SFRs
- Using Macros
- Accessing EEDATA from C Code – PIC24F MCUS, dsPIC30F/33F DSCs only

7.3 PROCESSOR HEADER FILES

The processor header files are distributed with the language tools. These header files define the available Special Function Registers (SFRs) for each dsPIC DSC device. To use a header file in C, use;

```
#include <p30fxxxx.h>
```

where `xxxx` corresponds to the device part number. The C header files are distributed in the `support\h` directory.

Inclusion of the header file is necessary in order to use SFR names (e.g., `CORCONbits`).

For example, the following module, compiled for the dsPIC30F2010 part, includes two functions: one for enabling the PSV window, and another for disabling the PSV window.

```
#include <p30f2010.h>
void
EnablePSV(void)
{
    CORCONbits.PSV = 1;
}
void
DisablePSV(void)
{
    CORCONbits.PSV = 0;
}
```

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__((__near__));
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__((__near__));
```

Note: The symbols CORCON and CORCONbits refer to the same register and will resolve to the same address at link time.

7.4 REGISTER DEFINITION FILES

The processor header files described in **Section 7.3 "Processor 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\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 exists a file named app2010.c, which contains an application for the dsPIC30F2010 part, then it may be compiled and linked using the following command line:

```
pic30-gcc -o app2010.o -T p30f2010.gld app2010.c
```

The -o command-line option names the output COFF executable file, and the -T option gives the name for the dsPIC30F2010 part. If p30f2010.gld is not found in the current directory, the linker searches in its known library paths. For the default installation, the linker scripts are included in the PIC30_LIBRARAY_PATH. For reference see **Section 3.6 "Environment Variables"**.

7.5 USING SFRS

There are three steps to follow when using SFRs in an application.

1. Include the processor header file for the appropriate device. This provides the source code with the SFRs that are available for that device. For instance, the following statement includes the header files for the dsPIC30F6014 part:
   `#include <p30f6014.h>`
2. Access SFRs like any other C variables. The source code can write to and/or read from the SFRs.
   For example, the following statement clears all the bits to zero in the special function register for Timer1.
   `TMR1 = 0;`
   This next statement represents the 15th bit in the T1CON register which is the "timer on" bit. It sets the bit named `TON` to `1` which starts the timer.
   `T1CONbits.TON = 1;`
3. Link with the register definition file or linker script for the appropriate device. The linker provides the addresses of the SFRs. (Remember the bit structure will have the same address as the SFR at link time.) Example 6.1 would use:
   `p30f6014.gld`

See "*MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317) for more information on using linker scripts.

The following example is a sample real time clock. It uses several SFRs. Descriptions for these SFRs are found in the `p30f6014.h` file. This file would be linked with the device specific linker script which is `p30f6014.gld`.

EXAMPLE 7-1: SAMPLE REAL-TIME CLOCK

```
/*
** Sample Real Time Clock for dsPIC
**
**  Uses Timer1, TCY clock timer mode
**  and interrupt on period match
*/

#include <p30f6014.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__)) _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;
  }
```

7.6 USING MACROS

Processor header files define, in addition to special function registers, useful macros for the 16-bit family of devices.

- Configuration Bits Setup Macros
- Inline Assembly Usage Macros
- Data Memory Allocation Macros
- ISR Declaration Macros

7.6.1 Configuration Bits Setup Macros

Macros are provided that can be used to set configuration bits. For example, to set the FOSC bit using a macro, the following line of code can be inserted before the beginning of your C source code:

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

This would enable the external clock with the PLL set to 16x and enable clock switching and fail-safe clock monitoring.

Similarly, to set the FBORPOR bit:

```
_FBORPOR(PBOR_ON & BORV_27 & PWRT_ON_64 & MCLR_DIS);
```

This would enable Brown-out Reset at 2.7 Volts and initialize the Power-up timer to 64 milliseconds and configure the use of the $\overline{\text{MCLR}}$ pin for I/O.

For a complete list of settings valid for each configuration bit, refer to the processor header file.

7.6.2 Inline Assembly Usage Macros

Some Macros used to define assembly code in C are listed below:

```
#define Nop()    {__asm__ volatile ("nop");}
#define ClrWdt() {__asm__ volatile ("clrwdt");}
#define Sleep()  {__asm__ volatile ("pwrsav #0");}
#define Idle()   {__asm__ volatile ("pwrsav #1");}
```

7.6.3 Data Memory Allocation Macros

Macros that may be used to allocate space in data memory are discussed below. There are two types: those that require an argument and those that do not.

The following macros require an argument N that specifies alignment. N must be a power of two, with a minimum value of 2.

```
#define _XBSS(N)   __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N)  __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N)   __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N)  __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata),  aligned(N)))
```

For example, to declare an uninitialized array in X memory that is aligned to a 32-byte address:

```
int _XBSS(32) xbuf[16];
```

To declare an initialized array in data EEPROM without special alignment:

```
int _EEDATA(2) table1[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

The following macros do not require an argument. They can be used to locate a variable in persistent data memory or in near data memory.

```
#define _PERSISTENT __attribute__((persistent))
#define _NEAR       __attribute__((near))
```

For example, to declare two variables that retain their values across a device reset:

```
int _PERSISTENT var1,var2;
```

7.6.4 ISR Declaration Macros

The following macros can be used to declare Interrupt Service Routines (ISRs):

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

For example, to declare an ISR for the timer0 interrupt:

```
void _ISR _INT0Interrupt(void);
```

To declare an ISR for the SPI1 interrupt with fast context save:

```
void _ISRFAST _SPI1Interrupt(void);
```

> **Note:** ISRs will be installed into the interrupt vector tables automatically if the reserved names listed in **Section 8.4 "Writing the Interrupt Vector"** are used.

DD 7.7 ACCESSING EEDATA FROM C CODE – PIC24F MCUS, dsPIC30F/33F DSCS ONLY

The compiler provides some convenience macro definitions to allow placement of data into the devices EE data area. This can be done quite simply:

```
int _EEDATA(2) user_data[] =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

`user_data` will be placed in the EE data space 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).

7.7.1 Accessing EEDATA via the 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 PSVPAG register must be set to the appropriate address for the program memory to be accessed. For EE data this will be 0xFF, but it is best to use the `__builtin_psvpage()` function.
- 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 7-2: EEDATA ACCESS VIA PSV

```
#include <p30fxxxx.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 PSVPAG is changed, variables in EE data space may be read by referring to them as normal C variables, as shown in the example.

Note:	This access model is not compatible with the compiler-managed PSV (`-mconst-in-code`) model. You should be careful to prevent conflict.

7.7.2 Accessing EEDATA using TBLRDx instructions

The TBLRDx instructions are not directly supported by the compiler, but they can be used via inline assembly. Like PSV accesses, a 23-bit address is formed from an SFR value and the address encoded as part of the instruction. To access the same memory as given in the previous example, the following code may be used:

To use the TBLRDx instructions:

- The TBLPAG register must be set to the appropriate address for the program memory to be accessed. For EE data, this will be 0x7F, but it is best to use the `__builtin_tblpage()` function.
- The TBLRDx instruction can be accessed from an `__asm__` statement or through one of the `__builtin_tblrd` functions; refer to the "*dsPIC30F/33F Programmer's Reference Manual*" (DS70157) for information on this instruction.

EXAMPLE 7-3: EEDATA ACCESS VIA TABLE READ

```
#include <p30fxxxx.h>
#define eedata_read(src, offset, dest) {  \
  register int eedata_addr;               \
  register int eedata_val;                \
                                          \
  eedata_addr = __builtin_tbloffset(&src)+offset; \
  __asm__("tblrdl [%1], %0" : "=r"(eedata_val) : "r"(eedata_addr)); \
  dest = eedata_val;                      \
  }

int main(void) {
  int value;

  TBLPAG = __builtin_tblpage(&user_data);

  eedata_read(user_data,2*sizeof(user_data[0]), value);
  if (value) ; /* do something */

  }
```

7.7.3 Additional Sources of Information

The device Family Reference Manuals have an excellent discussion on using the Flash program memory and EE data memory provided. These manuals also have information on run-time programming of both program memory and EE data memory.

Chapter 8. Interrupts

8.1 INTRODUCTION

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

The 16-bit devices support multiple interrupts from both internal and external sources. In addition, the devices allow high-priority interrupts to override any low priority interrupts that may be in progress.

The compiler provides full support for interrupt processing in C or inline assembly code. This chapter presents an overview of interrupt processing.

8.2 HIGHLIGHTS

Items discussed in this chapter are:

- **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.
- **Writing the Interrupt Vector** – The 16-bit devices use interrupt vectors to transfer application control when an interrupt occurs. An interrupt vector is a dedicated location in program memory that specifies the address of an ISR. Applications must contain valid function addresses in these locations to use interrupts.
- **Interrupt Service Routine Context Saving** – To handle returning from an interrupt to code in the same conditional state as before the interrupt, context information from specific registers must be saved.
- **Latency** – The time between when an interrupt is called and when the first ISR instruction is executed is the latency of the interrupt.
- **Nesting Interrupts** – The compiler supports nested interrupts.
- **Enabling/Disabling Interrupts** – Enabling and disabling interrupt sources occurs at two levels: globally and individually.
- **Sharing Memory Between Interrupt Service Routines and Mainline Code** – How to mitigate potential hazards when this technique is used.
- **PSV Usage with Interrupt Service Routines** – Using ISRs with managed psv pointers and CodeGuard Security psv constant sections.

8.3 WRITING AN INTERRUPT SERVICE ROUTINE

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

8.3.1 Guidelines for Writing ISRs

The guidelines for writing ISRs are:

- 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 not call other functions. This is recommended because of latency issues. See **Section 8.6 "Latency"** for more information.

8.3.2 Syntax for Writing ISRs

To declare a C function as an interrupt handler, tag the function with the interrupt attribute (see § 2.3 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 modified by an ISR should be qualified `volatile`.

The optional `irq` parameter allows you to place an interrupt vector at a specific interrupt, and the optional `altirq` parameter allows you to place an interrupt vector at a specified alternate interrupt. Each parameter requires a parenthesized interrupt ID number. (See **Section 8.4 "Writing the Interrupt Vector"** for a list of interrupt ID's.)

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.

8.3.3 Coding ISRs

The following prototype declares function `isr0` to be an interrupt handler:

```
void __attribute__((__interrupt__)) 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__(__save__(var1,var2))))
isr0(void);
```

To request the compiler to use the fast context save (using the `push.s` and `pop.s` instructions), tag the function with the `shadow` attribute (see **Section 2.3.2 “Specifying Attributes of Functions”**). For example:

```
void __attribute__((__interrupt__, __shadow__)) isr0(void);
```

8.3.4 Using Macros to Declare Simple ISRs

If an interrupt handler does not require any of the optional parameters of the interrupt attribute, then a simplified syntax may be used. The following macros are defined in the device-specific header files:

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

For example, to declare an interrupt handler for the timer0 interrupt:

```
#include <p30fxxxx.h>
void _ISR _INT0Interrupt(void);
```

To declare an interrupt handler for the SPI1 interrupt with fast context save:

```
#include <p30fxxxx.h>
void _ISRFAST _SPI1Interrupt(void);
```

8.4 WRITING THE INTERRUPT VECTOR

dsPIC30F/33F DSC and PIC24F/H MCU devices have two interrupt vector tables – a primary and an alternate table – each containing several exception vectors.

The exception sources have associated with them a primary and alternate exception vector, each occupying a program word, as shown in the tables below. The alternate vector name is used when the `ALTIVT` bit is set in the `INTCON2` register.

> **Note:** A device reset is not handled through the interrupt vector table. Instead, upon device reset, the program counter is cleared. This causes the processor to begin execution at address zero. By convention, the linker script constructs a `GOTO` instruction at that location which transfers control to the C run-time startup module.

To field an interrupt, a function's address must be placed at the appropriate address in one of the vector tables, and the function must preserve any system resources that it uses. It must return to the foreground task using a `RETFIE` processor instruction. Interrupt functions may be written in C. When a C function is designated as an interrupt handler, the compiler arranges to preserve all the system resources which the compiler uses, and to return from the function using the appropriate instruction. The compiler can optionally arrange for the interrupt vector table to be populated with the interrupt function's address.

To arrange for the compiler to fill in the interrupt vector to point to the interrupt function, name the function as denoted in the preceding table. For example, the stack error vector will automatically be filled if the following function is defined:

```
void __attribute__((__interrupt__)) _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__)) _AltStackError(void);
```

Again, note the use of the leading underscore.

For all interrupt vectors without specific handlers, a default interrupt handler will be installed. The default interrupt handler is supplied by the linker and simply resets the device. An application may also provide a default interrupt handler by declaring an interrupt function with the name `_DefaultInterrupt`.

The last nine interrupt vectors in each table do not have predefined hardware functions. The vectors for these interrupts may be filled by using the names indicated in the preceding table, or, names more appropriate to the application may be used, while still filling the appropriate vector entry by using the `irq` or `altirq` parameter of the interrupt attribute. For example, to specify that a function should use primary interrupt vector fifty-two, use the following:

```
void __attribute__((__interrupt__(__irq__(52)))) MyIRQ(void);
```

Similarly, to specify that a function should use alternate interrupt vector fifty-two, use the following:

```
void __attribute__((__interrupt__(__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 `__Interrupt`*n*, 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 `__AltInterrupt`*n*, where n is the vector number. Therefore, the C identifiers `_AltInterrupt45` through `_AltInterrupt53` are reserved by the compiler.

DD

8.4.1 dsPIC30F DSCs (Non-SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

TABLE 8-1: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input Capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output Compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_IC2Interrupt	_AltIC2Interrupt	IC2 Input Capture 2
5	_OC2Interrupt	_AltOC2Interrupt	OC2 Output Compare 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 Serial Peripheral Interface 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
11	_ADCInterrupt	_AltADCInterrupt	ADC convert completed
12	_NVMInterrupt	_AltNVMInterrupt	NMM NVM write completed
13	_SI2CInterrupt	_AltSI2CInterrupt	Slave I^2C™ interrupt
14	_MI2CInterrupt	_AltMI2CInterrupt	Master I^2C interrupt
15	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
16	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 0
17	_IC7Interrupt	_AltIC7Interrupt	IC7 Input Capture 7
18	_IC8Interrupt	_AltIC8Interrupt	IC8 Input Capture 8
19	_OC3Interrupt	_AltOC3Interrupt	OC3 Output Compare 3
20	_OC4Interrupt	_AltOC4Interrupt	OC4 Output Compare 4
21	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
22	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
23	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
24	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
25	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
26	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 Serial Peripheral Interface 2
27	_C1Interrupt	_AltC1Interrupt	CAN1 combined IRQ
28	_IC3Interrupt	_AltIC3Interrupt	IC3 Input Capture 3
29	_IC4Interrupt	_AltIC4Interrupt	IC4 Input Capture 4
30	_IC5Interrupt	_AltIC5Interrupt	IC5 Input Capture 5
31	_IC6Interrupt	_AltIC6Interrupt	IC6 Input Capture 6
32	_OC5Interrupt	_AltOC5Interrupt	OC5 Output Compare 5
33	_OC6Interrupt	_AltOC6Interrupt	OC6 Output Compare 6

TABLE 8-1: INTERRUPT VECTORS – dsPIC30F DSCs (NON-SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
34	_OC7Interrupt	_AltOC7Interrupt	OC7 Output Compare 7
35	_OC8Interrupt	_AltOC8Interrupt	OC8 Output Compare 8
36	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
37	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
38	_C2Interrupt	_AltC2Interrupt	CAN2 combined IRQ
39	_PWMInterrupt	_AltPWMInterrupt	PWM period match
40	_QEIInterrupt	_AltQEIInterrupt	QEI position counter compare
41	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC transfer completed
42	_LVDInterrupt	_AltLVDInterrupt	PLVD low voltage detected
43	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM fault A
44	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM fault B
45	_Interrupt45	_AltInterrupt45	Reserved
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_Interrupt49	_AltInterrupt49	Reserved
50	_Interrupt50	_AltInterrupt50	Reserved
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_Interrupt53	_AltInterrupt53	Reserved

DD 8.4.2 dsPIC30F DSCs (SMPS) Interrupt Vectors

The dsPIC30F SMPS devices are currently dsPIC30F1010, dsPIC30F2020 and dsPIC30F2023. All other dsPIC30F devices are non-SMPS.

TABLE 8-2: INTERRUPT VECTORS – dsPIC30F DSCs (SMPS)

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input Capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output Compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_Interrupt4	_AltInterrupt4	Reserved
5	_OC2Interrupt	_AltOC2Interrupt	OC2 Output Compare 2
6	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
7	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
8	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 Serial peripheral interface 1
9	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
10	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter

TABLE 8-2: INTERRUPT VECTORS – dsPIC30F DSCs (SMPS) (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
11	_ADCInterrupt	_AltADCInterrupt	ADC Convert completed
12	_NVMInterrupt	_AltNVMInterrupt	NVM write completed
13	_SI2CInterrupt	_AltSI2CInterrupt	Slave I^2C™ interrupt
14	_MI2CInterrupt	_AltMI2CInterrupt	Master I^2C interrupt
15	_Interrupt15	_AltInterrupt15	Reserved
16	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
17	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
18	_PWMSpEvent MatchInterrupt	_AltPWMSpEvent MatchInterrupt	PWM special event interrupt
19	_PWM1Interrupt	_AltPWM1Interrupt	PWM period match 1
20	_PWM2Interrupt	_AltPWM2Interrupt	PWM period match 2
21	_PWM3Interrupt	_AltPWM3Interrupt	PWM period match 3
22	_PWM4Interrupt	_AltPWM4Interrupt	PWM period match 4
23	_Interrupt23	_AltInterrupt23	Reserved
24	_Interrupt24	_AltInterrupt24	Reserved
25	_Interrupt25	_AltInterrupt25	Reserved
26	_Interrupt26	_AltInterrupt26	Reserved
27	_CNInterrupt	_AltCNInterrupt	Input Change Notification
28	_Interrupt28	_AltInterrupt28	Reserved
29	_CMP1Interrupt	_AltCMP1Interrupt	Analog comparator interrupt 1
30	_CMP2Interrupt	_AltCMP2Interrupt	Analog comparator interrupt 2
31	_CMP3Interrupt	_AltCMP3Interrupt	Analog comparator interrupt 3
32	_CMP4Interrupt	_AltCMP4Interrupt	Analog comparator interrupt 4
33	_Interrupt33	_AltInterrupt33	Reserved
34	_Interrupt34	_AltInterrupt34	Reserved
35	_Interrupt35	_AltInterrupt35	Reserved
36	_Interrupt36	_AltInterrupt36	Reserved
37	_ADCP0Interrupt	_AltADCP0Interrupt	ADC Pair 0 conversion complete
38	_ADCP1Interrupt	_AltADCP1Interrupt	ADC Pair 1 conversion complete
39	_ADCP2Interrupt	_AltADCP2Interrupt	ADC Pair 2 conversion complete
40	_ADCP3Interrupt	_AltADCP3Interrupt	ADC Pair 3 conversion complete
41	_ADCP4Interrupt	_AltADCP4Interrupt	ADC Pair 4 conversion complete
42	_ADCP5Interrupt	_AltADCP5Interrupt	ADC Pair 5 conversion complete
43	_Interrupt43	_AltInterrupt43	Reserved
44	_Interrupt44	_AltInterrupt44	Reserved
45	_Interrupt45	_AltInterrupt45	Reserved
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_Interrupt49	_AltInterrupt49	Reserved
50	_Interrupt50	_AltInterrupt50	Reserved
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_Interrupt53	_AltInterrupt53	Reserved

DD

8.4.3 PIC24F MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_ReservedTrap5	_AltReservedTrap5	Reserved
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input Capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output Compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_Interrupt4	_AltInterrupt4	Reserved
5	_IC2Interrupt	_AltIC2Interrupt	IC2 Input Capture 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 Output Compare 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 error interrupt
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 transfer completed interrupt
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 convert completed
14	_Interrupt14	_AltInterrupt14	Reserved
15	_Interrupt15	_AltInterrupt15	Reserved
16	_SI2C1Interrupt	_AltSI2C1Interrupt	Slave I^2C™ interrupt 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	Master I^2C interrupt 1
18	_CompInterrupt	_AltCompInterrupt	Comparator interrupt
19	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
20	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
21	_Interrupt21	_AltInterrupt21	Reserved
22	_Interrupt22	_AltInterrupt22	Reserved
23	_Interrupt23	_AltInterrupt23	Reserved
24	_Interrupt24	_AltInterrupt24	Reserved
25	_OC3Interrupt	_AltOC3Interrupt	OC3 Output Compare 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 Output Compare 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
29	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 error interrupt
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 transfer completed interrupt
34	_Interrupt34	_AltInterrupt34	Reserved

TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
35	_Interrupt35	_AltInterrupt35	Reserved
36	_Interrupt36	_AltInterrupt36	Reserved
37	_IC3Interrupt	_AltIC3Interrupt	IC3 Input Capture 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 Input Capture 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 Input Capture 5
40	_Interrupt40	_AltInterrupt40	Reserved
41	_OC5Interrupt	_AltOC5Interrupt	OC5 Output Compare 5
42	_Interrupt42	_AltInterrupt42	Reserved
43	_Interrupt43	_AltInterrupt43	Reserved
44	_Interrupt44	_AltInterrupt44	Reserved
45	_PMPInterrupt	_AltPMPInterrupt	Parallel master port interrupt
46	_Interrupt46	_AltInterrupt46	Reserved
47	_Interrupt47	_AltInterrupt47	Reserved
48	_Interrupt48	_AltInterrupt48	Reserved
49	_SI2C2Interrupt	_AltSI2C2Interrupt	Slave I^2C™ interrupt 2
50	_MI2C2Interrupt	_AltMI2C2Interrupt	Master I^2C interrupt 2
51	_Interrupt51	_AltInterrupt51	Reserved
52	_Interrupt52	_AltInterrupt52	Reserved
53	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
55	_Interrupt55	_AltInterrupt55	Reserved
56	_Interrupt56	_AltInterrupt56	Reserved
57	_Interrupt57	_AltInterrupt57	Reserved
58	_Interrupt58	_AltInterrupt58	Reserved
59	_Interrupt59	_AltInterrupt59	Reserved
60	_Interrupt60	_AltInterrupt60	Reserved
61	_Interrupt61	_AltInterrupt61	Reserved
62	_RTCCInterrupt	_AltRTCCInterrupt	Real-Time Clock And Calender
63	_Interrupt63	_AltInterrupt63	Reserved
64	_Interrupt64	_AltInterrupt64	Reserved
65	_U1ErrInterrupt	_AltU1ErrInterrupt	UART1 error interrupt
66	_U2ErrInterrupt	_AltU2ErrInterrupt	UART2 error interrupt
67	_CRCInterrupt	_AltCRCInterrupt	Cyclic Redundancy Check
68	_Interrupt68	_AltInterrupt68	Reserved
69	_Interrupt69	_AltInterrupt69	Reserved
70	_Interrupt70	_AltInterrupt70	Reserved
71	_Interrupt71	_AltInterrupt71	Reserved
72	_Interrupt72	_AltInterrupt72	Reserved
73	_Interrupt73	_AltInterrupt73	Reserved
74	_Interrupt74	_AltInterrupt74	Reserved
75	_Interrupt75	_AltInterrupt75	Reserved
76	_Interrupt76	_AltInterrupt76	Reserved
77	_Interrupt77	_AltInterrupt77	Reserved
78	_Interrupt78	_AltInterrupt78	Reserved
79	_Interrupt79	_AltInterrupt79	Reserved

TABLE 8-3: INTERRUPT VECTORS - PIC24F MCUs (CONTINUED)

IRQ#	Primary Name	Alternate Name	Vector Function
80	_Interrupt80	_AltInterrupt80	Reserved
81	_Interrupt81	_AltInterrupt81	Reserved
82	_Interrupt82	_AltInterrupt82	Reserved
83	_Interrupt83	_AltInterrupt83	Reserved
84	_Interrupt84	_AltInterrupt84	Reserved
85	_Interrupt85	_AltInterrupt85	Reserved
86	_Interrupt86	_AltInterrupt86	Reserved
87	_Interrupt87	_AltInterrupt87	Reserved
88	_Interrupt88	_AltInterrupt88	Reserved
89	_Interrupt89	_AltInterrupt89	Reserved
90	_Interrupt90	_AltInterrupt90	Reserved
91	_Interrupt91	_AltInterrupt91	Reserved
92	_Interrupt92	_AltInterrupt92	Reserved
93	_Interrupt93	_AltInterrupt93	Reserved
94	_Interrupt94	_AltInterrupt94	Reserved
95	_Interrupt95	_AltInterrupt95	Reserved
96	_Interrupt96	_AltInterrupt96	Reserved
97	_Interrupt97	_AltInterrupt97	Reserved
98	_Interrupt98	_AltInterrupt98	Reserved
99	_Interrupt99	_AltInterrupt99	Reserved
100	_Interrupt100	_AltInterrupt100	Reserved
101	_Interrupt101	_AltInterrupt101	Reserved
102	_Interrupt102	_AltInterrupt102	Reserved
103	_Interrupt103	_AltInterrupt103	Reserved
104	_Interrupt104	_AltInterrupt104	Reserved
105	_Interrupt105	_AltInterrupt105	Reserved
106	_Interrupt106	_AltInterrupt106	Reserved
107	_Interrupt107	_AltInterrupt107	Reserved
108	_Interrupt108	_AltInterrupt108	Reserved
109	_Interrupt109	_AltInterrupt109	Reserved
110	_Interrupt110	_AltInterrupt110	Reserved
111	_Interrupt111	_AltInterrupt111	Reserved
112	_Interrupt112	_AltInterrupt112	Reserved
113	_Interrupt113	_AltInterrupt113	Reserved
114	_Interrupt114	_AltInterrupt114	Reserved
115	_Interrupt115	_AltInterrupt115	Reserved
116	_Interrupt116	_AltInterrupt116	Reserved
117	_Interrupt117	_AltInterrupt117	Reserved

DD

8.4.4 dsPIC33F DSCs/PIC24H MCUs Interrupt Vectors

The table below specifies the interrupt vectors for these 16-bit devices.

TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
N/A	_ReservedTrap0	_AltReservedTrap0	Reserved
N/A	_OscillatorFail	_AltOscillatorFail	Oscillator fail trap
N/A	_AddressError	_AltAddressError	Address error trap
N/A	_StackError	_AltStackError	Stack error trap
N/A	_MathError	_AltMathError	Math error trap
N/A	_DMACError	_AltDMACError	DMA conflict error trap
N/A	_ReservedTrap6	_AltReservedTrap6	Reserved
N/A	_ReservedTrap7	_AltReservedTrap7	Reserved
0	_INT0Interrupt	_AltINT0Interrupt	INT0 External interrupt 0
1	_IC1Interrupt	_AltIC1Interrupt	IC1 Input Capture 1
2	_OC1Interrupt	_AltOC1Interrupt	OC1 Output Compare 1
3	_T1Interrupt	_AltT1Interrupt	TMR1 Timer 1 expired
4	_DMA0Interrupt	_AltDMA0Interrupt	DMA 0 interrupt
5	_IC2Interrupt	_AltIC2Interrupt	IC2 Input Capture 2
6	_OC2Interrupt	_AltOC2Interrupt	OC2 Output Compare 2
7	_T2Interrupt	_AltT2Interrupt	TMR2 Timer 2 expired
8	_T3Interrupt	_AltT3Interrupt	TMR3 Timer 3 expired
9	_SPI1ErrInterrupt	_AltSPI1ErrInterrupt	SPI1 error interrupt
10	_SPI1Interrupt	_AltSPI1Interrupt	SPI1 transfer completed interrupt
11	_U1RXInterrupt	_AltU1RXInterrupt	UART1RX Uart 1 Receiver
12	_U1TXInterrupt	_AltU1TXInterrupt	UART1TX Uart 1 Transmitter
13	_ADC1Interrupt	_AltADC1Interrupt	ADC 1 convert completed
14	_DMA1Interrupt	_AltDMA1Interrupt	DMA 1 interrupt
15	_Interrupt15	_AltInterrupt15	Reserved
16	_SI2C1Interrupt	_AltSI2C1Interrupt	Slave I^2C™ interrupt 1
17	_MI2C1Interrupt	_AltMI2C1Interrupt	Master I^2C interrupt 1
18	_Interrupt18	_AltInterrupt18	Reserved
19	_CNInterrupt	_AltCNInterrupt	CN Input change interrupt
20	_INT1Interrupt	_AltINT1Interrupt	INT1 External interrupt 1
21	_ADC2Interrupt	_AltADC2Interrupt	ADC 2 convert completed
22	_IC7Interrupt	_AltIC7Interrupt	IC7 Input Capture 7
23	_IC8Interrupt	_AltIC8Interrupt	IC8 Input Capture 8
24	_DMA2Interrupt	_AltDMA2Interrupt	DMA 2 interrupt
25	_OC3Interrupt	_AltOC3Interrupt	OC3 Output Compare 3
26	_OC4Interrupt	_AltOC4Interrupt	OC4 Output Compare 4
27	_T4Interrupt	_AltT4Interrupt	TMR4 Timer 4 expired
28	_T5Interrupt	_AltT5Interrupt	TMR5 Timer 5 expired
29	_INT2Interrupt	_AltINT2Interrupt	INT2 External interrupt 2
30	_U2RXInterrupt	_AltU2RXInterrupt	UART2RX Uart 2 Receiver
31	_U2TXInterrupt	_AltU2TXInterrupt	UART2TX Uart 2 Transmitter
32	_SPI2ErrInterrupt	_AltSPI2ErrInterrupt	SPI2 error interrupt
33	_SPI2Interrupt	_AltSPI2Interrupt	SPI2 transfer completed interrupt
34	_C1RxRdyInterrupt	_AltC1RxRdyInterrupt	CAN1 receive data ready

TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
35	_C1Interrupt	_AltC1Interrupt	CAN1 completed interrupt
36	_DMA3Interrupt	_AltDMA3Interrupt	DMA 3 interrupt
37	_IC3Interrupt	_AltIC3Interrupt	IC3 Input Capture 3
38	_IC4Interrupt	_AltIC4Interrupt	IC4 Input Capture 4
39	_IC5Interrupt	_AltIC5Interrupt	IC5 Input Capture 5
40	_IC6Interrupt	_AltIC6Interrupt	IC6 Input Capture 6
41	_OC5Interrupt	_AltOC5Interrupt	OC5 Output Compare 5
42	_OC6Interrupt	_AltOC6Interrupt	OC6 Output Compare 6
43	_OC7Interrupt	_AltOC7Interrupt	OC7 Output Compare 7
44	_OC8Interrupt	_AltOC8Interrupt	OC8 Output Compare 8
45	_Interrupt45	_AltInterrupt45	Reserved
46	_DMA4Interrupt	_AltDMA4Interrupt	DMA 4 interrupt
47	_T6Interrupt	_AltT6Interrupt	TMR6 Timer 6 expired
48	_T7Interrupt	_AltT7Interrupt	TMR7 Timer 7 expired
49	_SI2C2Interrupt	_AltSI2C2Interrupt	Slave I^2C interrupt 2
50	_MI2C2Interrupt	_AltMI2C2Interrupt	Master I^2C interrupt 2
51	_T8Interrupt	_AltT8Interrupt	TMR8 Timer 8 expired
52	_T9Interrupt	_AltT9Interrupt	TMR9 Timer 9 expired
53	_INT3Interrupt	_AltINT3Interrupt	INT3 External interrupt 3
54	_INT4Interrupt	_AltINT4Interrupt	INT4 External interrupt 4
55	_C2RxRdyInterrupt	_AltC2RxRdyInterrupt	CAN2 receive data ready
56	_C2Interrupt	_AltC2Interrupt	CAN2 completed interrupt
57	_PWMInterrupt	_AltPWMInterrupt	PWM period match
58	_QEIInterrupt	_AltQEIInterrupt	QEI position counter compare
59	_DCIErrInterrupt	_AltDCIErrInterrupt	DCI CODEC error interrupt
60	_DCIInterrupt	_AltDCIInterrupt	DCI CODEC transfer done
61	_DMA5Interrupt	_AltDMA5Interrupt	DMA channel 5 interrupt
62	_Interrupt62	_AltInterrupt62	Reserved
63	_FLTAInterrupt	_AltFLTAInterrupt	FLTA MCPWM fault A
64	_FLTBInterrupt	_AltFLTBInterrupt	FLTB MCPWM fault B
65	_U1ErrInterrupt	_AltU1ErrInterrupt	UART1 error interrupt
66	_U2ErrInterrupt	_AltU2ErrInterrupt	UART2 error interrupt
67	_Interrupt67	_AltInterrupt67	Reserved
68	_DMA6Interrupt	_AltDMA6Interrupt	DMA channel 6 interrupt
69	_DMA7Interrupt	_AltDMA7Interrupt	DMA channel 7 interrupt
70	_C1TxReqInterrupt	_AltC1TxReqInterrupt	CAN1 transmit data request
71	_C2TxReqInterrupt	_AltC2TxReqInterrupt	CAN2 transmit data request
72	_Interrupt72	_AltInterrupt72	Reserved
73	_Interrupt73	_AltInterrupt73	Reserved
74	_Interrupt74	_AltInterrupt74	Reserved
75	_Interrupt75	_AltInterrupt75	Reserved
76	_Interrupt76	_AltInterrupt76	Reserved
77	_Interrupt77	_AltInterrupt77	Reserved
78	_Interrupt78	_AltInterrupt78	Reserved
79	_Interrupt79	_AltInterrupt79	Reserved

TABLE 8-4: INTERRUPT VECTORS - dsPIC33F DSCs/PIC24H MCUs

IRQ#	Primary Name	Alternate Name	Vector Function
80	_Interrupt80	_AltInterrupt80	Reserved
81	_Interrupt81	_AltInterrupt81	Reserved
82	_Interrupt82	_AltInterrupt82	Reserved
83	_Interrupt83	_AltInterrupt83	Reserved
84	_Interrupt84	_AltInterrupt84	Reserved
85	_Interrupt85	_AltInterrupt85	Reserved
86	_Interrupt86	_AltInterrupt86	Reserved
87	_Interrupt87	_AltInterrupt87	Reserved
88	_Interrupt88	_AltInterrupt88	Reserved
89	_Interrupt89	_AltInterrupt89	Reserved
90	_Interrupt90	_AltInterrupt90	Reserved
91	_Interrupt91	_AltInterrupt91	Reserved
92	_Interrupt92	_AltInterrupt92	Reserved
93	_Interrupt93	_AltInterrupt93	Reserved
94	_Interrupt94	_AltInterrupt94	Reserved
95	_Interrupt95	_AltInterrupt95	Reserved
96	_Interrupt96	_AltInterrupt96	Reserved
97	_Interrupt97	_AltInterrupt97	Reserved
98	_Interrupt98	_AltInterrupt98	Reserved
99	_Interrupt99	_AltInterrupt99	Reserved
100	_Interrupt100	_AltInterrupt100	Reserved
101	_Interrupt101	_AltInterrupt101	Reserved
102	_Interrupt102	_AltInterrupt102	Reserved
103	_Interrupt103	_AltInterrupt103	Reserved
104	_Interrupt104	_AltInterrupt104	Reserved
105	_Interrupt105	_AltInterrupt105	Reserved
106	_Interrupt106	_AltInterrupt106	Reserved
107	_Interrupt107	_AltInterrupt107	Reserved
108	_Interrupt108	_AltInterrupt108	Reserved
109	_Interrupt109	_AltInterrupt109	Reserved
110	_Interrupt110	_AltInterrupt110	Reserved
111	_Interrupt111	_AltInterrupt111	Reserved
112	_Interrupt112	_AltInterrupt112	Reserved
113	_Interrupt113	_AltInterrupt113	Reserved
114	_Interrupt114	_AltInterrupt114	Reserved
115	_Interrupt115	_AltInterrupt115	Reserved
116	_Interrupt116	_AltInterrupt116	Reserved
117	_Interrupt117	_AltInterrupt117	Reserved

8.5 INTERRUPT SERVICE ROUTINE CONTEXT SAVING

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

To properly handle a return from interrupt, the setup (prologue) code for an ISR function automatically saves the compiler-managed working and special function registers on the stack for later restoration at the end of the ISR. You can use the optional `save` parameter of the `interrupt` attribute to specify additional variables and special function registers to be saved and restored.

In certain applications, it may be necessary to insert assembly statements into the interrupt service routine 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__(__preprologue__("inc _semaphore"))))
  isr0(void);
```

8.6 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 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** – The compiler saves the registers that it uses in the ISR. This includes the working registers and the RCOUNT special function register. Moreover, if the ISR calls an ordinary function, then the compiler will save all the working registers and RCOUNT, even if they are not all used explicitly in the ISR itself. This must be done, because the compiler cannot know, in general, which resources are used by the called function.

8.7 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 (IPC*n*). If there is a pending Interrupt Request (IRQ) with a priority level equal to or greater than the current processor priority level in the Processor Status register (CPUPRI field in the ST register), an interrupt will be presented to the processor.

8.8 ENABLING/DISABLING INTERRUPTS

Each interrupt source can be individually enabled or disabled. One interrupt enable bit for each IRQ is allocated in the Interrupt Enable Control registers (IEC*n*). Setting an interrupt enable bit to one (1) enables the corresponding interrupt; clearing the interrupt enable bit to zero (0) disables the corresponding interrupt. When the device comes out of reset, all interrupt enable bits are cleared to zero. In addition, the processor has a disable interrupt instruction (`DISI`) that can disable all interrupts for a specified number of instruction cycles.

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

The `DISI` instruction can be used in a C program through the use of `__builtin_disi`. For example:

```
__builtin__disi(16);
```

will emit the specified `DISI` instruction at the point it appears in the source program. A disadvantage of using `DISI` in this way is that the C programmer cannot always be sure how the C compiler will translate C source to machine instructions, so it may be difficult to determine the cycle count for the `DISI` instruction. It is possible to get around this difficulty by bracketing the code that is to be protected from interrupts by `DISI` instructions, the first of which sets the cycle count to the maximum value, and the second of which sets the cycle count to zero. For example,

```
__builtin__disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
__builtin__disi(0x0000); /* enable interrupts */
```

An alternative approach is to write directly to the DISICNT register to enable interrupts. The DISICNT register may be modified only after a `DISI` instruction has been issued and if the contents of the DISICNT register are not zero.

```
__builtin__disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

For some applications, it may be necessary to disable level 7 interrupts as well. These can only be disabled through the modification of the `COROCON IPL` field. The provided support files contain some useful preprocessor macro functions to help you safely modify the IPL value. These macros are:

```
SET_CPU_IPL(ipl)
SET_AND_SAVE_CPU_IPL(save_to, ipl)
RESTORE_CPU_IPL(saved_to)
```

For example, you may wish to protect a section of code from interrupt. The following code will adjust the current `IPL` setting and restore the `IPL` to its previous value.

```
void foo(void) {
  int current_cpu_ipl;

  SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7);  /* disable interrupts */
   /* protected code here */
  RESTORE_CPU_IPL(current_cpu_ipl);
}
```

8.9 SHARING MEMORY BETWEEN INTERRUPT SERVICE ROUTINES AND MAINLINE CODE

Care must be taken when modifying the same variable within a main or low-priority Interrupt Service Routine (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 the same variable. Therefore, embedded systems must implement an *atomic* operation to ensure that the intervening high-priority ISR will not write to the same variable from which the low-priority ISR has just read, but has not yet completed its write.

An atomic operation is one that cannot be broken down into its constituent parts - it cannot be interrupted. Depending upon the particular architecture involved, 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, and operations on multi-bit bitfields. Other factors will determine whether or not an atomic operation will be generated, such as memory model settings, optimization level and resource availability.

Consider the general expression:

```
foo = bar op baz;
```

The operator (op) may or may not be atomic, based on device architecture. In any event, the compiler may not be able to generate the atomic operation in all instances - this will very much depend upon several factors:

- the availability of an appropriate atomic machine instruction
- the resource availability - special registers or other constraints
- the 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; they very much depend on the particular application.

8.9.1 Development Issues

Here are some examples:

EXAMPLE 8-1: BAR MUST MATCH BAZ

If 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 8-2: 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 8-3: 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 8-4: 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.

8.9.2 Development Solutions

Here are some strategies to remove potential hazards:

- Design the software system such that the conflicting event cannot occur. Do not share memory between ISRs and other functions. Make ISRs as simple as possible and move the real work to main code.
- Use care when sharing memory and, if possible, avoid sharing bit fields which contain multiple bits.
- Protect non-atomic updates of shared memory from interrupts as you would protect critical sections of code. The following macro can be used for this purpose:

```
#define INTERRUPT_PROTECT(x) {              \
   char saved_ipl;                          \
                                            \
   SET_AND_SAVE_CPU_IPL(saved_ipl,7);       \
   x;                                       \
   RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

  This macro disables interrupts by increasing the current priority level to 7, performing the desired statement and then restoring the previous priority level.

8.9.3 Application Example

The following example highlights some of the points discussed in this section:

```
void __attribute__((interrupt))
     HigherPriorityInterrupt(void) {
      /* User Code Here */
      LATGbits.LATG15 = 1;  /* Set LATG bit 15 */
      IPC0bits.INT0IP = 2;  /* Set Interrupt 0
                               priority (multiple
                               bits involved) to 2 */
       }

int main(void) {
    /* More User Code */
    LATGbits.LATG10 ^= 1;  /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements XOR operation,
                              then writes result to LATG */

    LATG = 0x1238;         /* No problem, this is a write
                              only assignment operation */

    LATGbits.LATG5 = 1;    /* No problem likely,
                              this is an assignment of a
                              single bit and will use a single
                              instruction bit set operation */
```

```
    LATGbits.LATG2 = 0;     /* No problem likely,
                               single instruction bit clear
                               operation probably used */

    LATG += 0x0001;         /* Potential HAZARD -
                               First reads LATG into a W reg,
                               implements add operation,
                               then writes result to LATG */

    IPC0bits.T1IP = 5;      /* HAZARD -
                               Assigning a multiple bitfield
                               can generate a multiple
                               instruction sequence */

}
```

A statement can be protected from interrupt using the INTERRUPT_PROTECT macro provided above. For this example:

```
INTERRUPT_PROTECT(LATGbits.LATG15 ^= 1); /* Not interruptible by
                                            level 1-7 interrupt
                                            requests and safe
                                            at any optimization
                                            level */
```

8.10 PSV USAGE WITH INTERRUPT SERVICE ROUTINES

The introduction of managed psv pointers and CodeGuard Security psv constant sections in compiler v3.0 means that Interrupt Service Routines (ISRs) cannot make any assumptions about the setting of PSVPAG. This is a migration issue for existing applications with ISRs that reference the auto_psv constants section. In previous versions of the compiler, the ISR could assume that the correct value of PSVPAG was set during program startup (unless the programmer had explicitly changed it.)

To help mitigate this problem, two new function attributes will be introduced: auto_psv and no_auto_psv. If an ISR references const variables or string literals using the constants-in-code memory model, the auto_psv attribute should be added to the function definition. This attribute will cause the compiler to preserve the previous contents of PSVPAG and set it to section .const. Upon exit, the previous value of PSVPAG will be restored. For example:

```
void __attribute__((interrupt, auto_psv)) myISR()
{
  /* This function can reference const variables and
  string literals with the constants-in-code memory model. */
}
```

The no_auto_psv attribute is used to indicate that an ISR does not reference the auto_psv constants section. If neither attribute is specified, the compiler will assume auto_psv and will insert the necessary instructions to ensure correct operation at run time. A warning diagnostic message will also be issued. The warning will help alert customers to the migration issue, and to the possibility of reducing interrupt latency by specifying the no_auto_psv attribute.

NOTES:

Chapter 9. Mixing Assembly Language and C Modules

9.1 INTRODUCTION

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

9.2 HIGHLIGHTS

Items discussed in this chapter are:

- **Mixing Assembly Language and C Variables and Functions** – Separate assembly language modules may be assembled, then linked with compiled C modules.
- **Using Inline Assembly Language** – Assembly language instructions may be embedded directly into the C code. The inline assembler supports both simple (non-parameterized) assembly language statement, as well as extended (parameterized) statements, where C variables can be accessed as operands of an assembler instruction.

9.3 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 **Section 4.12 "Register Conventions"**. In particular, registers W0-W7 are used for parameter passing. An assembly language function will receive parameters, and should pass arguments to called functions, in these registers.
- Functions not called during interrupt handling must preserve registers W8-W15. That is, the values in these registers must be saved before they are modified and restored before returning to the calling function. Registers W0-W7 may be used without restoring their values.
- Interrupt functions must preserve all registers. Unlike a normal function call, an interrupt may occur at any point during the execution of a program. When returning to the normal program, all registers must be as they were before the interrupt occurred.
- Variables or functions declared within a separate assembly file that will be referenced by any C source file should be declared as global using the assembler directive `.global`. External symbols should be preceded by at least one underscore. The C function `main` is named `_main` in assembly and conversely an assembly symbol `_do_something` will be referenced in C as `do_something`. Undeclared symbols used in assembly files will be treated as externally defined.

The following example shows how to use variables and functions in both assembly language and C regardless of where they were originally defined.

The file `ex1.c` defines `foo` and `cVariable` to be used in the assembly language file. The C file also shows how to call an assembly function, `asmFunction`, and how to access the assembly defined variable, `asmVariable`.

EXAMPLE 9-1: 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 _begin
_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 9-2: CALLING AN ASSEMBLY FUNCTION IN C

```
/*
** file: call1.c
*/
extern int asmFunction(int, int);
int x;
void
main(void)
{
 x = asmFunction(0x100, 0x200);
}
```

The assembly-language function sums its two parameters and returns the result.

```
;
; file: call2.s
;
  .global _asmFunction
_asmFunction:
  add w0,w1,w0
  return
 .end
```

Parameter passing in C is detailed in **Section 4.11.2 "Return Value"**. In the preceding example, the two integer arguments are passed in the W0 and W1 registers. The integer return result is transferred via register W0. More complicated parameter lists may require different registers and care should be taken in the hand-written assembly to follow the guidelines.

9.4 USING INLINE ASSEMBLY LANGUAGE

Within a C function, the `asm` statement may be used to insert a line of assembly language code into the assembly language that the compiler generates. Inline assembly has two forms: simple and extended.

In the **simple** form, the assembler instruction is written using the syntax:

`asm ("`*`instruction`*`");`

where *`instruction`* is a valid assembly-language construct. If you are writing inline assembly in ANSI C programs, write `__asm__` instead of `asm`.

> **Note:** Only a single string can be passed to the simple form of inline assembly.

In an **extended** assembler instruction using `asm`, the operands of the instruction are specified using C expressions. The extended syntax is:

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
                 [ : [ "constraint"(input-operand) [ , ... ] ]
                      [ "clobber" [ , ... ] ]
               ]
             ]);
```

You must specify an assembler instruction *`template`*, plus an operand *`constraint`* string for each operand. The *`template`* specifies the instruction mnemonic, and optionally placeholders for the operands. The *`constraint`* strings specify operand constraints, for example, that an operand must be in a register (the usual case), or that an operand must be an immediate value.

Constraint letters and modifiers supported by the compiler are listed in Table 9-1 and Table 9-2 respectively.

TABLE 9-1: 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, %*n* represents the first register for the operand (*n*). To access the second, third, or fourth register, use a modifier letter.
T	A near or far data operand.
U	A near data operand.

TABLE 9-2: CONSTRAINT MODIFIERS SUPPORTED BY THE COMPILER

Letter	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 *earlyclobber* 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 *n*, i.e., %d*n*..
q	Fourth register for operand number *n*, i.e., %q*n*..
t	Third register for operand number *n*, i.e., %t*n*..

EXAMPLE 9-3: 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 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 9-4: 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 9-5: 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 9-6: 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) << 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 9-7: 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 9-8: 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 9-9: 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 9-10: MAKING CONTROL FLOW CHANGES

There are special precautions that must be taken when making control flow changes within inline assembly statements.

There is no way, for example, to tell the compiler that an inline asm statement may result in a change of control flow. The control should enter the asm statement and always proceed to the next statement.

Good control flow:

```
asm("call _foo" : /* outputs */
                : /* inputs */
                : "w0", "w1", "w2", "w3", "w4", "w5",
                  "w6", "w7");
/* next statement */
```

This is acceptable because after calling `foo`, the next statement will be executed. The code tells the compiler that some registers do not survive this statement; these represent the registers that will not be preserved by `foo`.

Bad control flow:

```
asm("bra OV, error");
/* next statement */
return 0;

asm("error: ");
return 1;
```

This is unacceptable as the compiler will assume that the next statement, `return 0`, is executed when it may not be. In this case, the `asm("error: ")` and following statements will be deleted because they are unreachable. See further information regarding labels in asm statements.

Acceptable control flow:

```
asm("cp0 _foo\n\t"
    "bra nz, eek\n\t"
    "; some assembly\n\t"
    "bra eek_end\n\t"
    "eek:\n\t"
    "; more assembly\n"
    "eek_end:" : /* outputs */
               : /* inputs */
               : "cc");
/* next statement */
```

This is acceptable, but may not function as expected, (i.e., the next statement is always executed, regardless of any branch inside the asm statement). See further information regarding labels in asm statements. Note that the code indicates that the status flags are no longer valid in this statement by identifying `cc` as clobbered.

Labels and Control Flow:

Additionally, labels inside assembly statements can behave unexpectedly with certain optimization options. The inliner may cause labels within asm statements to be defined multiple times.

Also the procedural aggregator tool (`-mpa`) does not accept the local label syntax. See the following example:

```
inline void foo() {
  asm("do #6, loopend");
  /* some C code */
  asm("loopend: ");
  return;
}
```

This is bad for a number of reasons. First, the asm statements introduce an implied control flow that the compiler does not know about. Second, if `foo()` is inlined, the label `loopend` will be defined many times. Third, the C code could be badly optimized because the compiler cannot see the loop structure. This example breaks the rule that the asm statement should not interfere with the program flow; `asm("loopend:")` will not always flow to the next statement.

A solution would be to use the local label syntax as described in the "*MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide*" (DS51317), as in the following example:

```
inline void foo() {
  asm("do #6, 0f");
  /* some C code */
  asm("0: ");
  return;
}
```

The above form is slightly better; at least it will fix the multiply-defined label issue. However the procedural aggregator tool (`-mpa`) does not accept the `0:` form of label.

EXAMPLE 9-11: USING REQUIRED REGISTERS

Some instructions in the dsPIC DSC instruction set require operands to be in a particular register or groups of registers. Table 9-1 lists some constraint letters that may be appropriate to satisfy the constraints of the instruction that you wish to generate.

If the constraints are not sufficient or you wish to nominate particular registers for use inside asm statements, you may use the register-nominating extensions provided by the compiler to support you (and reduce the need to mark registers as clobbered) as the following code snippet shows. This snippet uses a fictitious instruction that has some odd register requirements:

```
{ register int in1 asm("w7");
  register int in2 asm("w9");
  register int out1 asm("w13");
  register int out2 asm("w0");

  in1 = some_input1;
  in2 = some_input2;
  __asm__ volatile ("funky_instruction %2,%3,%0; = %1" :
                    /* outputs */ "=r"(out1), "=r"(out2) :
                    /* inputs */ "r"(in1), "r"(in2));
  /* use out1 and out2 in normal C */
}
```

In this example, `funky_instruction` has one explicit output, `out1`, and one implicit output, `out2`. Both have been placed in the asm template so that the compiler can track the register usage properly (though the implicit output is in a comment statement). The input shown is normal. Otherwise, the extended register declarator syntax is used to nominate particular hard registers which satisfy the constraints of our fictitious `funky_instruction`.

EXAMPLE 9-12: 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 9-2.

NOTES:

Appendix A. Implementation-Defined Behavior

A.1 INTRODUCTION

This section discusses implementation-defined behavior for the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30). The ISO standard for C requires that vendors document the specifics of "implementation defined" features of the language.

A.2 HIGHLIGHTS

Items discussed in this chapter are:

- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating Point
- Arrays and Pointers
- Registers
- Structures, Unions, Enumerations and Bit fields
- Qualifiers
- Declarators
- Statements
- Preprocessing Directives
- Library Functions
- Signals
- Streams and Files
- tmpfile
- errno
- Memory
- abort
- exit
- getenv
- system
- strerror

A.3 TRANSLATION

Implementation-Defined Behavior for Translation is covered in section G.3.1 of the ANSI C Standard.

Is each non-empty sequence of white-space characters, other than new line, retained or is it replaced by one space character? (ISO 5.1.1.2)

It is replaced by one space character.

How is a diagnostic message identified? (ISO 5.1.1.3)

Diagnostic messages are identified by prefixing them with the source file name and line number corresponding to the message, separated by colon characters (':').

Are there different classes of message? (ISO 5.1.1.3)

Yes.

If yes, what are they? (ISO 5.1.1.3)

Errors, which inhibit production of an output file, and warnings, which do not inhibit production of an output file.

What is the translator return status code for each class of message? (ISO 5.1.1.3)

The return status code for errors is 1; for warnings it is 0.

Can a level of diagnostic be controlled? (ISO 5.1.1.3)

Yes.

If yes, what form does the control take? (ISO 5.1.1.3)

Compiler command line options may be used to request or inhibit the generation of warning messages.

A.4 ENVIRONMENT

Implementation-Defined Behavior for Environment is covered in section G.3.2 of the ANSI C Standard.

What library facilities are available to a freestanding program? (ISO 5.1.2.1)

All of the facilities of the standard C library are available, provided that a small set of functions is customized for the environment, as described in the "Run Time Libraries" section.

Describe program termination in a freestanding environment. (ISO 5.1.2.1)

If the function main returns or the function exit is called, a HALT instruction is executed in an infinite loop. This behavior is customizable.

Describe the arguments (parameters) passed to the function `main`? (ISO 5.1.2.2.1)

No parameters are passed to `main`.

Which of the following is a valid interactive device: (ISO 5.1.2.3)

Asynchronous terminal No

Paired display and keyboard No

Inter program connection No

Other, please describe? None

A.5 IDENTIFIERS

Implementation-Defined Behavior for Identifiers is covered in section G.3.3 of the ANSI C Standard.

How many characters beyond thirty-one (31) are significant in an identifier without external linkage? (ISO 6.1.2)

All characters are significant.

How many characters beyond six (6) are significant in an identifier with external linkage? (ISO 6.1.2)

All characters are significant.

Is case significant in an identifier with external linkage? (ISO 6.1.2)

Yes.

A.6 CHARACTERS

Implementation-Defined Behavior for Characters is covered in section G.3.4 of the ANSI C Standard.

Detail any source and execution characters which are not explicitly specified by the Standard? (ISO 5.2.1)

None.

List escape sequence value produced for listed sequences. (ISO 5.2.2)

TABLE A-1: ESCAPE SEQUENCE CHARACTERS AND VALUES

Sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

How many bits are in a character in the execution character set? (ISO 5.2.4.2)

8.

What is the mapping of members of the source character sets (in character and string literals) to members of the execution character set? (ISO 6.1.3.4)

The identity function.

What is the equivalent type of a plain `char`? (ISO 6.2.1.1)

Either (user defined). The default is `signed char`. A compiler command-line option may be used to make the default `unsigned char`.

A.7 INTEGERS

Implementation-Defined Behavior for Integers is covered in section G.3.5 of the ANSI C Standard.

The following table describes the amount of storage and range of various types of integers: (ISO 6.1.2.5)

TABLE A-2: INTEGER TYPES

Designation	Size (bits)	Range
`char`	8	-128 ... 127
`signed char`	8	-128 ... 127
`unsigned char`	8	0 ... 255
`short`	16	-32768 ... 32767
`signed short`	16	-32768 ... 32767
`unsigned short`	16	0 ... 65535
`int`	16	-32768 ... 32767
`signed int`	16	-32768 ... 32767
`unsigned int`	16	0 ... 65535
`long`	32	-2147483648 ... 2147438647
`signed long`	32	-2147483648 ... 2147438647
`unsigned long`	32	0 ... 4294867295

What is the result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented? (ISO 6.2.1.2)

There is a loss of significance. No error is signaled.

What are the results of bitwise operations on signed integers? (ISO 6.3)

Shift operators retain the sign. Other operators act as if the operand(s) are unsigned integers.

What is the sign of the remainder on integer division? (ISO 6.3.5)

+

What is the result of a right shift of a negative-valued signed integral type? (ISO 6.3.7)

The sign is retained.

A.8 FLOATING POINT

Implementation-Defined Behavior for Floating Point is covered in section G.3.6 of the ANSI C Standard.

Is the scaled value of a floating constant that is in the range of the representable value for its type, the nearest representable value, or the larger representable value immediately adjacent to the nearest representable value, or the smallest representable value immediately adjacent to the nearest representable value? (ISO 6.1.3.1)

The nearest representable value.

The following table describes the amount of storage and range of various types of floating point numbers: (ISO 6.1.2.5)

TABLE A-3: FLOATING-POINT TYPES

Designation	Size (bits)	Range
`float`	32	1.175494e-38 … 3.40282346e+38
`double*`	32	1.175494e-38 … 3.40282346e+38
`long double`	64	2.22507385e-308 … 1.79769313e+308

* `double` is equivalent to `long double` if `-fno-short-double` is used.

What is the direction of truncation, when an integral number is converted to a floating-point number, that cannot exactly represent the original value? (ISO 6.2.1.3)

Down.

What is the direction of truncation, or rounding, when a floating-point number is converted to a narrower floating-point number? (ISO 6.2.1.4)

Down.

A.9 ARRAYS AND POINTERS

Implementation-Defined Behavior for Arrays and Pointers is covered in section G.3.7 of the ANSI C Standard.

What is the type of the integer required to hold the maximum size of an array that is, the type of the size of operator, `size_t`? (ISO 6.3.3.4, ISO 7.1.1)

`unsigned int`.

What is the size of integer required for a pointer to be converted to an integral type? (ISO 6.3.4)

16 bits.

What is the result of casting a pointer to an integer, or vice versa? (ISO 6.3.4)

The mapping is the identity function.

What is the type of the integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t`? (ISO 6.3.6, ISO 7.1.1)

`unsigned int`.

A.10 REGISTERS

Implementation-Defined Behavior for Registers is covered in section G.3.8 of the ANSI C Standard.

To what extent does the storage class specifier `register` actually effect the storage of objects in registers? (ISO 6.5.1)

If optimization is disabled, an attempt will be made to honor the `register` storage class; otherwise, it is ignored.

A.11 STRUCTURES, UNIONS, ENUMERATIONS AND BIT FIELDS

Implementation-Defined Behavior for Structures, Unions, Enumerations and Bit Fields is covered in sections A.6.3.9 and G.3.9 of the ANSI C Standard.

What are the results if a member of a union object is accessed using a member of a different type? (ISO 6.3.2.3)

No conversions are applied.

Describe the padding and alignment of members of structures? (ISO 6.5.2.1)

Chars are byte-aligned. All other objects are word-aligned.

What is the equivalent type for a plain `int` bit field*? (ISO 6.5.2.1)*

User defined. By default, `signed int` bit field. May be made an `unsigned int` bit field using a command line option.

What is the order of allocation of bit fields within an int? (ISO 6.5.2.1)

Bits are allocated from least-significant to most-significant.

Can a bit field straddle a storage-unit boundary? (ISO 6.5.2.1)

Yes.

Which integer type has been chosen to represent the values of an enumeration type? (ISO 6.5.2.2)

`int`.

A.12 QUALIFIERS

Implementation-Defined Behavior for Qualifiers is covered in section G.3.10 of the ANSI C Standard.

Describe what action constitutes an access to an object that has volatile-qualified type? (ISO 6.5.3)

If an object is named in an expression, it has been accessed.

A.13 DECLARATORS

Implementation-Defined Behavior for Declarators is covered in section G.3.11 of the ANSI C Standard.

What is the maximum number of declarators that may modify an arithmetic, structure, or union type? (ISO 6.5.4)

No limit.

A.14 STATEMENTS

Implementation-Defined Behavior for Statements is covered in section G.3.12 of the ANSI C Standard.

What is the maximum number of case values in a switch statement? (ISO 6.6.4.2)

No limit.

A.15 PREPROCESSING DIRECTIVES

Implementation-Defined Behavior for Preprocessing Directives is covered in section G.3.13 of the ANSI C Standard.

Does the value of a single-character character constant in a constant expression, that controls conditional inclusion, match the value of the same character constant in the execution character set? (ISO 6.8.1)

Yes.

Can such a character constant have a negative value? (ISO 6.8.1)

Yes.

What method is used for locating includable source files? (ISO 6.8.2)

The preprocessor searches the current directory, followed by directories named using command-line options.

How are headers identified, or the places specified? (ISO 6.8.2)

The headers are identified on the `#include` directive, enclosed between < and > delimiters, or between " and " delimiters. The places are specified using command-line options.

Are quoted names supported for includable source files? (ISO 6.8.2)

Yes.

What is the mapping between delimited character sequences and external source file names? (ISO 6.8.2)

The identity function.

Describe the behavior of each recognized #pragma directive. (ISO 6.8.6)

TABLE A-4: #PRAGMA BEHAVIOR

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

A.16 LIBRARY FUNCTIONS

Implementation-Defined Behavior for Library Functions is covered in section G.3.14 of the ANSI C Standard.

What is the null pointer constant to which the macro NULL expands? (ISO 7.1.5)

0.

How is the diagnostic printed by the assert function recognized, and what is the termination behavior of this function? (ISO 7.2)

The assert function prints the file name, line number and test expression, separated by the colon character (':'). It then calls the abort function.

What characters are tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions? (ISO 7.3.1)

TABLE A-5: CHARACTERS TESTED BY IS FUNCTIONS

Function	Characters tested
`isalnum`	One of the letters or digits: `isalpha` or `isdigit`.
`isalpha`	One of the letters: `islower` or `isupper`.
`iscntrl`	One of the five standard motion control characters, backspace and alert: `\f`, `\n`, `\r`, `\t`, `\v`, `\b`, `\a`.
`islower`	One of the letters 'a' through 'z'.
`isprint`	A graphic character or the space character: `isalnum` or `ispunct` or `space`.
`isupper`	One of the letters 'A' through 'Z'.
`ispunct`	One of the characters: ! " # % & ' ( ) ; < = > ? [\ ] * + , - . / : ^

What values are returned by the mathematics functions after a domain errors? (ISO 7.5.1)

NaN.

Do the mathematics functions set the integer expression `errno` to the value of the macro ERANGE on underflow range errors? (ISO 7.5.1)

Yes.

Do you get a domain error or is zero returned when the `fmod` function has a second argument of zero? (ISO 7.5.6.4)

Domain error.

A.17 SIGNALS

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

TABLE A-6: SIGNAL FUNCTION

Name	Description
SIGABRT	Abnormal termination.
SIGINT	Receipt of an interactive attention signal.
SIGILL	Detection of an invalid function image.
SIGFPE	An erroneous arithmetic operation.
SIGSEGV	An invalid access to storage.
SIGTERM	A termination request sent to the program.

Describe the parameters and the usage of each signal recognized by the signal function. (ISO 7.7.1.1)

Application defined.

Describe the default handling and the handling at program startup for each signal recognized by the signal function? (ISO 7.7.1.1)

None.

If the equivalent of signal (sig,SIG_DFL) is not executed prior to the call of a signal handler, what blocking of the signal is performed? (ISO 7.7.1.1)

None.

Is the default handling reset if a SIGILL signal is received by a handler specified to the signal function? (ISO 7.7.1.1)

No.

A.18 STREAMS AND FILES

Does the last line of a text stream require a terminating new line character? (ISO 7.9.2)

No.

Do space characters, that are written out to a text stream immediately before a new line character, appear when the stream is read back in? (ISO 7.9.2)

Yes.

How many null characters may be appended to data written to a binary stream? (ISO 7.9.2)

None.

Is the file position indicator of an append mode stream initially positioned at the start or end of the file? (ISO 7.9.3)

Start.

Does a write on a text stream cause the associated file to be truncated beyond that point? (ISO 7.9.3)

Application defined.

Describe the characteristics of file buffering. (ISO 7.9.3)

Fully buffered.

Can zero-length file actually exist? (ISO 7.9.3)

Yes.

What are the rules for composing a valid file name? (ISO 7.9.3)

Application defined.

Can the same file be open multiple times? (ISO 7.9.3)

Application defined.

What is the effect of the remove function on an open file? (ISO 7.9.4.1)

Application defined.

What is the effect if a file with the new name exists prior to a call to the rename function? (ISO 7.9.4.2)

Application defined.

What is the form of the output for %p conversion in the `fprintf` function? (ISO 7.9.6.1)

A hexadecimal representation.

What form does the input for %p conversion in the `fscanf` function take? (ISO 7.9.6.2)

A hexadecimal representation.

A.19 TMPFILE

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

Yes.

A.20 ERRNO

What value is the macro `errno` set to by the `fgetpos` or `ftell` function on failure? (ISO 7.9.9.1, (ISO 7.9.9.4)

Application defined.

What is the format of the messages generated by the `perror` function? (ISO 7.9.10.4)

The argument to `perror`, followed by a colon, followed by a text description of the value of `errno`.

A.21 MEMORY

What is the behavior of the `calloc`, `malloc` or `realloc` function if the size requested is zero? (ISO 7.10.3)

A block of zero length is allocated.

A.22 ABORT

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

Nothing.

A.23 EXIT

What is the status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE? (ISO 7.10.4.3)

The value of the argument.

A.24 GETENV

What limitations are there on environment names? (ISO 7.10.4.4)

Application defined.

Describe the method used to alter the environment list obtained by a call to the `getenv` function. (ISO 7.10.4.4)

Application defined.

A.25 SYSTEM

Describe the format of the string that is passed to the system function. (ISO 7.10.4.5)

Application defined.

What mode of execution is performed by the system function? (ISO 7.10.4.5)

Application defined.

A.26 STRERROR

Describe the format of the error message output by the `strerror` function. (ISO 7.11.6.2)

A plain character string.

List the contents of the error message strings returned by a call to the `strerror` function. (ISO 7.11.6.2)

TABLE A-7: ERROR MESSAGE STRINGS

Errno	Message
0	No error
EDOM	Domain error
ERANGE	Range error
EFPOS	File positioning error
EFOPEN	File open error
nnn	Error #nnn

NOTES:

Appendix B. Built-in Functions

B.1 INTRODUCTION

This appendix describes the built-in functions that are specific to MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30).

Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using inline assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and do not involve function calls or library routines.

There are a number of reasons why providing built-in functions is preferable to requiring programmers to use inline assembly. They include the following:

1. Providing built-in functions for specific purposes simplifies coding.
2. Certain optimizations are disabled when inline assembly is used. This is not the case for built-in functions.
3. For machine instructions that use dedicated registers, coding inline assembly while avoiding register allocation errors can require considerable care. The built-in functions make this process simpler as you do not need to be concerned with the particular register requirements for each individual machine instruction.

This chapter is organized as follows:

Built-In Function List

__builtin_addab
__builtin_add
__builtin_btg
__builtin_clr
__builtin_clr_prefetch
__builtin_divf
__builtin_divmodsd
__builtin_divmodud
__builtin_divsd
__builtin_divud
__builtin_dmaoffset
__builtin_ed
__builtin_edac
__builtin_fbcl
__builtin_lac
__builtin_mac
__builtin_modsd
__builtin_modud
__builtin_movsac
__builtin_mpy
__builtin_mpyn
__builtin_msc
__builtin_mulss
__builtin_mulsu
__builtin_mulus
__builtin_muluu
__builtin_nop
__builtin_psvpage
__builtin_psvoffset
__builtin_readsfr
__builtin_return_address
__builtin_sac
__builtin_sacr
__builtin_sftac
__builtin_subab
__builtin_tbladdress
__builtin_tblpage
__builtin_tbloffset
__builtin_tblrdh
__builtin_tblrdl
__builtin_tblwth
__builtin_tblwtl
__builtin_write_NVM
__builtin_write_PWMSFR
__builtin_write_RTCWEN
__builtin_write_OSCCONL
__builtin_write_OSCCONH

B.2 BUILT-IN FUNCTION LIST

This section describes the programmer interface to the compiler built-in functions. Since the functions are "built in", there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler's namespace (they all have the prefix `__builtin_`), so they will not conflict with function or variable names in the programmer's namespace.

__builtin_addab

Description: Add accumulators A and B with the result written back to the specified accumulator. For example:

```
register int result asm("A");
register int B asm("A");

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.

__builtin_add

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

```
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

__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 0x`*n*`);`

Argument:
- `*` A pointer to the data item for which a bit should be toggled.
- 0x*n* A literal value in the range of 0 to 15.

Return Value: Returns a btg machine instruction.

Assembler Operator / Machine Instruction: `btg`

Error Messages An error message will be displayed if the parameter values are not within range

__builtin_clr

Description: Clear the specified accumulator. For example:

```
  register int result asm("A");
  result = __builtin_clr();
```

will generate:

```
  clr A
```

Prototype: `int __builtin_clr(void);`

Argument: None

Return Value: Returns the cleared value result to an accumulator.

__builtin_clr

Assembler Operator / Machine Instruction:	`clr`
Error Messages	An error message will be displayed if the result is not an accumulator register.

__builtin_clr_prefetch

Description:

Clear an accumulator and prefetch data ready for a future MAC operation.
`xptr` may be null to signify no X prefetch to be performed, in which case the values of *`xincr`* and *`xval`* are ignored, but required.
`yptr` may be null to signify no Y prefetch to be performed, in which case the values of *`yincr`* and *`yval`* are ignored, but required.
`xval` and *`yval`* nominate the address of a C variable where the prefetched value will be stored.
`xincr` and *`yincr`* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.
If *`AWB`* is non null, the other accumulator will be written back into the referenced variable.
For example:

```
register int result asm("A");
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);
```

__builtin_clr_prefetch

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	

__builtin_divf

Description:	Computes the quotient *num* / *den*. A math error exception occurs if *den* is zero. Function arguments are unsigned, as is the function result.
Prototype:	`unsigned int __builtin_divf(unsigned int num, 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.f`

__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*" (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.	
Prototype:	`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.	

__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*" (DS70157). Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.
Prototype:	`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.

__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`

__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`

__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.

__builtin_ed

Description: Squares `sqr`, 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:

```
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 **xptr, int xincr,
  int **yptr, 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`

__builtin_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

__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:

```
register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(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

__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.

__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:

```
register int result asm("A");
int value;
result = __builtin_lac(value,3);
```

Might generate:

```
lac w4, #3, A
```

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 addition 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

__builtin_mac

Description: Computes *a* x *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:

```
register int result asm("A");
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 cleared value result to an accumulator.

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

__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.

__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.

__builtin_movsac

Description: Computes nothing, but prefetches data ready for a future MAC operation.
xptr may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required.
yptr may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required.
xval and *yval* nominate the address of a C variable where the prefetched value will be stored.
xincr and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.
If *AWB* is non null, the other accumulator will be written back into the referenced variable.
For example:

```
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_movsac(&xmemory, &xVal, 2,
                    &ymemory, &yVal, 2, 0, 0);
```

might generate:

```
        movsac A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```
int __builtin_movsac(
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB
  int AWB_accum);
```

Argument:

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 prefetch data.

Assembler Operator / Machine Instruction: `movsac`

Error Messages An error message will be displayed if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
- *AWB_accum* is not an accumulator register and *AWB* is not null

__builtin_mpy

Description:

Computes *a* x *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:

```
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 cleared value result to an accumulator.

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

__builtin_mpyn

Description: Computes -*a* x *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:

```
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_mpyn(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 cleared value result to an accumulator.

Assembler Operator / Machine Instruction: mpyn

Error Messages An error message will be displayed if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null

__builtin_msc

Description:

Computes *a* x *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:

```
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	IAccumulator 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 cleared value result to an accumulator.

Assembler Operator / Machine Instruction:

`msc`

Error Messages

An error message will be displayed if:

- the result is not an accumulator register
- *Accum* is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
- *AWB_accum* is not an accumulator register and *AWB* is not null

__builtin_mulss

Description: Computes the product *p0* x *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:

```
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* x *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`

__builtin_mulsu

Description: Computes the product *p0* x *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:

```
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* x *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`

__builtin_mulus

Description: Computes the product *p0* x *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:

```
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* x *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`

__builtin_muluu

Description: Computes the product *p0* x *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:

```
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* x *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`

__builtin_nop

Description: Generates a `nop` instruction.

Prototype: `void __builtin_nop(void);`

__builtin_nop

Argument:	None.
Return Value:	Returns a no operation (nop).
Assembler Operator / Machine Instruction:	nop

__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 **Section 2.3.1 "Specifying Attributes of Variables"**.
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);`

__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 **Section 2.3.1 "Specifying Attributes of Variables"**.
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);`

__builtin_readsfr

Description:	Reads the SFR.
Prototype:	`unsigned int __builtin_readsfr(const void *p);`
Argument:	*p* object address
Return Value:	Returns the SFR.
Assembler Operator / Machine Instruction:	`readsfr`
Error Messages	The following error message is produced when this function is used incorrectly:

__builtin_return_address

Description:	Returns the return address of the current function, or of one of its callers. For the *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`

__builtin_sac

Description:	Shifts value by *shift* (a literal between -8 and 7) and returns the value.
	For example:
	`register int value asm("A");`
	`int result;`
	
	`result = __builtin_sac(value,3);`
	Might generate:
	`sac A, #3, w0`

Prototype:	`int __builtin_sac(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:	`sac`
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

__builtin_sacr

Description: Shifts value by *shift* (a literal between -8 and 7) and returns the value which is rounded using the rounding mode determined by the CORCONbits.RND control bit.
For example:

```
register int value asm("A");
int result;

result = __builtin_sac(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.

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

__builtin_sftac

Description: Shifts accumulator by *shift*. The valid shift range is -16 to 16.
For example:

```
register int result asm("A");
int i;

result = __builtin_sftac(result,i);
```

Might generate:

```
sftac A, w0
```

Prototype: `int __builtin_sftac(int Accum, int shift);`

Argument:
Accum *Accumulator to shift.*
shift Amount to shift.

Return Value: Returns the shifted result to an accumulator.

Assembler Operator / Machine Instruction: `sftac`

Error Messages An error message will be displayed if:
- the result is not an accumulator register
- *Accum* is not an accumulator register
- the shift value is not a literal within range

__builtin_subab

Description:	Subtracts accumulators A and B with the result written back to the specified accumulator. For example:
	`register int result asm("A");`
	`register int B asm("B");`
	`result = __builtin_subab(result,B);`
	will generate:
	`sub A`

Prototype:	`int __builtin_subab(int Accum_a, int Accum_b);`
Argument:	*Accum_a* *Accumulator from which to subtract.* *Accum_b* Accumulator to subtract.
Return Value:	Returns the subtraction result to an accumulator.
Assembler Operator / Machine Instruction:	`sub`
Error Messages	An error message will be displayed if the result is not an accumulator register.

__builtin_tbladdress

Description:	Returns a value that represents the address of an object in program memory. The argument *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 **Section 2.3.1 "Specifying Attributes of Variables"**.
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);`

__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 **Section 2.3.1 "Specifying Attributes of Variables"**.
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.

__builtin_tblpage

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);`

__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 **Section 2.3.1 "Specifying Attributes of Variables"**.
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);`

__builtin_tblrdh

Description:	Issues the `tblrdh.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 the data sheet or *dsPIC Family Reference Manual* for complete details regarding reading and writing program Flash.
Prototype:	`unsigned int __builtin_tblrdh(unsigned int offset);`
Argument:	*`offset`* desired memory offset
Return Value:	None.
Assembler Operator / Machine Instruction:	`tblrdh`
Error Messages	None.

__builtin_tblrdl

Description:	Issues the `tblrdl.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 the data sheet or "*dsPIC30F Family Reference Manual*" (DS70046) for complete details regarding reading and writing program Flash.
Prototype:	`unsigned int __builtin_tblrdl(unsigned int offset);`
Argument:	*offset* desired memory offset
Return Value:	None.
Assembler Operator / Machine Instruction:	`tblrdl`
Error Messages	None.

__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 the data sheet or "*dsPIC30F Family Reference Manual*" (DS70046) 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.

__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 the data sheet or "*dsPIC30F Family Reference Manual*" (DS70046) 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`

__builtin_tblwtl

Error Messages None.

__builtin_write_NVM

Description: Enables the Flash for writing by issuing the correct unlock sequence and enabling the Write bit of the NVMCON register.
Interrupts may need to be disable for proper operation.
This builtin function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.

Prototype:
```
void __builtin_write_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.

__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 #<it>PWM_KEY</it>, w3
mov #<it>value</it>, w2
mov #0x4321, w1
mov #0xABCD, w0
mov w1,[w3]
mov w0,[w3]
mov w2,[w3]
```

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.

__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 RCFGCAL SFR. Interrupts may need to be disable for proper operation. This builtin function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	`void __builtin_write_RTCWEN(void);`
Argument:	None.
Return Value:	None.

Assembler Operator / Machine Instruction:	`mov #0x55, Wn`
	`mov Wn, _NVMKEY`
	`mov #0xAA, Wn`
	`mov Wn, _NVMKEY`
	`bset _RCFGCAL, #13`
	`nop`
	`nop`

Error Messages	None.

__builtin_write_OSCCONL

Description:	Unlocks and writes its argument to OSCCONL. Interrupts may need to be disable for proper operation. This builtin function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	`void __builtin_write_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.

__builtin_write_OSCCONH

Description:	Unlocks and writes its argument to OSCCONH. Interrupts may need to be disable for proper operation. This builtin function can be used as a part of a complex sequence discussed in the data sheet or family reference manual. See this documentation for more information.
Prototype:	`void __builtin_write_OSCCONH(unsigned char value);`
Argument:	*value* character to be written
Return Value:	None.

Assembler Operator / 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.

Appendix C. MPLAB C Compiler for PIC18 MCUs vs. 16-Bit Devices

C.1 INTRODUCTION

The purpose of this chapter is to highlight the differences between the MPLAB C Compiler for PIC18 MCUs (formerly MPLAB C18) and the MPLAB C C Compiler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB C30). For more details on the PIC18 MCU compiler, please refer to the "*MPLAB® C18 C Compiler User's Guide*" (DS51288).

C.2 HIGHLIGHTS

This chapter discusses the following areas of difference between the two compilers:

- Data Formats
- Pointers
- Storage Classes
- Stack Usage
- Storage Qualifiers
- Predefined Macro Names
- Integer Promotions
- String Constants
- Access Memory
- Inline Assembly
- Pragmas
- Memory Models
- Calling Conventions
- Startup Code
- Compiler-Managed Resources
- Optimizations
- Object Module Format
- Implementation-Defined Behavior
- Bit fields

C.3 DATA FORMATS

TABLE C-1: NUMBER OF BITS USED IN DATA FORMATS

Data Format	MPLAB® C Compiler for	
	PIC18 MCUs(1)	16-Bit Devices(2)
`char`	8	8
`int`	16	16
`short long`	24	-
`long`	32	32
`long long`	-	64
`float`	32	32
`double`	32	32 or 64(3)

Note 1: The PIC18 MCU Compiler uses its own data format, which is similar to IEEE-754 format, but with the top nine bits rotated (see Table C-2).

2: The 16-Bit Device Compiler uses IEEE-754 format.

3: See **Section 5.5 "Floating Point"**.

TABLE C-2: FLOATING-POINT VS. IEEE-754 FORMAT

Standard	Byte 3	Byte 2	Byte 1	Byte 0
PIC18 MCU Compiler	eeeeeeee$_0$	sddd dddd$_{16}$	dddd dddd$_8$	dddd dddd$_0$
16-Bit Device Compiler	seeeeeee$_1$	e$_0$ddd dddd$_{16}$	dddd dddd$_8$	dddd dddd$_0$

Legend: s = sign bit, d = mantissa, e = exponent

C.4 POINTERS

TABLE C-3: NUMBER OF BITS USED FOR POINTERS

Memory Type	MPLAB® C Compiler for	
	PIC18 MCUs	16-Bit Devices
Program Memory - Near	16	16
Program Memory - Far	24	16
Data Memory	16	16

C.5 STORAGE CLASSES

The PIC18 MCU Compiler allows the non-ANSI storage class specifiers `overlay` for variables and `auto` or `static` for function arguments.

The 16-Bit Device Compiler does not allow these specifiers.

C.6 STACK USAGE

TABLE C-4: TYPE OF STACK USED

Items on Stack	MPLAB® C Compiler for	
	PIC18 MCUs	16-Bit Devices
Return Addresses	hardware	software
Local Variables	software	software

C.7 STORAGE QUALIFIERS

The PIC18 MCU Compiler uses the non-ANSI `far`, `near`, `rom` and `ram` type qualifiers.

The 16-Bit Device Compiler uses the non-ANSI `far`, `near` and `space` attributes.

EXAMPLE C-1: DEFINING A NEAR VARIABLE

PIC18	`near int gVariable;`
16-Bit	`__attribute__((near)) int gVariable;`

EXAMPLE C-2: DEFINING A FAR VARIABLE

PIC18	`far int gVariable;`
16-Bit	`__attribute__((far)) int gVariable;`

EXAMPLE C-3: CREATING A VARIABLE IN PROGRAM MEMORY

PIC18	`rom int gArray[6] = {0,1,2,3,4,5};`
16-Bit	`__attribute__((space(psv)))` `  const int gArray[6] = {0,1,2,3,4,5};`

C.8 PREDEFINED MACRO NAMES

The PIC18 MCU Compiler defines `__18CXX`, `__18F242`, ... (all other processors with `__` prefix) and `__SMALL__` or `__LARGE__`, depending on the selected memory model.

The 16-Bit Device Compiler defines `__dsPIC30`.

C.9 INTEGER PROMOTIONS

The PIC18 MCU Compiler performs integer promotions at the size of the largest operand even if both operands are smaller than an `int`. This compiler provides the `-Oi+` option to conform to the standard.

The 16-Bit Device Compiler performs integer promotions at `int` precision or greater as mandated by ISO.

C.10 STRING CONSTANTS

The PIC18 MCU Compiler keeps string constants in program memory in its `.stringtable` section. This compiler supports several variants of the string functions. For instance, the `strcpy` function has four variants allowing the copying of a string to and from both data and program memory.

The 16-Bit Device Compiler accesses string constants from data memory or from program memory through the PSV window, allowing constants to be accessed like any other data.

C.11 ACCESS MEMORY

16-bit devices do not have access memory.

C.12 INLINE ASSEMBLY

The PIC18 MCU Compiler uses non-ANSI `_asm` and `_endasm` to identify a block of inline assembly.

The 16-Bit Device Compiler uses non-ANSI `asm`, which looks more like a function call. The compiler use of the `asm` statement is detailed in **Section 9.4 “Using Inline Assembly Language”**.

C.13 PRAGMAS

The PIC18 MCU Compiler uses pragmas for sections (`code`, `romdata`, `udata`, `idata`), interrupts (high-priority and low-priority) and variable locations (bank, section).

The 16-Bit Device Compiler uses non-ANSI attributes instead of pragmas.

TABLE C-5: PRAGMAS VS. ATTRIBUTES

Pragma (PIC18 MCU Compiler)	Attribute (16-Bit Device Compiler)
`#pragma udata [name]`	`__attribute__ ((section ("name")))`
`#pragma idata [name]`	`__attribute__ ((section ("name")))`
`#pragma romdata [name]`	`__attribute__ ((space (prog)))`
`#pragma code [name]`	`__attribute__ ((section ("name"))),` `__attribute__ ((space (prog)))`
`#pragma interruptlow`	`__attribute__ ((interrupt))`
`#pragma interrupt`	`__attribute__ ((interrupt, shadow))`
`#pragma varlocate bank`	NA*
`#pragma varlocate name`	NA*

*16-bit devices do not have banks.

EXAMPLE C-4: SPECIFY AN UNINITIALIZED VARIABLE IN A USER SECTION IN DATA MEMORY

PIC18	`#pragma udata mybss` `int gi;`
16-Bit	`int __attribute__((__section__(".mybss"))) gi;`

EXAMPLE C-5: LOCATE THE VARIABLE MABONGA AT ADDRESS 0X100 IN DATA MEMORY

PIC18	`#pragma idata myDataSection=0x100;` `int Mabonga = 1;`
16-Bit	`int __attribute__((address(0x100))) Mabonga = 1;`

EXAMPLE C-6: SPECIFY A VARIABLE TO BE PLACED IN PROGRAM MEMORY

PIC18	`#pragma romdata const_table`
	`const rom char my_const_array[10] =`
	`{0,1,2,3,4,5,6,7,8,9};`

16-Bit	`const __attribute__((space(auto_psv)))` `char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};`

Note: The 16-Bit Device Compiler does not directly support accessing variables in program space. Variables so allocated must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window. See **Section 4.14 "Program Space Visibility (PSV) Usage"** for more on the PSV window.

EXAMPLE C-7: LOCATE THE FUNCTION `PRINTSTRING` AT ADDRESS `0X8000` IN PROGRAM MEMORY

PIC18	`#pragma code myTextSection=0x8000;` `int PrintString(const char *s){...};`
16-Bit	`int __attribute__((address(0x8000))) PrintString` `  (const char *s) {...};`

EXAMPLE C-8: COMPILER AUTOMATICALLY SAVES AND RESTORES THE VARIABLES VAR1 AND VAR2

PIC18	`#pragma interrupt isr0 save=var1, var2`
	`  void isr0(void)`
	`  {`
	`  /* perform interrupt function here */`
	`  }`

16-Bit	`void __attribute__((__interrupt__(__save__(var1,var2))))`
	`  isr0(void)`
	`  {`
	`  /* perform interrupt function here */`
	`  }`

C.14 MEMORY MODELS

The PIC18 MCU Compiler uses non-ANSI small and large memory models. Small uses the 16-bit pointers and restricts program memory to be less than 64 KB (32 KB words).

The 16-Bit Device Compiler uses non-ANSI small code and large code models. Small code restricts program memory to be less than 96 KB (32 KB words). In large code, pointers may go through a jump table.

C.15 CALLING CONVENTIONS

There are many differences in the calling conventions of the MPLAB C Compiler for PIC18 MCUs and the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs. Please refer to **Section 4.11 "Function Call Conventions"** for a discussion of 16-Bit Device Compiler calling conventions.

C.16 STARTUP CODE

The PIC18 MCU Compiler provides three startup routines – one that performs no user data initialization, one that initializes only variables that have initializers, and one that initializes all variables (variables without initializers are set to zero as required by the ANSI standard).

The 16-Bit Device Compiler provides two startup routines – one that performs no user data initialization and one that initializes all variables (variables without initializers are set to zero as required by the ANSI standard) except for variables in the persistent data section.

C.17 COMPILER-MANAGED RESOURCES

The PIC18 MCU Compiler has the following managed resources: PC, WREG, STATUS, PROD, section .tmpdata, section MATH_DATA, FSR0, FSR1, FSR2, TBLPTR, TABLAT.

The 16-Bit Device Compiler has the following managed resources: W0-W15, RCOUNT, SR.

C.18 OPTIMIZATIONS

The following optimizations are part of each compiler.

MPLAB® C Compiler for	
PIC18 MCUs	**16-Bit Devices**
Branches(-Ob+) Code Straightening(-Os+) Tail Merging(-Ot+) Unreachable Code Removal(-Ou+) Copy Propagation(-Op+) Redundant Store Removal(-Or+) Dead Code Removal(-Od+)	Optimization settings (-On where n is 1, 2, 3 or s)(1)
Duplicate String Merging (-Om+)	-fwritable-strings
Banking (-On+)	N/A – Banking not used
WREG Content Tracking(-Ow+)	All registers are automatically tracked
Procedural Abstraction(-Opa+)	Procedural Abstraction(-mpa)

Note 1: These optimization settings will satisfy most needs. Additional flags may be used for "fine-tuning". See **Section 3.5.6 "Options for Controlling Optimization"** for more information.

C.19 OBJECT MODULE FORMAT

The MPLAB C Compiler for PIC18 MCUs and the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs use different COFF File Formats that are not interchangeable.

C.20 IMPLEMENTATION-DEFINED BEHAVIOR

For the right-shift of a negative-signed integral value:

- The PIC18 MCU Compiler does not retain the sign bit
- The 16-Bit Device Compiler retains the sign bit

C.21 BIT FIELDS

Bit fields in the PIC18 MCU Compiler cannot cross byte storage boundaries and, therefore, cannot be greater than 8 bits in size.

The 16-Bit Device 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 cannot 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 16-Bit Device 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 16-Bit Device 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.

NOTES:

Appendix D. Diagnostics

D.1 INTRODUCTION

This appendix lists the most common diagnostic messages generated by the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB C30).

The compiler can produce two kinds of diagnostic messages: errors and warnings. Each kind has a different purpose.

- *Errors* reports problems that make it impossible to compile your program. The compiler reports errors with the source file name and line number where the problem is apparent.
- *Warnings* reports 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 where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of non-standard features of the compiler. Many warnings are issued only if you ask for them, with one of the `-W` options (for instance, `-Wall` requests a variety of useful warnings).

In rare instances, the compiler may issue an internal error message report. This signifies that the compiler itself has detected a fault that should be reported to Microchip support. Details on contacting support are contained elsewhere in this manual.

D.2 ERRORS

Symbols

\x used with no following HEX digits

The escape sequence \x should be followed by hex digits.

'&' constraint used with no register class

The asm statement is invalid.

'**%**' **constraint used with last operand**

The asm statement is invalid.

#elif after #else

In a preprocessor conditional, the #else clause must appear after any #elif clauses.

#elif without #if

In a preprocessor conditional, the #if must be used before using the #elif.

#else after #else

In a preprocessor conditional, the #else clause must appear only once.

#else without #if

In a preprocessor conditional, the #if must be used before using the #else.

#endif without #if

In a preprocessor conditional, the #if must be used before using the #endif.

#error '*message*'

This error appears in response to a #error directive.

#if with no expression

A 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 filename as an argument.

'#' is not followed by a macro parameter

The '#' operator should be followed by a macro argument name.

'##' cannot appear at either end of a macro expansion

The concatenation operator, '##' may not appear at the start or the end of a macro expansion.

A

a parameter list with an ellipsis can't match an empty parameter name list declaration

The declaration and definition of a function must be consistent.

"*symbol*" after #line is not a positive integer

#line is expecting a source line number which must be positive.

aggregate value used where a complex was expected

Do not use aggregate values where complex values are expected.

aggregate value used where a float was expected

Do not use aggregate values where floating-point values are expected.

aggregate value used where an integer was expected

Do not use aggregate values where integer values are expected.

alias arg not a string

The argument to the alias attribute must be a string that names the target for which the current identifier is an alias.

alignment may not be specified for '*identifier*'

The aligned attribute may only be used with a variable.

'__alignof' applied to a bit-field

The '`__alignof`' operator may not be applied to a bit-field.

alternate interrupt vector is not a constant

The interrupt vector number must be an integer constant.

alternate interrupt vector number *n* is not valid

A valid interrupt vector number is required.

ambiguous abbreviation argument

The specified command-line abbreviation is ambiguous.

an argument type that has a default promotion can't match an empty parameter name list declaration.

The declaration and definition of a function must be consistent.

args to be formatted is not ...

The first-to-check index argument of the format attribute specifies a parameter that is not declared '…'.

argument '*identifier*' doesn't match prototype

Function argument types should match the function's prototype.

argument of 'asm' is not a constant string

The argument of 'asm' must be a constant string.

argument to '-B' is missing

The directory name is missing.

argument to '-l' is missing

The library name is missing.

argument to '-specs' is missing

The name of the specs file is missing.

argument to '-specs=' is missing

The name of the specs file is missing.

argument to '-x' is missing

The language name is missing.

argument to '-Xlinker' is missing

The argument to be passed to the linker is missing.

arithmetic on pointer to an incomplete type

Arithmetic on a pointer to an incomplete type is not allowed.

array index in non-array initializer

Do not use array indices in non-array initializers.

array size missing in '*identifier*'

An array size is missing.

array subscript is not an integer

Array subscripts must be integers.

'asm' operand constraint incompatible with operand size

The asm statement is invalid.

'asm' operand requires impossible reload

The asm statement is invalid.

asm template is not a string constant

Asm templates must be string constants.

assertion without predicate

#assert or #unassert must be followed by a predicate, which must be a single identifier.

'*attribute*' attribute applies only to functions

The attribute '*attribute*' may only be applied to functions.

B

bit-field '*identifier*' has invalid type

Bit-fields must be of enumerated or integral type.

bit-field '*identifier*' width not an integer constant

Bit-field widths must be integer constants.

both long and short specified for '*identifier*'

A variable cannot be of type long and of type short.

both signed and unsigned specified for '*identifier*'

A variable cannot be both signed and unsigned.

braced-group within expression allowed only inside a function

It is illegal to have a braced-group within expression outside a function.

break statement not within loop or switch

Break statements must only be used within a loop or switch.

__builtin_longjmp second argument must be 1

`__builtin_longjmp` requires its second argument to be 1.

C

called object is not a function

Only functions may be called in C.

cannot convert to a pointer type

The expression cannot be converted to a pointer type.

cannot put object with volatile field into register

It is not legal to put an object with a volatile field into a register.

cannot reload integer constant operand in 'asm'

The asm statement is invalid.

cannot specify both near and far attributes

The attributes near and far are mutually exclusive, only one may be used for a function or variable.

cannot take address of bit-field '*identifier*'

It is not legal to attempt to take address of a bit-field.

can't open '*file*' for writing

The system cannot open the specified **'***file***'**. Possible causes are not enough disk space to open the file, the directory does not exist, or there is no write permission in the destination directory.

can't set '*attribute*' attribute after definition

The '*attribute***'** attribute must be used when the symbol is defined.

case label does not reduce to an integer constant

Case labels must be compile-time integer constants.

case label not within a switch statement

Case labels must be within a switch statement.

cast specifies array type

It is not permissible for a cast to specify an array type.

cast specifies function type

It is not permissible for a cast to specify a function type.

cast to union type from type not present in union

When casting to a union type, do so from type present in the union.

char-array initialized from wide string

Char-arrays should not be initialized from wide strings. Use ordinary strings.

***file: compiler* compiler not installed on this system**

Only the C compiler is distributed; other high-level languages are not supported.

complex invalid for '*identifier*'

The complex qualifier may only be applied to integral and floating types.

conflicting types for '*identifier*'

Multiple, inconsistent declarations exist for identifier.

continue statement not within loop

Continue statements must only be used within a loop.

conversion to non-scalar type requested

Type conversion must be to a scalar (not aggregate) type.

D

data type of '*name*' isn't suitable for a register

The data type does not fit into the requested register.

declaration for parameter '*identifier*' but no such parameter

Only parameters in the parameter list may be declared.

declaration of '*identifier*' as array of functions

It is not legal to have an array of functions.

declaration of '*identifier*' as array of voids

It is not legal to have an array of voids.

'*identifier*' declared as function returning a function

Functions may not return functions.

'*identifier*' declared as function returning an array

Functions may not return arrays.

decrement of pointer to unknown structure

Do not decrement a pointer to an unknown structure.

'default' label not within a switch statement

Default case labels must be within a switch statement.

'*symbol*' defined both normally and as an alias

A 'symbol' can not be used as an alias for another symbol if it has already been defined.

'defined' cannot be used as a macro name

The macro name cannot be called 'defined'.

dereferencing pointer to incomplete type

A dereferenced pointer must be a pointer to an incomplete type.

division by zero in #if

Division by zero is not computable.

duplicate case value

Case values must be unique.

duplicate label '*identifier*'

Labels must be unique within their scope.

duplicate macro parameter '*symbol*'

'symbol' has been used more than once in the parameter list.

duplicate member '*identifier*'

Structures may not have duplicate members.

duplicate (or overlapping) case value

Case ranges must not have a duplicate or overlapping value. The error message 'this is the first entry overlapping that value' will provide the location of the first occurrence of the duplicate or overlapping value. Case ranges are an extension of the ANSI standard for the compiler.

E

elements of array '*identifier*' have incomplete type

Array elements should have complete types.

empty character constant

Empty character constants are not legal.

empty file name in '#*keyword*'

The filename 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 pre prologue 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 filename.

‘*filename*’ is too large

The specified file is too large to process the file. Its probably larger than 4 GB, and the preprocessor refuses to deal with such large files. It is required that files be less than 4 GB in size.

ISO C forbids data definition with no type or storage class

A type specifier or storage class specifier is required for a data definition in ISO C.

ISO C requires a named argument before ‘...’

ISO C requires a named argument before ‘...’.

L

label *label* referenced outside of any function

Labels may only be referenced inside functions.

label ‘*label*’ used but not defined

The specified *label* is used but is not defined.

language ‘*name*’ not recognized

Permissible languages include: c assembler none.

***filename*: linker input file unused because linking not done**

The specified *filename* was specified on the command line, and it was taken to be a linker input file (since it was not recognized as anything else). However, the link step was not run. Therefore, this file was ignored.

long long long is too long for GCC

The compiler supports integers no longer than `long long`.

long or short specified with char for ‘*identifier*’

The long and short qualifiers cannot be used with the char type.

long or short specified with floating type for ‘*identifier*’

The long and short qualifiers cannot be used with the float type.

long, short, signed or unsigned invalid for ‘*identifier*’

The long, short and signed qualifiers may only be used with integral types.

M

macro names must be identifiers

Macro names must start with a letter or underscore followed by more letters, numbers or underscores.

macro parameters must be comma-separated

Commas are required between parameters in a list of parameters.

macro ‘*name*’ passed *n* arguments, but takes just *n*

Too many arguments were passed to macro ‘name’.

macro '*name*' requires *n* arguments, but only *n* given

Not enough arguments were passed to macro 'name'.

matching constraint not valid in output operand

The asm statement is invalid.

'*symbol*' may not appear in macro parameter list

'symbol' is not allowed as a parameter.

Missing '=' for 'save' in interrupt pragma

The save parameter requires an equal sign before the variable(s) are listed. For example, `#pragma interrupt isr0 save=var1,var2`

missing '('after predicate

#assert or #unassert expects parentheses around the answer. For example: ns`#assert PREDICATE (ANSWER)`

missing '(' in expression

Parentheses are not matching, expecting an opening parenthesis.

missing ')' after "defined"

Expecting a closing parenthesis.

missing ')' in expression

Parentheses are not matching, expecting a closing parenthesis.

missing ')' in macro parameter list

The macro is expecting parameters to be within parentheses and separated by commas.

missing ')' to complete answer

#assert or #unassert expects parentheses around the answer.

missing argument to '*option*' option

The specified command-line option requires an argument.

missing binary operator before token '*token*'

Expecting an operator before the 'token'.

missing terminating '*character*' character

Missing terminating character such as a single quote ', 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 lot 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 that 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.

D.3 WARNINGS

Symbols

'/*' within comment

A comment mark was found within a comment.

'$' character(s) in identifier or number

Dollar signs in identifier names are an extension to the standard.

#'*directive*' is a GCC extension

#warning, #include_next, #ident, #import, #assert and #unassert directives are GCC extensions and are not of ISO C89.

#import is obsolete, use an #ifndef wrapper in the header file

The #import directive is obsolete. #import was used to include a file if it hadn't already been included. Use the #ifndef directive instead.

#include_next in primary source file

`#include_next` starts searching the list of header file directories after the directory in which the current file was found. In this case, there were no previous header files so it is starting in the primary source file.

#pragma pack (pop) encountered without matching #pragma pack (push, <n>)

The pack(pop) pragma must be paired with a pack(push) pragma, which must precede it in the source file.

#pragma pack (pop, *identifier*) encountered without matching #pragma pack (push, *identifier*, <n>)

The pack(pop) pragma must be paired with a pack(push) pragma, which must precede it in the source file.

#warning: *message*

The directive #warning causes the preprocessor to issue a warning and continue preprocessing. The tokens following #warning are used as the warning message.

A

absolute address specification ignored

Ignoring the absolute address specification for the code section in the #pragma statement because it is not supported in the compiler. Addresses must be specified in the linker script and code sections can be defined with the keyword `__attribute__`.

address of register variable '*name*' requested

The register specifier prevents taking the address of a variable.

alignment must be a small power of two, not n

The alignment parameter of the pack pragma must be a small power of two.

anonymous enum declared inside parameter list

An anonymous enum is declared inside a function parameter list. It is usually better programming practice to declare enums outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous struct declared inside parameter list

An anonymous struct is declared inside a function parameter list. It is usually better programming practice to declare structs outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous union declared inside parameter list

An anonymous union is declared inside a function parameter list. It is usually better programming practice to declare unions outside parameter lists, since they can never become complete types when defined inside parameter lists.

anonymous variadic macros were introduced in C99

Macros which accept a variable number of arguments is a C99 feature.

argument ‘*identifier*’ might be clobbered by ‘longjmp’ or ‘vfork’

An argument might be changed by a call to longjmp. These warnings are possible only in optimizing compilation.

array ‘*identifier*’ assumed to have one element

The length of the specified array was not explicitly stated. In the absence of information to the contrary, the compiler assumes that it has one element.

array subscript has type ‘char’

An array subscript has type ‘*char*’.

array type has incomplete element type

Array types should not have incomplete element types.

asm operand *n* probably doesn’t match constraints

The specified extended asm operand probably doesn’t match its constraints.

assignment of read-only member ‘*name*’

The member ‘name’ was declared as const and cannot be modified by assignment.

assignment of read-only variable ‘*name*’

‘name’ was declared as const and cannot be modified by assignment.

‘*identifier*’ attribute directive ignored

The named attribute is not a known or supported attribute, and is therefore ignored.

‘*identifier*’ attribute does not apply to types

The named attribute may not be used with types. It is ignored.

‘*identifier*’ attribute ignored

The named attribute is not meaningful in the given context, and is therefore ignored.

‘*attribute*’ attribute only applies to function types

The specified attribute can only be applied to the return types of functions and not to other declarations.

B

backslash and newline separated by space

While processing for escape sequences, a backslash and newline were found separated by a space.

backslash-newline at end of file

While processing for escape sequences, a backslash and newline were found at the end of the file.

bit-field ‘*identifier*’ type invalid in ISO C

The type used on the specified identifier is not valid in ISO C.

braces around scalar initializer

A redundant set of braces around an initializer is supplied.

built-in function '*identifier*' declared as non-function

The specified function has the same name as a built-in function, yet is declared as something other than a function.

C

C++ style comments are not allowed in ISO C89

Use C style comments **'**/*' and **'***/**'** instead of C++ style comments **'**//**'**.

call-clobbered register used for global register variable

Choose a register that is normally saved and restored by function calls (W8-W13), so that library routines will not clobber it.

cannot inline function 'main'

The function **'**main**'** is declared with the *inline* attribute. This is not supported, since main must be called from the C start-up code, which is compiled separately.

can't inline call to '*identifier*' called from here

The compiler was unable to inline the call to the specified function.

case value '*n*' not in enumerated type

The controlling expression of a switch statement is an enumeration type, yet a case expression has the value n, which does not correspond to any of the enumeration values.

case value '*value*' not in enumerated type '*name*'

'*value***'** is an extra switch case that is not an element of the enumerated type **'***name***'**.

cast does not match function type

The return type of a function is cast to a type that does not match the function**'**s type.

cast from pointer to integer of different size

A pointer is cast to an integer that is not 16 bits wide.

cast increases required alignment of target type

When compiling with the `-Wcast-align` command-line option, the compiler verifies that casts do not increase the required alignment of the target type. For example, this warning message will be given if a pointer to char is cast as a pointer to int, since the aligned for char (byte alignment) is less than the alignment requirement for int (word alignment).

character constant too long

Character constants must not be too long.

comma at end of enumerator list

Unnecessary comma at the end of the enumerator list.

comma operator in operand of #if

Not expecting a comma operator in the #if directive.

comparing floating point with == or != is unsafe

Floating-point values can be approximations to infinitely precise real numbers. Instead of testing for equality, use relational operators to see whether the two values have ranges that overlap.

comparison between pointer and integer

A pointer type is being compared to an integer type.

comparison between signed and unsigned

One of the operands of a comparison is signed, while the other is unsigned. The signed operand will be treated as an unsigned value, which may not be correct.

comparison is always *n*

A comparison involves only constant expressions, so the compiler can evaluate the run time result of the comparison. The result is always *n*.

comparison is always *n* due to width of bit-field

A comparison involving a bit-field always evaluates to *n* because of the width of the bit-field.

comparison is always false due to limited range of data type

A comparison will always evaluate to false at run time, due to the range of the data types.

comparison is always true due to limited range of data type

A comparison will always evaluate to true at run time, due to the range of the data types.

comparison of promoted ~unsigned with constant

One of the operands of a comparison is a promoted ~unsigned, while the other is a constant.

comparison of promoted ~unsigned with unsigned

One of the operands of a comparison is a promoted ~unsigned, while the other is unsigned.

comparison of unsigned expression >= 0 is always true

A comparison expression compares an unsigned value with zero. Since unsigned values cannot be less than zero, the comparison will always evaluate to true at run time.

comparison of unsigned expression < 0 is always false

A comparison expression compares an unsigned value with zero. Since unsigned values cannot be less than zero, the comparison will always evaluate to false at run time.

comparisons like X<=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 catenate 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 noinline

The function 'name' has been declared as inline, but the noinline attribute prevents the function from being considered for inlining.

inlining failed in call to '*identifier*' called from here

The compiler was unable to inline the call to the specified function.

integer constant is so large that it is unsigned

An integer constant value appears in the source code without an explicit unsigned modifier, yet the number cannot be represented as a signed int; therefore, the compiler automatically treats it as an unsigned int.

integer constant is too large for '*type*' type

An integer constant should not exceed 2^32 - 1 for an unsigned long int, 2^63 - 1 for a long long int or 2^64 - 1 for an unsigned long long int.

integer overflow in expression

When folding an integer constant expression, the compiler found that the expression overflowed; that is, it could not be represented as an int.

invalid application of 'sizeof' to a function type

It is not recommended to apply the sizeof operator to a function type.

invalid application of 'sizeof' to a void type

The sizeof operator should not be applied to a void type.

invalid digit 'digit' in octal constant

All digits must be within the radix being used. For instance, only the digits 0 thru 7 may be used for the octal radix.

invalid second arg to __builtin_prefetch; using zero

Second argument must be `0` or `1`.

invalid storage class for function '*name*'

'auto' storage class should not be used on a function defined at the top level. 'static' storage class should not be used if the function is not defined at the top level.

invalid third arg to __builtin_prefetch; using zero

Third argument must be 0, 1, 2, or 3.

'*identifier*' is an unrecognized format function type

The specified *identifier*, used with the format attribute, is not one of the recognized format function types `printf`, `scanf`, or `strftime`.

'*identifier*' is narrower than values of its type

A bit-field member of a structure has for its type an enumeration, but the width of the field is insufficient to represent all enumeration values.

'*storage class*' is not at beginning of declaration

The specified storage class is not at the beginning of the declaration. Storage classes are required to come first in declarations.

ISO C does not allow extra ';' outside of a function

An extra ';' was found outside a function. This is not allowed by ISO C.

ISO C does not support '++' and '--' on complex types

The increment operator and the decrement operator are not supported on complex types in ISO C.

ISO C does not support '~' for complex conjugation

The bitwise negation operator cannot be use for complex conjugation in ISO C.

ISO C does not support complex integer types

Complex integer types, such as `__complex__` short int, are not supported in ISO C.

ISO C does not support plain 'complex' meaning 'double complex'

Using `__complex__` without another modifier is equivalent to 'complex double' which is not supported in ISO C.

ISO C does not support the '*char*' '*kind of format*' format

ISO C does not support the specification character 'char' for the specified 'kind of format'.

ISO C doesn't support unnamed structs/unions

All structures and/or unions must be named in ISO C.

ISO C forbids an empty source file

The file contains no functions or data. This is not allowed in ISO C.

ISO C forbids empty initializer braces

ISO C expects initializer values inside the braces.

ISO C forbids nested functions

A function has been defined inside another function.

ISO C forbids omitting the middle term of a ?: expression

The conditional expression requires the middle term or expression between the '?' and the ':'.

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 though 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*" re-asserted

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 >= 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 use to clearly express the intent of the programmer, even though the intent is unambiguous to the programmer and to the compiler.

T

'*identifier*' takes only zero or two arguments

Expecting zero or two arguments only.

the meaning of '\a' is different in traditional C

When the `-wtraditional` option is used, the escape sequence '\a' is not recognized as a meta-sequence: its value is just 'a'. In non-traditional compilation, '\a' represents the ASCII BEL character.

the meaning of '\x' is different in traditional C

When the `-wtraditional` option is used, the escape sequence '\x' is not recognized as a meta-sequence: its value is just 'x'. In non-traditional compilation, '\x' introduces a hexadecimal escape sequence.

third argument of '*identifier*' should probably be 'char **'

Expecting third argument of specified identifier to be of type 'char **'.

this function may return with or without a value

All exit paths from non-void function should return an appropriate value. The compiler detected a case where a non-void function terminates, sometimes with and sometimes without an explicit return value. Therefore, the return value might be unpredictable.

this target machine does not have delayed branches

The `-fdelayed-branch` option is not supported.

too few arguments for format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the number of actual arguments was fewer than that required by the format string.

too many arguments for format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the number of actual arguments was more than that required by the format string.

traditional C ignores #'*directive*' with the # indented

Traditionally, a directive is ignored unless its # is in column 1.

traditional C rejects initialization of unions

Unions cannot be initialized in traditional C.

traditional C rejects the 'ul' suffix

Suffix 'u' is not valid in traditional C.

traditional C rejects the unary plus operator

The unary plus operator is not valid in traditional C.

trigraph ??*char* converted to *char*

Trigraphs, which are a three-character sequence, can be used to represent symbols that may be missing from the keyboard. Trigraph sequences convert as follows:

??(= [	??) =]	??< = {	??> = }	??= = #	??/ = \	??' = ^	??! = |	??- = ~

trigraph ??*char* ignored

Trigraph sequence is being ignored. *char* can be (,), <, >, =, /, ', !, or -

type defaults to 'int' in declaration of '*identifier*'

In the absence of an explicit type declaration for the specified *identifier*, the compiler assumes that its type is int.

type mismatch with previous external decl
previous external decl of '*identifier*'

The type of the specified identifier does not match the previous declaration.

type mismatch with previous implicit declaration

An explicit declaration conflicts with a previous implicit declaration.

type of '*identifier*' defaults to 'int'

In the absence of an explicit type declaration, the compiler assumes that *identifier*'s type is int.

type qualifiers ignored on function return type

The type qualifier being used with the function return type is ignored.

U

undefining 'defined'

'defined' cannot be used as a macro name and should not be undefined.

undefining '*name*'

The #undef directive was used on a previously defined macro name 'name'.

union cannot be made transparent

The `transparent_union` attribute was applied to a union, but the specified variable does not satisfy the requirements of that attribute.

'union *identifier*' declared inside parameter list

The specified union is declared inside a function parameter list. It is usually better programming practice to declare unions outside parameter lists, since they can never become complete types when defined inside parameter lists.

union defined inside parms

A union is defined inside a function parameter list.

union has no members

The union is empty, it has no members.

unknown conversion type character '*character*' in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the conversion characters in the format string was invalid (unrecognized).

unknown conversion type character 0x*number* in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the conversion characters in the format string was invalid (unrecognized).

unknown escape sequence '*sequence*'

'sequence' is not a valid escape code. An escape code must start with a '\' and use one of the following characters: n, t, b, r, f, b, \, ', ", a, or ?, or it must be a numeric sequence in octal or hex. In octal, the numeric sequence must be less than 400 octal. In hex, the numeric sequence must start with an 'x' and be less than 100 hex.

unnamed struct/union that defines no instances

struct/union is empty and has no name.

unreachable code at beginning of *identifier*

There is unreachable code at beginning of the specified function.

unrecognized gcc debugging option: *char*

The 'char' is not a valid letter for the -d*letters* debugging option.

unused parameter '*identifier*'

The specified function parameter is not used in the function.

unused variable '*name*'

The specified variable was declared but not used.

use of '*' and 'flag' together in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that both the flags '*' and 'flag' appear in the format string.

use of C99 long long integer constants

Integer constants are not allowed to be declared long long in ISO C89.

use of '*length*' length modifier with '*type*' type character

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the specified length was incorrectly used with the specified *type*.

'*name*' used but never defined

The specified function was used but never defined.

'*name*' used with '*spec*' '*function*' format

'name' is not valid with the conversion specification 'spec' in the format of the specified function.

useless keyword or type name in empty declaration

An empty declaration contains a useless keyword or type name.

V

__VA_ARGS__ can only appear in the expansion of a C99 variadic macro

The predefined macro __VA_ARGS should be used in the substitution part of a macro definition using ellipses.

value computed is not used

A value computed is not used.

variable ‘*name*’ declared ‘inline’

The keyword ‘inline’ should be used with functions only.

variable ‘%s’ might be clobbered by ‘longjmp’ or ‘vfork’

A non-volatile automatic variable might be changed by a call to longjmp. These warnings are possible only in optimizing compilation.

volatile register variables don’t work as you might wish

Passing a variable as an argument could transfer the variable to a different register (w0-w7) than the one specified (if not w0-w7) for argument transmission. Or the compiler may issue an instruction that is not suitable for the specified register and may need to temporarily move the value to another place. These are only issues if the specified register is modified asynchronously (i.e., though an ISR).

W

-Wformat-extra-args ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-extra-args`.

-Wformat-nonliteral ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-nonliteral`.

-Wformat-security ignored without -Wformat

`-Wformat` must be specified to use `-Wformat-security`.

-Wformat-y2k ignored without -Wformat

`-Wformat` must be specified to use.

-Wid-clash-LEN is no longer supported

The option `-Wid-clash-LEN` is no longer supported.

-Wmissing-format-attribute ignored without -Wformat

`-Wformat` must be specified to use `-Wmissing-format-attribute`.

-Wuninitialized is not supported without -O

Optimization must be on to use the `-Wuninitialized` option.

‘*identifier*’ was declared ‘extern’ and later ‘static’

The specified identifier was previously declared ‘extern’ and is now being declared as static.

‘*identifier*’ was declared implicitly ‘extern’ and later ‘static’

The specified identifier was previously declared implicitly ‘extern’ and is now being declared as static.

‘*identifier*’ was previously implicitly declared to return ‘int’

There is a mismatch against the previous implicit declaration.

'*identifier*' was used with no declaration before its definition

When compiling with the `-Wmissing-declarations` command-line option, the compiler ensures that functions are declared before they are defined. In this case, a function definition was encountered without a preceding function declaration.

'*identifier*' was used with no prototype before its definition

When compiling with the `-Wmissing-prototypes` command-line option, the compiler ensures that function prototypes are specified for all functions. In this case, a function call was encountered without a preceding function prototype for the called function.

writing into constant object (arg *n*)

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the specified argument number *n* was a const object that the format specifier indicated should be written into.

Z

zero-length *identifier* format string

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that the format string was empty ("").

Appendix E. Deprecated Features

E.1 INTRODUCTION

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects which depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependency on deprecated features. Support for these features may be removed entirely in future versions of the language tools.

E.2 HIGHLIGHTS

Deprecated features covered are:

- Predefined Constants
- Variables in Specified Registers

E.3 PREDEFINED CONSTANTS

The following preprocessing symbols are defined by the compiler.

Symbol	Defined with -ansi command-line option?
dsPIC30	No
__dsPIC30	Yes
__dsPIC30__	Yes

The ELF-specific version of the compiler defines the following preprocessing symbols.

Symbol	Defined with -ansi command-line option?
dsPIC30ELF	No
__dsPIC30ELF	Yes
__dsPIC30ELF__	Yes

The COFF-specific version of the compiler defines the following preprocessing symbols.

Symbol	Defined with -ansi command-line option?
dsPIC30COFF	No
__dsPIC30COFF	Yes
__dsPIC30COFF__	Yes

For the most current information, see **Section 3.7 "Predefined Macro Names"**.

E.4 VARIABLES IN SPECIFIED REGISTERS

The compiler allows you to put a few global variables into specified hardware registers.

> **Note:** Using too many registers, in particular register W0, may impair the ability of the 16-bit compiler to compile. It is not recommended that registers be placed into fixed registers.

You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses. Stores into local register variables may be deleted when they appear to be unused. References to local register variables may be deleted, moved or simplified.

These local variables are sometimes convenient for use with the extended inline assembly (see **Chapter 9. "Mixing Assembly Language and C Modules"**), if you want to write one output of the assembler instruction directly into a particular register. (This will work provided the register you specify fits the constraints specified for that operand in the inline assembly statement).

E.4.1 Defining Global Register Variables

You can define a global register variable like this:

```
register int *foo asm ("w8");
```

Here `w8` is the name of the register which should be used. Choose a register that is normally saved and restored by function calls (W8-W13), so that library routines will not clobber it.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted, moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them especially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e., in a source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. This problem can be avoided by recompiling `qsort` with the same global register variable definition.

If you want to recompile `qsort` or other source files that do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler command-line option `-ffixed-`*`reg`*. You need not actually add a global register declaration to their source code.

A function that can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function that is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value that belongs to its caller.

The library function `longjmp` will restore to each global register variable the value it had at the time of the `setjmp`.

All global register variable declarations must precede all function definitions. If such a declaration appears after function definitions, the register may be used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

E.4.2 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.

NOTES:

Appendix F. ASCII Character Set

TABLE F-1: ASCII CHARACTER SET

Least Significant Character \ Most Significant Character	0	1	2	3	4	5	6	7
Hex								
0	NUL	DLE	Space	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NOTES:

Appendix G. GNU Free Documentation License

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

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

G.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, 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.

G.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of

mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "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.

G.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

G.4 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

G.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

d) Preserve all the copyright notices of the Document.

e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

h) Include an unaltered copy of this License.

i) Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

k) For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

m) Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

n) Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

G.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

G.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

G.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

G.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

G.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

G.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free 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.

G.12 RELICENSING

"Massive Multiauthor 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 Multiauthor 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 siteunder CC-BY-SA on the same site at any time before August 1, 2009,provided the MMC is eligible for relicensing.

NOTES:

Glossary

This glossary applies to al Microchip development tools documentation. Therefore, some terms have tool-dependent meanings. An abbreviation of each tool is used to identify each meaning. Currently, the following abbreviations are used:

IDE - MPLAB IDE

PM3 - MPLAB PM3 Programmer

C18 - MPLAB C Compiler for PIC18 MCUs

C30 - MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs

ASM30 - MPLAB Assembler for PIC24 MCUs and dsPIC DSCs

Absolute Section

A section with a fixed (absolute) address that cannot be changed by the linker.

Access Memory

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

Access Entry Points

Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. They support the separate linking of boot and secure application segments.

Address

Value that identifies a location in memory.

Alphabetic Character

Alphabetic characters are those characters that are letters of the English alphabet (a, b, …, z, A, B, …, Z).

Alphanumeric

Alphanumeric characters are comprised of alphabetic characters and integers (0,1, …, 9).

ANDed Breakpoints

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

Anonymous Structure

C30 – An unnamed structure.

C18 – An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union. For example, in the following code, `hi` and `lo` are members of an anonymous structure inside the union `caster`.

```
union castaway
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

ANSI

American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

Application

A set of software and hardware that may be controlled by a PIC® microcontroller.

Archive

A collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver to combine the object files into one library file. A library can be linked with object modules and other libraries to create executable code.

Archiver

A tool that creates and manipulates libraries.

ASCII

American Standard Code for Information Interchange is a character set encoding that uses seven binary digits to represent each character. It includes upper and lower case letters, digits, symbols, and control characters.

Assembler

A language tool that translates assembly language source code into machine code.

Assembly Language

A programming language that describes binary machine code in a symbolic form.

Assigned Section

A section which has been assigned to a target memory block in the linker script file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

Asynchronous Stimulus

Data generated to simulate external inputs to a simulator device.

Attribute

Characteristics of variables or functions in a C program that are used to describe machine-specific properties.

Attribute, Section

Characteristics of sections, such as "executable", "readonly", or "data" that can be specified as flags in the assembler `.section` directive.

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones (2^0), the next counts multiples of two (2^1), the next counts multiples of four (2^2), etc.

Bookmarks

Use bookmarks to easily locate specific lines in a file.

Under the Edit menu, select Bookmarks to manage bookmarks. Toggle (enable/disable) a bookmark, move to the next or previous bookmark, or clear all bookmarks.

Breakpoint

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build

Compile and link all the source files for an application.

C

A general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators.

Calibration Memory

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

Clean

Under the MPLAB IDE Project menu, Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

COFF

Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface

A means of communication between a program and its user based solely on textual input and output.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Assembly

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU

See Central Processing Unit.

Cross Reference File

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

Data Directives

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

Data Memory

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Debugger

Hardware that performs debugging.

Debugger System

The Debugger systems include the pod, processor module, device adapter, target board, cables, and MPLAB IDE software.

Debugging Information

Compiler and assembler options that, when selected, provide varying degrees of information that is used to debug application code. See compiler or assembler documentation for details on selecting debug options.

Deprecated Features

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller

A microcontroller device with digital signal processing capability, i.e., Microchip dsPIC DSC devices.

Digital Signal Processing

The computer manipulation of digital signals, i.e., analog signals (sound or image) which have been converted to digital form (sampled).

Digital Signal Processor

A microprocessor that is designed for use in digital signal processing.

Directives

Statements in source code that provide control of the language tool's operation.

Download

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

DSC

See Digital Signal Controller.

DSP

See Digital Signal Processor.

dsPIC DSCs

dsPIC Digital Signal Controllers (DSCs) refers to all Microchip DSC families.

DWARF

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

Emulation

The process of executing software loaded into emulation memory as if it were firmware residing on a microcontroller device.

Emulation Memory

Program memory contained within the emulator.

Emulator

Hardware that performs emulation.

Emulator System

The MPLAB REAL ICE system consists of a pod, a driver (and potentially a receiver) card, target board, cables, and MPLAB IDE software.

Endianness

The ordering of bytes in a multi-byte object.

Environment

IDE **–** The particular layout of the desktop for application development.

PM3 **–** A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

Epilogue

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that usually can be erased by exposure to ultraviolet radiation.

Error File

A file containing error messages and diagnostics generated by a language tool.

Errors

Errors report problems that make it impossible to continue processing your program. When possible, errors identify the source file name and line number where the problem is apparent.

Event

A description of a bus cycle which may include address, data, pass count, external input, cycle type (e.g., fetch, R/W), and time stamp. Events are used to describe triggers, breakpoints and interrupts.

Executable Code

Software that is ready to be loaded for execution.

Export

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

Expressions

Combinations of constants and/or symbols separated by arithmetic or logical operators.

Extended Microcontroller Mode

In Extended Microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC18 device.

Extended Mode (PIC18 MCUs)

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

External Label

A label that has external linkage.

External Linkage

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

External Symbol

A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution

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

External Input Line

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

External RAM

Off-chip Read/Write memory.

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

File Registers

On-chip data memory, including GPRs and SFRs.

Filter

Determine by selection what data is included/excluded in a trace display or data file.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the PC, this prefetched instruction is explicitly ignored, causing an FNOP cycle.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

Free-Standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI '89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

GPR

General Purpose Register. The portion of device data memory (RAM) available for general use.

Halt

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Header, Debug

A circuit board containing a special debug device (-ICE/-ICD) used with in-circuit debuggers and emulators to debug application code. For low pin count devices, resources can be recovered when using a debug header. See the *Header Board Specification* (DS51292) for details.

Heap

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code

Executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hex File

An ASCII file containing hexadecimal addresses and values (hex code) suitable for programming a device.

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones (16^0), the next counts multiples of 16 (16^1), the next counts multiples of 256 (16^2), etc.

High Level Language

A language for writing programs that is further removed from the processor than assembly.

ICD

In-Circuit Debugger. The MPLAB ICD3 In-Circuit Debugger, MPLAB ICD2 In-Circuit Debugger, PICkit 3 D.E. In-Circuit Debugger (Debug Express add-on), and PICkit 2 D.E. In-Circuit Debugger (Debug Express add-on) are the Microchip in-circuit debuggers.

ICE

In-Circuit Emulator. The MPLAB REAL ICE system is Microchip's next-generation in-circuit emulator.

ICSP

In-Circuit Serial Programming. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

IDE

Integrated Development Environment. MPLAB IDE is Microchip's integrated development environment.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

Bring data into the MPLAB IDE from an outside source, e.g., from a hex file.

Initialized Data

Data that is defined with an initial value. In C,

```
int myVar=5;
```

defines a variable that will reside in an initialized data section.

Instruction Set

The collection of machine language instructions that a particular processor understands.

Instructions

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

Internal Linkage

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

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications.

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Request

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine

ALU30, C18, C30 **–** A function that handles an interrupt.

IDE **–** User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

IRQ

See Interrupt Request.

ISO

See International Organization for Standardization.

ISR

See Interrupt Service Routine.

L-value

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency

The time between an event and its response.

Librarian

See Archiver.

Library

See Archive.

Linker

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives

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

Listing File

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

Little Endian

A data ordering scheme for multi-byte data whereby the least significant byte is stored at the lower addresses.

Local Label

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

Logic Probes

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Loop-Back Test Board

Used to test the functionality of the MPLAB REAL ICE in-circuit emulator.

LVDS

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

LVDS differs from normal input/output (I/O) in a few ways:

Normal digital I/O works with 5 volts as a high (binary '1') and 0 volts as a low (binary '0'). When you use a differential, you add a third option (-5 volts), which provides an extra level with which to encode, and results in a higher maximum data transfer rate.

A higher data transfer rate means fewer wires are required, as in UW (Ultra Wide) and UW-2/3 SCSI hard disks, which use only 68 wires. These devices require a high transfer rate over short distances. Using standard I/O transfer, SCSI hard drives would require a lot more than 68 wires.

Low voltage means that the standard 5 volts is replaced by either 3.3 volts or 1.5 volts.

LVDS uses a dual wire system, running 180 degrees of each other. This enables noise to travel at the same level, which in turn can get filtered more easily and effectively.

With standard I/O signaling, data storage is contingent on the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: http://www.webopedia.com/TERM/L/LVDS.html.

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro

A macro instruction is an instruction that represents a sequence of instructions in abbreviated form.

Macro Directives

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

Makefile

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB IDE, i.e., with a `make`.

Under *Project>Build Options>Project*, **Directories** tab, you must have selected "Assemble/Compile/Link in the project directory" under "Build Directory Policy" for this feature to work.

Make Project

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

MCLR - Master Clear

Master Clear (MCLR) is a function on a pin that causes a processor Reset. MCLR is usually multiplexed with other functions such as V_{PP}. Also, MCLR is usually complementary ($\overline{MCLR}$); that is, when the pin is pulled low, a Reset occurs, and when the pin is pulled high, the device operates normally.

Internal MCLR – When the MCLR enable configuration bit is cleared (0), a Reset signal is generated internally.

External MCLR – When the MCLR enable configuration bit is set (1), the pin becomes an external Reset input.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also uC.

Memory Model

C30 **–** A representation of the memory available to the application.

C18 **–** A description that specifies the size of pointers that point to program memory.

Message

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller

A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

Microcontroller Mode

One of the possible program memory configurations of PIC18 microcontrollers. In Microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in Microcontroller mode.

Microprocessor Mode

One of the possible program memory configurations of PIC18 microcontrollers. In Microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

Text instructions that can be translated directly into machine code. Also referred to as opcodes.

MPASM™ Assembler

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices, and Microchip memory devices.

MPLAB *Language Tool* for *Device*

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

MPLAB ICD

Microchip in-circuit debuggers that work with MPLAB IDE. The ICDs support Flash devices with built-in debug circuitry. The main component of each ICD is the pod. A complete system consists of a pod, debug header (with a *device*-ICD), target board, cables, and MPLAB IDE software.

MPLAB IDE

Microchip's Integrated Development Environment. MPLAB IDE comes with an editor, project manager, and simulator.

MPLAB PM3

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB IDE or as a stand-alone programmer. Replaces PRO MATE II.

MPLAB REAL ICE™ In-Circuit Emulator

Microchip next-generation in-circuit emulator that works with MPLAB IDE. The MPLAB REAL ICE emulator supports PIC MCUs and dsPIC DSCs. The main component of each ICE is the pod. A complete system consists of a pod, a driver (and, potentially, a receiver) card, cables, and MPLAB IDE software.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE in support of PIC MCU and dsPIC DSC devices.

MPLIB™ Object Librarian

Microchip's librarian that can work with MPLAB IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler or MPLAB C18 C compiler.

MPLINK™ Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though it is not a necessity.

MRU

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

Native Data Size

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device's data memory: bytes for PIC18 devices and words for 16-bit devices.

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB IDE project component.

Non-Extended Mode (PIC18 MCUs)

In Non-Extended mode, the compiler will not use the extended instructions nor the indexed with literal offset addressing.

Non Real Time

Refers to the processor at a breakpoint, or executing single-step instructions, or MPLAB IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

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

Object Code

The machine code generated by an assembler or compiler.

Object File

A file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives

Directives that are used only when creating an object file.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones (8^0), the next digit counts multiples of eight (8^1), the next digit counts multiples of 64 (8^2), etc.

Off-Chip Memory

Off-chip memory refers to the memory selection option for the PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

One-to-One Project-Workspace Model

The most common configuration for application development in MPLAB IDE to is have one project in one workspace. Select *Configure>Settings*, **Projects** tab and check “Use one-to-one project-workspace model”.

Opcodes

Operational Codes. *See* Mnemonics.

Operators

Symbols, like the plus sign ‘+’ and the minus sign ‘-’, that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

Pass Counter

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or PC.

PC Host

Any PC running a supported Windows operating system.

Persistent Data

Data that is never cleared or initialized. This allows an application to preserve data across a device Reset.

Phantom Byte

An unimplemented byte in the dsPIC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC hex files.

PIC MCUs

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

PICkit 1, 2, and 3

Microchip's developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

Plug-ins

The MPLAB IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools may be found under the Tools menu.

Pod

MPLAB REAL ICE system: The box that contains the emulation control circuitry for the ICE device on the header or target board. An ICE device can be a production device with built-in ICE circuitry or a special ICE version of a production device (i.e., *device*-ICE).

MPLAB ICD: The box that contains the debug control circuitry for the ICD device on the header or target board. An ICD device can be a production device with built-in ICD circuitry or a special ICD version of a production device (i.e., *device*-ICD).

Power-on-Reset Emulation

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

Pragma

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

A production programmer is a tool that has resources designed into it that program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

Microchip production programmers, such as MPLAB PM3, MPLAB REAL ICE in-circuit emulator, and MPLAB ICD 3, have been designed to be robust enough to tolerate these demanding environments.

Some top-end tools have additional accessories. The MPLAB REAL ICE Performance Pak has accelerators to speed up the communication and ICSP process. The MPLAB PM3 programmer has interchangeable socket modules to support various devices out-of-circuit.

Profile

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

Program Counter

The location that contains the address of the instruction that is currently executing.

Program Counter Unit

ALU30 – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory

IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

ALU30, C30 – The memory area in a device where instructions are stored.

Project

A project contains the files that are needed to build an application (source code, linker script files, etc.) along with their associations, to various build tools and build options.

Prologue

A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement that is specified in the runtime model. This code executes before user code for a given function.

Prototype System

A term referring to a user's target application, or target board.

PWM Signals

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

Qualifier

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

Radix

The number base, hex, or decimal, used in specifying an address.

Random Access Memory (RAM)

Data emory in which information can be accessed in any order.

Raw Data

The binary representation of code or data associated with a section.

Read-Only Memory (ROM)

Memory hardware that allows fast access to permanently stored data, but prevents addition to, or modification of, the data.

Real Time

When an in-circuit emulator or debugger is released from the Halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Real-Time Watch

A Watch window where the variables change in real-time as the application is run. See individual tool documentation to determine how to set up a real-time watch. Not all tools support real-time watches.

Recursive Calls

A function that calls itself, either directly or indirectly.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion, or through execution during interrupt processing.

Relaxation

The process of converting an instruction to an identical, but smaller, instruction. This is useful for saving on code size. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object whose address has not been assigned to a fixed location in memory.

Relocatable Section

ALU30 – A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

ROM

Read-Only Memory (Program Memory). Memory that cannot be modified.

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

Run-time Model

Describes the use of target architecture resources.

Scenario

For MPLAB SIM simulator, a particular setup for stimulus control.

Section

A portion of an application located at a specific address of memory.

Section Attribute

A characteristic ascribed to a section (e.g., an `access` section).

Sequenced Breakpoints

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up, i.e., the last breakpoint in the sequence occurs first.

Serialized Quick Turn Programming

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

SFR

See Special Function Registers.

Shell

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

Simulator

A software program that models the operation of devices.

Single Step

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcode appears on the bus as a fetch during the execution of the previous instruction. The source data address and value, and the destination data address appear when the opcode is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code

A text listing of commands that may be completed or assembled into object code. Source code is written in a formal programming language that can be translated into machine code or executed by an interpreter.

Source File

An ASCII text file containing source code.

Special Function Registers

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers, or other modes or peripherals.

SQTP

See Serialized Quick Turn Programming.

Stack, Hardware

Locations in MCU or DSC where the return address is stored when a function call is made.

Stack, Software

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is typically managed by the compiler when developing code in a high-level language.

MPLAB Starter Kit for *Device*

Microchip's starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program your own changes.

Static RAM or SRAM

Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

Status Bar

The Status Bar is located on the bottom of the MPLAB IDE window and indicates current information, such as cursor position, development mode and device, and active tool bar.

Step Into

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

Step Over

Step Over allows you to debug code without stepping into subroutines. When stepping over a `CALL` instruction, the next breakpoint will be set at the instruction after the `CALL`. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of `CALL` instructions.

Step Out

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked, or register.

Stopwatch

A counter for measuring execution cycles.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

Symbol

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute

Represents an immediate value, such as a definition, through the assembly `.equ` directive.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.”

Target

Refers to user hardware.

Target Application

Software residing on the target board.

Target Board

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

Target Processor

The microcontroller device on the target application board.

Template

Lines of text that you build to insert into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar

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

Trace

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE’s trace window.

Trace Memory

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

Trace Macro

A macro that will provide trace information from emulator data. Since this is a software trace: the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

Trigger Output

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs

Three-character sequences, all starting with ??, that are defined by ISO C as replacements for single characters.

Unassigned Section

A section that has not been assigned to a specific target memory block in the linker script file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

Data that is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

Upload

The Upload function transfers data from a tool (such as an emulator or programmer) to the host PC, or from the target board to the emulator.

USB

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. USB 2.0 supports data rates up to 480 Mbps and is often referred to as high-speed USB.

Vector

The memory locations that an application will jump to when a Reset or an interrupt occurs.

Warning

IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

ALU30, C30 – Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text '`warning`:' to distinguish them from error messages.

Watch Variable

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

Watch Window

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

Watchdog Timer

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

WDT

See Watchdog Timer.

Workbook

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

WorkSpace

A workspace contains MPLAB IDE information on the selected device, selected debug tool and/or programmer, open windows and their location, and other IDE configuration settings.

Index

Symbols

Numerics

A

D

E

X

Worldwide Sales and Service

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://www.microchip.com/support
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-2819-3180
Fax: 86-571-2819-3189

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung
Tel: 886-7-213-7830
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

05/02/11