



**MPLAB[®] C30
C COMPILER
USER'S GUIDE**

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

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

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

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contained in this publication regarding device applications and the like is intended through suggestion only and may be superseded by updates.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

Trademarks

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


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

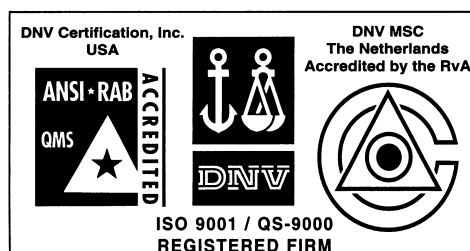
Application Maestro, dsPICDEM, dsPICDEM.net, ECAN, ECONOMONITOR, FanSense, FlexROM, fuzzyLAB, In-Circuit Serial Programming, ICSP, ICEPIC, microPort, Migratable Memory, MPASM, MPLIB, MPLINK, MPSIM, PICkit, PICDEM, PICDEM.net, PowerCal, PowerInfo, PowerMate, PowerTool, rLAB, rPIC, Select Mode, SmartSensor, SmartShunt, SmartTel and Total Endurance are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

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

© 2003, Microchip Technology Incorporated, Printed in the U.S.A., All Rights Reserved.

 Printed on recycled paper.



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

Table of Contents

Preface	1
Chapter 1. Compiler Overview	
1.1 Introduction	7
1.2 Highlights	7
1.3 MPLAB C30 Description	7
1.4 MPLAB C30 and Other Development Tools	7
1.5 MPLAB C30 Feature Set	9
Chapter 2. Differences Between MPLAB C30 and ANSI C	
2.1 Introduction	11
2.2 Highlights	11
2.3 Keyword Differences	11
2.4 Statement Differences	23
2.5 Built-in return address Function	24
Chapter 3. Using MPLAB C30 C Compiler	
3.1 Introduction	25
3.2 Highlights	25
3.3 Overview	25
3.4 File Naming Conventions	26
3.5 Options	26
3.6 Environment Variables	50
3.7 Compiling a Single File on the Command Line	51
3.8 Compiling Multiple Files on the Command Line	52
Chapter 4. MPLAB C30 C Compiler Runtime Environment	
4.1 Introduction	53
4.2 Highlights	53
4.3 Address Spaces	53
4.4 Code and Data Sections	55
4.5 Startup and Initialization	56
4.6 Memory Spaces	57
4.7 Memory Models	57
4.8 X and Y Data Spaces	59
4.9 Locating Code and Data	60
4.10 Software Stack	61
4.11 The C Stack Usage	62
4.12 The C Heap Usage	64
4.13 Function Call Conventions	65
4.14 Register Conventions	67

4.15 Bit Reversed and Modulo Addressing	68
4.16 Program Space Visibility (PSV) Usage	68
Chapter 5. Data Types	
5.1 Introduction	69
5.2 Highlights	69
5.3 Data Representation	69
5.4 Integer	69
5.5 Floating Point	70
5.6 Pointers	70
Chapter 6. Device Support Files	
6.1 Introduction	71
6.2 Highlights	71
6.3 Processor Header Files	71
6.4 Register Definition Files	72
6.5 Using SFRs	73
6.6 Using Macros	75
Chapter 7. Interrupts	
7.1 Introduction	77
7.2 Highlights	77
7.3 Writing an Interrupt Service Routine	78
7.4 Writing the Interrupt Vector	80
7.5 Interrupt Service Routine Context Saving	82
7.6 Latency	83
7.7 Nesting Interrupts	83
7.8 Enabling/Disabling Interrupts	83
Chapter 8. Mixing Assembly Language and C Modules	
8.1 Introduction	85
8.2 Highlights	85
8.3 Mixing Assembly Language and C Variables and Functions	85
8.4 Using Inline Assembly Language	87
Appendix A. Implementation-Defined Behavior	
A.1 Introduction	91
A.2 Highlights	91
A.3 Translation	92
A.4 Environment	92
A.5 Identifiers	93
A.6 Characters	93
A.7 Integers	94
A.8 Floating Point	94
A.9 Arrays and Pointers	95
A.10 Registers	95
A.11 Structures, Unions, Enumerations and Bitfields	96
A.12 Qualifiers	96

A.13 Declarators	96
A.14 Statements	96
A.15 Preprocessing Directives	97
A.16 Library Functions	98
A.17 Signals	99
A.18 Streams and Files	99
A.19 tmpfile	100
A.20 errno	100
A.21 Memory	100
A.22 abort	100
A.23 exit	100
A.24 getenv	101
A.25 system	101
A.26 strerror	101
Appendix B. MPLAB C30 C Compiler Diagnostics	
B.1 Introduction	103
B.2 Highlights	103
B.3 Errors	103
B.4 Warnings	126
Appendix C. MPLAB C18 vs. MPLAB C30 C Compiler	
C.1 Introduction	149
C.2 Highlights	149
C.3 Data Formats	150
C.4 Pointers	150
C.5 Storage Classes	150
C.6 Stack Usage	150
C.7 Storage Qualifiers	151
C.8 Predefined Macro Names	151
C.9 Integer Promotions	151
C.10 Numeric Constants	151
C.11 String Constants	151
C.12 Anonymous Structures	152
C.13 Access Memory	152
C.14 Inline Assembly	152
C.15 Pragmas	152
C.16 Memory Models	154
C.17 Calling Conventions	154
C.18 Startup Code	154
C.19 Compiler-Managed Resources	154
C.20 Optimizations	155
C.21 Object Module Format	155
C.22 Implementation-Defined Behavior	155
C.23 Bitfields	156

MPLAB® C30 User's Guide

Appendix D. ASCII Character Set 157

Appendix E. GNU Free Documentation License 159

Glossary 165

Index 173

Worldwide Sales and Service 182

Preface

INTRODUCTION

The purpose of this document is to help you use Microchip's MPLAB C30 C compiler for dsPIC® devices to develop your application. MPLAB C30 is a GNU-based language tool, based on source code from the Free Software Foundation (FSF). For more information about the FSF, see www.fsf.org.

Other GNU language tools available from Microchip are:

- MPLAB ASM30 Assembler
- MPLAB LINK30 Linker
- MPLAB LIB30 Librarian/Archiver

HIGHLIGHTS

The information covered in this chapter is as follows:

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

ABOUT THIS GUIDE

Document Layout

The document layout is as follows:

- **Chapter 1: Compiler Overview** – describes MPLAB C30, development tools and feature set.
- **Chapter 2: Differences between MPLAB C30 and ANSI C** – describes the differences between the C language supported by MPLAB C30 syntax and the standard ANSI-89 C.
- **Chapter 3: Using MPLAB C30** – describes how to use the MPLAB C30 compiler from the command line.
- **Chapter 4: MPLAB C30 Runtime Environment** – describes the MPLAB C30 runtime model, including information on sections, initialization, memory models, the software stack and much more.
- **Chapter 5: Data Types** – describes MPLAB C30 integer, floating point and pointer data types.
- **Chapter 6: Device Support Files** – describes the MPLAB C30 header and register definition files, as well as how to use with SFR's.
- **Chapter 7: Interrupts** – describes how to use interrupts.
- **Chapter 8: Mixing Assembly Language and C Modules** – provides guidelines to using MPLAB C30 with MPLAB ASM30 assembly language modules.

- **Appendix A: Implementation-Defined Behavior** – details MPLAB C30 specific parameters described as implementation-defined in the ANSI standard.
- **Appendix B: MPLAB C30 Diagnostics** – lists error and warning messages generated by MPLAB C30.
- **Appendix C: Differences Between MPLAB C18 and MPLAB C30** – highlights the differences between the PIC18XXXXXX compiler (MPLAB C18) and the dsPIC compiler (MPLAB C30).
- **Appendix D: ASCII Character Set** – contains the ASCII character set.
- **Appendix E: GNU Free Documentation License** – usage license for the Free Software Foundation.

Conventions Used in this Guide

This manual uses the following documentation conventions:

TABLE 1: DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Main Document (Arial font):		
Italic characters	Referenced books	<i>MPLAB IDE User's Guide</i>
	Emphasized text	...is the <i>only</i> compiler...
Interface References (Arial font):		
Initial caps	A window, dialog or menu selection	Configuration Bits window, Settings dialog, Enable Programmer
Quotes	A field name in a window or dialog	"Save files before running the debugger"
Underlined, italic text with right arrow	A menu selection path	<u>File>Save</u>
Bold characters	A dialog button or tab	OK button, Power tab
Characters in angle brackets < >	A key on the keyboard	<Tab>, <Ctrl-C>
Code References (Courier font):		
Plain characters	File names and paths	c:\autoexec.bat
	Bit values	0, 1
	Sample code	#define START
Square brackets []	Optional arguments	mpasmwin [main.asm]
Curly brackets and pipe character: { }	Choice of mutually exclusive arguments An OR selection	errorlevel {0 1}
Italic characters	A variable argument; it can be either a type of data (in lower case characters) or a specific example (in uppercase characters).	pic30-gcc <i>filename</i>
Ellipses...	Replaces repeated instances of text	list ["list_option...", "list_option"]
0xnnnn	A hexadecimal number where <i>n</i> is a hexadecimal digit	0xFFFF, 0x007A, 0x1A
'bnnnn	A binary number where <i>n</i> is a digit	'b00100, 'b10

Documentation Updates

All documentation becomes dated, and this document is no exception. Since Microchip language and other tools are constantly evolving to meet customer needs, some actual tool descriptions may differ from those in this document. Please refer to our web site to obtain the latest documentation available.

Documentation Numbering Conventions

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

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

RECOMMENDED READING

This document describes how to use MPLAB C30 compiler for dsPIC devices. For more information on MPLAB C30 and the use of other tools, the following are recommended reading.

README Files

For the latest information on Microchip tools, read the associated README files (ASCII text files) included with the software.

dsPIC Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools (MPLAB ASM30, MPLAB LINK30 and MPLAB C30) for dsPIC digital signal controllers (DSC's). Examples using the dsPIC simulator, MPLAB SIM30, are provided.

MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide (DS51317)

A guide to using the dsPIC DSC assembler, MPLAB ASM30, dsPIC DSC linker, MPLAB LINK30 and various dsPIC DSC utilities, including MPLAB LIB30 archiver/librarian.

GNU HTML Documentation

This documentation is provided on the language tool CD-ROM. It describes the standard GNU development tools, upon which MPLAB C30 is based.

dsPIC30F Enhanced Flash 16-Bit Digital Signal Controllers General Purpose and Sensor Families Data Sheet (DS70083)

Data sheet for dsPIC30F digital signal controller (DSC). Gives an overview of the device and its architecture. Details memory organization, DSP operation and peripheral functionality. Includes electrical characteristics.

dsPIC30F Family Reference Manual (DS70046)

Family reference guide explains the operation of the dsPIC30F MCU family architecture and peripheral modules.

dsPIC30F Programmer's Reference Manual (DS70030)

Programmer's guide to dsPIC30F devices. Includes the programmer's model and instruction set.

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

Our web site (<http://www.microchip.com>) contains a wealth of documentation. Individual data sheets, application notes, tutorials and user's guides are all available for easy download. All documentation is in Adobe® Acrobat (pdf) format.

TROUBLESHOOTING

See the README files for information on common problems not addressed in this document.

THE MICROCHIP WEB SITE

Microchip provides online support on the Microchip World Wide Web (WWW) site. The website is used by Microchip as a means to make files and information easily available to customers. To view the site, you must have access to the Internet and a web browser, such as, Netscape® Navigator or Microsoft® Internet Explorer. The Microchip web site is available at: <http://www.microchip.com>.

The web site provides a variety of services. Users may download files for the latest development tools, data sheets, application notes, user's guides, articles, and sample programs. A variety of information specific to the business of Microchip is also available, including listings of Microchip sales offices, distributors and factory representatives.

Technical Support

- Frequently Asked Questions (FAQ)
- Online Discussion Groups – Conferences for products, Development Systems, technical information and more
- Microchip Consultant Program Member Listing
- Links to other useful web sites related to Microchip products

Developer's Toolbox

- Design Tips
- Device Errata

Other Available Information

- Latest Microchip Press Releases
- Listing of Seminars and Events
- Job Postings

DEVELOPMENT SYSTEMS CUSTOMER NOTIFICATION SERVICE

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

Go to the Microchip web site at (<http://www.microchip.com>) and click on Customer Change Notification. Follow the instructions to register.

The Development Systems product group categories are:

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

Here is a description of these categories:

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

Emulators – The latest information on Microchip in-circuit emulators. This includes the MPLAB ICE 2000 and MPLAB ICE 4000.

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

MPLAB – The latest information on Microchip MPLAB IDE, the Windows Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB SIM and MPLAB SIM30 simulators, MPLAB IDE Project Manager and general editing and debugging features.

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

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

Corporate Applications Engineers (CAEs) may be contacted at (480) 792-7627.

In addition, there is a Systems Information and Upgrade Line. This line provides system users a listing of the latest versions of all of Microchip's development systems software products. Plus, this line provides information on how customers can receive any currently available upgrade kits.

The Hotline Numbers are:

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

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

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The dsPIC® family of digital signal controllers (DSCs) combines the high performance required in DSP applications with standard microcontroller features needed for embedded applications.

The dsPIC DSCs 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 MPLAB ASM30 assembler and MPLAB LINK30 linker. The assembler and linker are discussed in detail in the *MPLAB ASM30, MPLAB LINK30 and Utilities User's Guide*, (DS51317).

1.2 HIGHLIGHTS

Items discussed in this chapter are:

- MPLAB C30 Description
- MPLAB C30 and Other Development Tools
- MPLAB C30 Feature Set

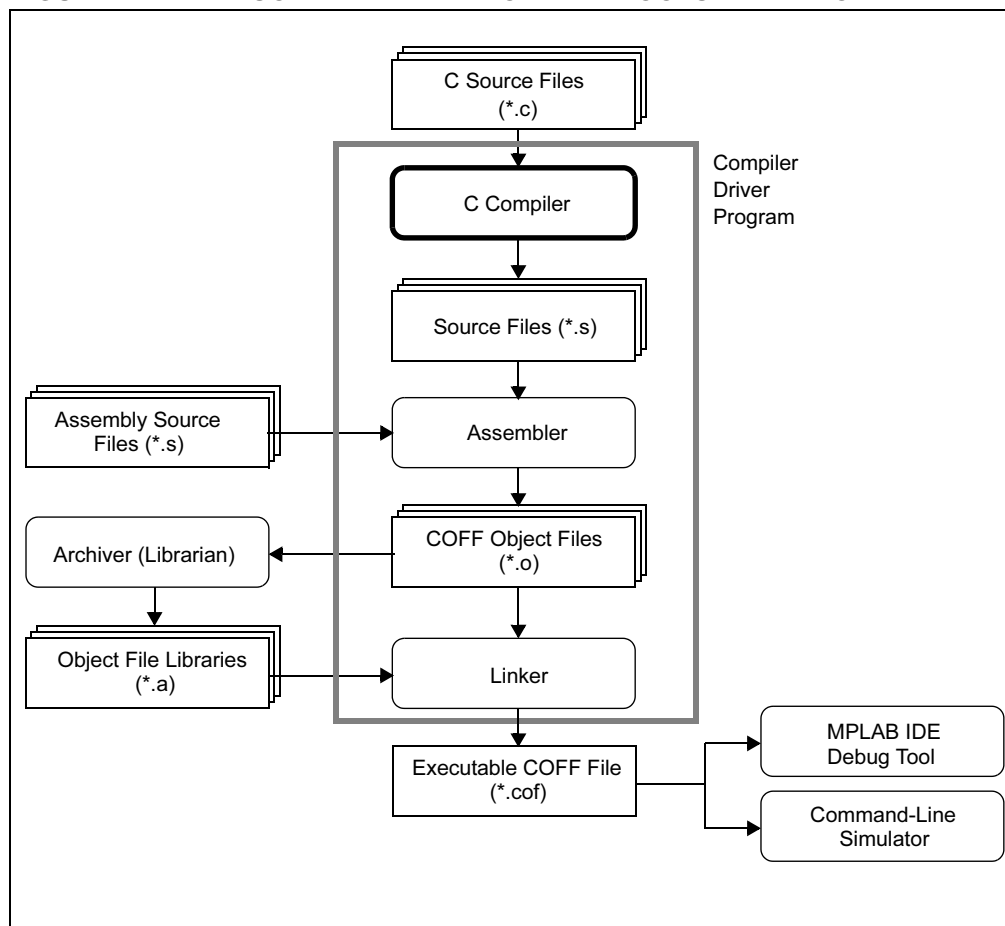
1.3 MPLAB C30 DESCRIPTION

MPLAB C30 is an ANSI x3.159-1989-compliant, optimizing C compiler that includes language extensions for dsPIC embedded-control applications. The compiler is a Windows® console application that provides a platform for developing C code. The compiler is a port of the GCC compiler from the Free Software Foundation.

1.4 MPLAB C30 AND OTHER DEVELOPMENT TOOLS

MPLAB C30 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 file format. The COFF 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 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 MPLAB C30 FEATURE SET

The MPLAB C30 C compiler is a full-featured, optimizing compiler that translates standard ANSI C programs into dsPIC assembly language source. The compiler also supports many command-line options and language extensions that allow full access to the dsPIC device hardware capabilities, and afford fine control of the compiler code generator. This section describes key features of the compiler.

1.5.1 ANSI C Standard

The MPLAB C30 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 dsPIC device-specific optimizations that take advantage of the particular features of the dsPIC device architecture.

1.5.3 ANSI Standard Library Support

MPLAB C30 is distributed with a complete ANSI C standard library. All library functions have been validated, and conform to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping, and math functions (trigonometric, exponential and hyperbolic). The standard I/O functions for file handling are also included, and, as distributed, they support full access to the host file system using the command-line simulator. The fully functional source code for the low-level file I/O functions is provided in the compiler distribution, and may be used as a starting point for applications that require this capability.

1.5.4 Flexible Memory Models

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

1.5.5 Compiler Driver

MPLAB C30 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).

NOTES:

Chapter 2. Differences Between MPLAB C30 and ANSI C

2.1 INTRODUCTION

This section discusses the differences between the C language supported by MPLAB C30 syntax and 1989 standard ANSI C.

2.2 HIGHLIGHTS

This section discusses:

- Keyword Differences
- Statement Differences

2.3 KEYWORD DIFFERENCES

This section describes the keyword differences between plain ANSI C and the C accepted by MPLAB C30. 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 MPLAB C30 port of GCC.

2.3.1 Specifying Attributes of Variables

The MPLAB C30 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:

- `aligned`
- `mode`
- `packed`
- `transparent_union`
- `unused`
- `section`
- `space`
- `near`
- `far`
- `sfr`
- `weak`
- `deprecated`

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)).
```

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 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 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 which is ever used for any data type on the target machine – which in the case of the dsPIC device, is two bytes (one word).

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` (see below).

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	MPLAB C30 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 MPLAB C30 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.

Differences Between MPLAB C30 and ANSI C

packed

The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment – one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows `a`:

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

Note: The dsPIC device requires that words be aligned on even byte boundaries, so care must be taken when using the `packed` attribute to avoid runtime addressing errors.

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.

unused

This attribute, attached to a variable, means that the variable is meant to be possibly unused. MPLAB C30 will not produce an unused variable warning for this variable.

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. This can be useful for explicitly allocating variables in the dsPIC X or Y data spaces.

```
struct array {
    int;
    struct array x __attribute__((section(".xdata")))={0};
    struct array y __attribute__((section(".ydata")))={0};
}
```

space (*space*)

Normally, the compiler allocates variables in data space. The `space` attribute can be used to direct the compiler to allocate a variable in program space. The `space` parameter can be `data`, to place the variable in data space (the default case), or `prog`, to place the variable in program space. Memory spaces are discussed further in **Section 4.6 “Memory Spaces”**.

Note: The MPLAB C30 compiler does not directly support accessing variables in program space. Variables so allocated must be explicitly accessed by the programmer, usually using table-access in-line assembly instructions, or using the Program Space Visibility window. See **Section 4.16 “Program Space Visibility (PSV) Usage”** for more information on the PSV window.

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));
```

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).

sfr (address)

The `sfr` attribute tells the compiler that the variable is allocated in near data space (the first 8 KB of data memory), and also specifies the runtime address of the variable, using the `address` parameter. Such variables can sometimes be accessed more efficiently than variable not allocated (or not known to be allocated) in near data space.

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

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

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.

Differences Between MPLAB C30 and ANSI C

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, 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.

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.

2.3.2 Specifying Attributes of Functions

In MPLAB C30, 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:

- `noreturn`
- `const`
- `format`
- `format_arg`
- `no_instrument_function`
- `unused`
- `weak`
- `alias`
- `section`
- `near`
- `far`
- `shadow`
- `interrupt`
- `deprecated`

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.

`noreturn`

A few standard library functions, such as `abort` and `exit`, cannot return. MPLAB C30 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`.

Differences Between MPLAB C30 and ANSI C

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 sub expression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `const`. For example:

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

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

Note that a function that has pointer arguments and examines the data pointed 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.

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 (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 MPLAB C30 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 MPLAB C30 can check the calls to `printf`, `scanf` or `strftime` function whose operands are a call to one of your own function.

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.

unused

This attribute, attached to a function, means that the function is meant to be possibly unused. MPLAB C30 will not produce an unused function warning for this function.

weak

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

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.

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.

near

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

far

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

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)
```


Differences Between MPLAB C30 and ANSI C

```
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 `pre prologue` specifies assembly code that is to be emitted before the compiler-generated `prologue` code. See **Chapter 7. "Interrupts"** for a full description, including examples.

deprecated

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

2.3.3 Inline Functions

By declaring a function `inline`, you can direct MPLAB C30 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.

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 MPLAB C30 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, MPLAB C30 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

MPLAB C30 allows you to put a few global variables into specified hardware 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 8. "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).

2.3.4.1 DEFINING GLOBAL REGISTER VARIABLES

You can define a global register variable in MPLAB C30 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.

Differences Between MPLAB C30 and ANSI C

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 which 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 which 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.

2.3.4.2 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 guarantee that MPLAB C30 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.

2.3.5 Complex Numbers

MPLAB C30 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.

MPLAB C30 can allocate complex automatic variables in a non contiguous 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 MPLAB C30 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

MPLAB C30 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 MPLAB C30.

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`.

Differences Between MPLAB C30 and ANSI C

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 MPLAB C30. 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 MPLAB C30 port of GCC.

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];
```

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

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 BUILT-IN RETURN ADDRESS FUNCTION

```
__builtin_return_address (level)
```

This function returns the return address of the current function, or of one of its callers. The `level` argument is number of frames to scan up the call stack. 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.

The `level` argument must be a constant integer. 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.

Chapter 3. Using MPLAB C30 C Compiler

3.1 INTRODUCTION

This chapter discusses using the MPLAB C30 C compiler on the command line. For information on using MPLAB C30 with MPLAB® IDE, please refer to the *dsPIC® Language Tools Getting Started* (DS70094).

3.2 HIGHLIGHTS

Items discussed in this chapter are:

- Overview
- File Naming Conventions
- Options
- Environment Variables
- Compiling a Single File on the Command Line
- Compiling Multiple Files on the Command Line

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 GNU toolset. A few are specific to the MPLAB C30 compiler.

The basic form of the compiler command line is:

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

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

The available options are described in **Section 3.5 “Options”**.

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

```
pic30-gcc -o hello.cof 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	Extensions
<i>file.c</i>	A C source file that must be preprocessed.	<i>file.c</i>
<i>file.h</i>	A header file (not to be compiled or linked).	<i>file.h</i>
<i>file.i</i>	A C source file that should not be preprocessed.	<i>file.i</i>
<i>file.o</i>	An object file.	<i>file.o</i>
<i>file.p</i>	A pre procedural-abstraction assembly language file.	<i>file.p</i>
<i>file.s</i>	Assembler code.	<i>file.s</i>
<i>file.S</i>	Assembler code that must be preprocessed.	<i>file.S</i>
other	A file to be passed to the linker.	other

3.5 OPTIONS

MPLAB C30 has many options for controlling compilation, all of which are case-sensitive.

- Options Specific to dsPIC 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 dsPIC Devices

For more information on the memory models, see **Section 4.7 “Memory Models”**.

TABLE 3-2: dsPIC DEVICE-SPECIFIC OPTIONS

Option	Definition
<code>-mconst-in-code</code>	Put constants in the program memory space. The compiler will access these constants using the PSV window. (This is the default.)
<code>-mconst-in-data</code>	Put constants in the data memory space.
<code>-mlarge-code</code>	Compile using the large code model. No assumptions are made about the locality of called functions.
<code>-mlarge-data</code>	Compile using the large data model. No assumptions are made about the location of static and external variables.
<code>-mpa⁽¹⁾</code>	Enable the procedural abstraction optimization. There is no limit on the nesting level.
<code>-mpa=<i>n</i>⁽¹⁾</code>	Enable the procedural abstraction optimization up to level <i>n</i> . If <i>n</i> is zero, the optimization is disabled. If <i>n</i> is 1, first level of abstraction is allowed; that is, instruction sequences in the source code may be abstracted into a subroutine. If <i>n</i> is 2, a second level of abstraction is allowed; that is, instructions that were put into a subroutine in the first level may be abstracted into a subroutine one level deeper. This pattern continues for larger values of <i>n</i> . The net effect is to limit the subroutine call nesting depth to a maximum of <i>n</i> .
<code>-mno-pa⁽¹⁾</code>	Do not enable the procedural-abstraction optimization. (This is the default.)
<code>-msmall-code</code>	Compile using the small code model. Called functions are assumed to be proximate (within 32 Kwords of the caller). (This is the default.)
<code>-msmall-data</code>	Compile using the small data model. All static and external variables are assumed to be located in the lower 8 KB of data memory space. (This is the default.)
<code>-msmall-scalar</code>	Like <code>-msmall-data</code> , except that only static and external scalars are assumed to be in the lower 8 KB of data memory space. (This is the default.)
<code>-mtext=<i>name</i></code>	Specifying <code>-mtext=<i>name</i></code> will cause text to be placed in a section named <i>name</i> rather than the default <code>.text</code> section. No white spaces should appear around the <code>=</code> .
<code>-msmart-io</code> <code>[=0 1 2]</code>	This option attempts to statically analyze format strings passed to <code>printf</code> , <code>scanf</code> , and the ‘f’ and ‘v’ variations of these functions. Uses of non-floating point format arguments will be converted to use an integer-only variation of the call. <code>-msmart-io=0</code> disables this option, while <code>-msmart-io=2</code> causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. <code>-msmart-io=1</code> is the default and will only convert the literal values it can prove.

Note 1: The procedural 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` are harder to debug; it is not recommended that this option is used while debugging. The procedural abstractor is invoked as a separate phase of compilation, after the production of an assembly file. This phase does not optimize across translation units.

3.5.2 Options for Controlling the Kind of Output

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 <i>file</i>	Place the output in <i>file</i> .
-S	Stop after compilation proper, i.e., before invoking the assembler. The default file extension is .s.
-v	Print the commands executed during each stage of compilation.
-x	<p>You can specify the input language explicitly with the -x option:</p> <p>-x language</p> <p>Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next -x option. The following values are supported by MPLAB C30:</p> <pre>c c-header cpp-output assembler assembler-with-cpp</pre> <p>-x none</p> <p>Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another -x option has been used. For example:</p> <pre>pic30-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s</pre> <p>Without the -x none, the compiler will assume all the input files are for the assembler.</p>
--help	Print a description of the command line options.

3.5.3 Options for Controlling the C Dialect

TABLE 3-4: C DIALECT CONTROL OPTIONS

Option	Definition
-ansi	Support all 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 bitfield is signed or unsigned, when the declaration does not use either signed or unsigned. By default, such a bitfield is signed, unless -traditional is used, in which case bitfields are always unsigned.
-funsigned-char	Let the type char be unsigned, like unsigned char.
-fwritable-strings	Store strings in the writable data segment and don't make them unique.

3.5.4 Options for Controlling Warnings and Errors

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which 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 MPLAB C30 C Compiler.

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

Option	Definition
<code>-fsyntax-only</code>	Check the code for syntax, but don't do anything beyond that.
<code>-pedantic</code>	Issue all the warnings demanded by strict ANSI C; reject all programs that use forbidden extensions.
<code>-pedantic-errors</code>	Like <code>-pedantic</code> , except that errors are produced rather than warnings.
<code>-w</code>	Inhibit all warning messages.
<code>-Wall</code>	All of the <code>-w</code> options listed in this table combined. This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
<code>-Wchar-subscripts</code>	Warn if an array subscript has type <code>char</code> .
<code>-Wcomment</code> <code>-Wcomments</code>	Warn whenever a comment-start sequence <code>/*</code> appears in a <code>/*</code> comment, or whenever a Backslash-Newline appears in a <code>//</code> comment.
<code>-Wdiv-by-zero</code>	Warn about compile-time integer division by zero. To inhibit the warning messages, use <code>-Wno-div-by-zero</code> . Floating point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs. (This is the default.)
<code>-Werror-implicit-function-declaration</code>	Give an error whenever a function is used before being declared.
<code>-Wformat</code>	Check calls to <code>printf</code> and <code>scanf</code> , etc., to make sure that the arguments supplied have types appropriate to the format string specified.
<code>-Wimplicit</code>	Equivalent to specifying both <code>-Wimplicit-int</code> and <code>-Wimplicit-function-declaration</code> .
<code>-Wimplicit-function-declaration</code>	Give a warning whenever a function is used before being declared.
<code>-Wimplicit-int</code>	Warn when a declaration does not specify a type.
<code>-Wmain</code>	Warn if the type of <code>main</code> is suspicious. <code>main</code> should be a function with external linkage, returning <code>int</code> , taking either zero, two, or three arguments of appropriate types.
<code>-Wmissing-braces</code>	Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for <code>a</code> is not fully bracketed, but that for <code>b</code> is fully bracketed. <pre>int a[2][2] = { 0, 1, 2, 3 }; int b[2][2] = { { 0, 1 }, { 2, 3 } };</pre>

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

Option	Definition
-Wmultichar -Wno-multichar	Warn if a multi-character <i>character</i> constant is used. Usually, such constants are typographical errors. Since they have implementation-defined values, they should not be used in portable code. The following example illustrates the use of a multi-character <i>character</i> constant: <pre>char xx(void) { return('xx'); }</pre>
-Wparentheses	Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often find confusing.
-Wreturn-type	Warn whenever a function is defined with a return-type that defaults to <i>int</i> . Also warn about any <i>return</i> statement with no return-value in a function whose return-type is not <i>void</i> .
-Wsequence-point	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. <p>The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a <i>&&</i>, <i> </i>, <i>?</i> : 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 sub expressions 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 have ruled that function calls do not overlap.</p> <p>It is not specified, when, between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that "Between the previous and next sequence point, an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored." If a program breaks these rules, the results on any particular implementation are entirely unpredictable.</p> <p>Examples of code with undefined behavior are <i>a = a++;</i>, <i>a[n] = b[n++]</i> and <i>a[i++] = 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.</p>

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

Option	Definition
-Wswitch	Warn whenever a <code>switch</code> statement has an index of enumerat type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) <code>case</code> labels outside the enumeration range also provoke warnings when this option is used.
-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 MPLAB C30 to emit warnings from system headers as if they occurred in user code. However, note that using <code>-Wall</code> in conjunction with this option will not warn about unknown pragmas in system headers; for that, <code>-Wunknown-pragmas</code> must also be used.
-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 <code>volatile</code> , or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.
-Wunknown-pragmas	Warn when a <code>#pragma</code> directive is encountered which is not understood by MPLAB C30. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the <code>-Wall</code> command line option.
-Wunused	Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used. In order to get a warning about an unused function parameter, both <code>-W</code> and <code>-Wunused</code> must be specified. Casting an expression to void suppresses this warning for an expression. Similarly, the <code>unused</code> attribute suppresses this warning for unused variables, parameters and labels.
-Wunused-function	Warn whenever a static function is declared but not defined or a non-inline static function is unused.
-Wunused-label	Warn whenever a label is declared but not used. To suppress this warning use the <code>unused</code> attribute (see Section 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	<p>Print extra warning messages for these events:</p> <ul style="list-style-type: none"> • A nonvolatile automatic variable might be changed by a call to <code>longjmp</code>. These warnings are possible only in optimizing compilation. The compiler sees only the calls to <code>setjmp</code>. It cannot know where <code>longjmp</code> will be called; in fact, a signal handler could call it at any point in the code. As a result, a warning may be generated even when there is in fact no problem because <code>longjmp</code> cannot in fact be called at the place that would cause a problem. • A function could exit both via <code>return value</code>; and <code>return;</code>. Completing the function body without passing any return statement is treated as <code>return;</code>. • An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as <code>x[i, j]</code> will cause a warning, but <code>x[(void) i, j]</code> will not. • An unsigned value is compared against zero with <code><</code> or <code><=</code>. • A comparison like <code>x<=y<=z</code> appears; this is equivalent to <code>(x<=y ? 1 : 0) <= z</code>, which is a different interpretation from that of ordinary mathematical notation. • Storage-class specifiers like <code>static</code> are not the first things in a declaration. According to the C Standard, this usage is obsolescent. • If -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.)

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

Option	Definition
	<ul style="list-style-type: none"> An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for <code>x.h</code>: <pre>struct s { int f, g; }; struct t { struct s h; int i; }; struct t x = { 1, 2, 3 };</pre> An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because <code>x.h</code> would be implicitly initialized to zero: <pre>struct s { int f, g, h; }; struct s x = { 3, 4 };</pre>
<code>-Waggregate-return</code>	Warn if any functions that return structures or unions are defined or called.
<code>-Wbad-function-cast</code>	Warn whenever a function call is cast to a non-matching type. For example, warn if <code>int foof()</code> is cast to anything <code>*</code> .
<code>-Wcast-align</code>	Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a <code>char *</code> is cast to an <code>int *</code> on machines where integers can only be accessed at two- or four-byte boundaries.
<code>-Wcast-qual</code>	Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a <code>const char *</code> is cast to an ordinary <code>char *</code> .
<code>-Wconversion</code>	Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment <code>x = -1</code> if <code>x</code> is unsigned. But do not warn about explicit casts like <code>(unsigned) -1</code> .
<code>-Werror</code>	Make all warnings into errors.
<code>-Winline</code>	Warn if a function can not be inlined, and either it was declared as inline, or else the <code>-finline-functions</code> option was given.
<code>-Wlarger-than-len</code>	Warn whenever an object of larger than <code>len</code> bytes is defined.
<code>-Wlong-long</code> <code>-Wno-long-long</code>	Warn if <code>long long</code> type is used. This is default. To inhibit the warning messages, use <code>-Wno-long-long</code> . Flags <code>-Wlong-long</code> and <code>-Wno-long-long</code> are taken into account only when <code>-pedantic</code> flag is used.
<code>-Wmissing-declarations</code>	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.
<code>-Wmissing-format-attribute</code>	If <code>-Wformat</code> is enabled, also warn about functions which might be candidates for format attributes. Note these are only possible candidates, not absolute ones. This option has no effect unless <code>-Wformat</code> is enabled.

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

Option	Definition
-Wmissing-noreturn	Warn about functions that might be candidates for attribute <code>noreturn</code> . These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. Actually, do not ever return before adding the <code>noreturn</code> attribute; otherwise subtle code generation bugs could be introduced.
-Wmissing-prototypes	Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. (This option can be used to detect global functions that are not declared in header files.)
-Wnested-externs	Warn if an <code>extern</code> declaration is encountered within a function.
-Wno-deprecated-declarations	Do not warn about uses of functions, variables, and types marked as deprecated by using the <code>deprecated</code> attribute.
-Wpadded	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.
-Wpointer-arith	Warn about anything that depends on the size of a function type or of <code>void</code> . MPLAB C30 assigns these types a size of 1, for convenience in calculations with <code>void *</code> pointers and pointers to functions.
-Wredundant-decls	Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.
-Wshadow	Warn whenever a local variable shadows another local variable.
-Wsign-compare -Wno-sign-compare	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by <code>-W</code> ; to get the other warnings of <code>-W</code> without this warning, use <code>-W -Wno-sign-compare</code> .
-Wstrict-prototypes	Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)
-Wtraditional	Warn about certain constructs that behave differently in traditional and ANSI C. <ul style="list-style-type: none"> Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C. A function declared external in one block and then used after the end of the block. A switch statement has an operand of type <code>long</code>. A nonstatic function declaration follows a static one. This construct is not accepted by some traditional C compilers.
-Wundef	Warn if an undefined identifier is evaluated in an <code>#if</code> directive.
-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.

TABLE 3-6: WARNING/ERROR OPTIONS NOT IMPLIED BY -Wall (CONTINUED)

Option	Definition
-Wwrite-strings	Give string constants the type <code>const char[length]</code> so that copying the address of one into a non- <code>const char *</code> pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using <code>const</code> in declarations and prototypes. Otherwise, it will just be a nuisance; which is why <code>-Wall</code> does not request these warnings.

3.5.5 Options for Debugging

TABLE 3-7: DEBUGGING OPTIONS

Option	Definition
-g	Produce debugging information. <ul style="list-style-type: none">• MPLAB C30 supports the use of <code>-g</code> with <code>-O</code>, making it possible to debug optimized code. The shortcuts taken by optimized code may occasionally produce surprising results:• 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 <code>'foo.c'</code> with <code>'-c -save-temps'</code> would produce the following files: <code>'foo.i'</code> (preprocessed file) <code>'foo.p'</code> (pre procedural-abstraction assembly language file) <code>'foo.s'</code> (assembly language file) <code>'foo.o'</code> (object file)

3.5.6 Options for Controlling Optimization

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 <code>register</code> 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 <code>-fthread-jumps</code> and <code>-fdefer-pop</code> . The compiler turns on <code>-fomit-frame-pointer</code> .
-O2	Optimize even more. MPLAB C30 performs nearly all supported optimizations that do not involve a space-speed trade-off. -O2 turns on all optional optimizations except for loop unrolling (<code>-funroll-loops</code>), function inlining (<code>-finline-functions</code>), and strict aliasing optimizations (<code>-fstrict-aliasing</code>). It also turns on force copy of memory operands (<code>-fforce-mem</code>) and frame pointer elimination (<code>-fomit-frame-pointer</code>). As compared to -O, this option increases both compilation time and the performance of the generated code.
-O3	Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on the <code>inline-functions</code> option.
-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
-falign-functions -falign-functions= <i>n</i>	Align the start of functions to the next power-of-two greater than <i>n</i> , skipping up to <i>n</i> bytes. For instance, -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 <i>n</i> is a power of two; so <i>n</i> is rounded up. If <i>n</i> is not specified, use a machine-dependent default.
-falign-labels -falign-labels= <i>n</i>	Align all branch targets to a power-of-two boundary, skipping up to <i>n</i> 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 <i>n</i> is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.
-falign-loops -falign-loops= <i>n</i>	Align loops to a power-of-two boundary, skipping up to <i>n</i> 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 <i>n</i> is not specified, use a machine-dependent default.
-fcallee-saves	Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.
-fcse-follow-jumps	In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an if statement with an else clause, CSE will follow the jump when the condition tested is false.
-fcse-skip-blocks	This is similar to -fcse-follow-jumps, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if.
-fexpensive-optimizations	Perform a number of minor optimizations that are relatively expensive.
-ffunction-sections -fdata-sections	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file. Only use these options when there are significant benefits from doing so. When you specify these options, the assembler and linker may create larger object and executable files and will also be slower.
-fgcse	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
-fgcse-lm	When -fgcse-lm is enabled, global common subexpression elimination will attempt to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.
-fgcse-sm	When -fgcse-sm is enabled, a store motion pass is run after global common subexpression elimination. This pass will attempt to move stores out of loops. When used in conjunction with -fgcse-lm, loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.
-fmove-all-movables	Forces all invariant computations in loops to be moved outside the loop.
-fno-defer-pop	Always pop the arguments to each function call as soon as that function returns. The compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.
-fno-peephole -fno-peephole2	Disable any machine-specific peephole optimizations. The difference between -fno-peephole and -fno-peephole2 is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.
-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.
-freduce-all-givs	Forces all general-induction variables in loops to be strength-reduced. These options may generate better or worse code; results are highly dependent on the structure of loops within the source code.
-frename-registers	Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization will most benefit processors with lots of registers. It can, however, make debugging impossible, since variables will no longer stay in a "home register".
-frerun-cse-after-loop	Rerun common subexpression elimination after loop optimizations has been performed.
-frerun-loop-opt	Run the loop optimizer twice.
-fschedule-insns	Attempt to reorder instructions to eliminate execution stalls due to required data being unavailable.
-fschedule-insns2	Similar to -fschedule-insns, but requests an additional pass of instruction scheduling after register allocation has been done.
-fstrength-reduce	Perform the optimizations of loop strength reduction and elimination of iteration variables.

TABLE 3-9: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
-fstrict-aliasing	<p>Allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C, this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an <code>unsigned int</code> can alias an <code>int</code>, but not a <code>void*</code> or a <code>double</code>. A character type may alias any other type.</p> <p>Pay special attention to code like this:</p> <pre>union a_union { int i; double d; }; int f() { union a_union t; t.d = 3.0; return t.i; }</pre> <p>The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with <code>-fstrict-aliasing</code>, type-punning is allowed, provided the memory is accessed through the union type. So, the code above will work as expected. However, this code might not:</p> <pre>int f() { a_union t; int* ip; t.d = 3.0; ip = &t.i; return *ip; }</pre>
-fthread-jumps	<p>Perform optimizations where a check is made to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.</p>
-funroll-loops	<p>Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or runtime. <code>-funroll-loops</code> implies both <code>-fstrength-reduce</code> and <code>-frerun-cse-after-loop</code>.</p>
-funroll-all-loops	<p>Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. <code>-funroll-all-loops</code> implies <code>-fstrength-reduce</code> as well as <code>-frerun-cse-after-loop</code>.</p>

Options of the form `-fflag` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. In the table below, only one of the forms is listed (the one that is not the default.)

TABLE 3-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
<code>-fforce-mem</code>	Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common sub expressions. When they are not common sub expressions, instruction combination should eliminate the separate register-load. The <code>-O2</code> option turns on this option.
<code>-finline-functions</code>	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared <code>static</code> , and then the function is normally not output as assembler code in its own right.
<code>-finline-limit=n</code>	<p>By default, MPLAB C30 limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (i.e., marked with the <code>inline</code> keyword). <code>n</code> is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of <code>n</code> is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption.</p> <p>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.</p> <p>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 to another.</p>
<code>-fkeep-inline-functions</code>	Even if all calls to a given function are integrated, and the function is declared <code>static</code> , output a separate runtime callable version of the function. This switch does not affect <code>extern</code> inline functions.
<code>-fkeep-static-consts</code>	Emit variables declared <code>static const</code> when optimization isn't turned on, even if the variables aren't referenced. MPLAB C30 enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the <code>-fno-keep-static-consts</code> option.
<code>-fno-function-cse</code>	<p>Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.</p> <p>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.</p>

TABLE 3-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
-fno-inline	Do not pay attention to the <code>inline</code> keyword. Normally this option is used to keep the compiler from expanding any functions inline. If optimization is not enabled, no functions can be expanded inline.
-fomit-frame-pointer	Do not keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions.
-foptimize-sibling-calls	Optimize sibling and tail recursive calls.

3.5.7 Options for Controlling the Preprocessor

TABLE 3-11: PREPROCESSOR OPTIONS

Option	Definition
-Aquestion (answer)	Assert the answer <i>answer</i> for question <i>question</i> , in case it is tested with a preprocessing conditional such as <code>#if #question(answer)</code> . -A- disables the standard assertions that normally describe the target machine. For example, the function prototype for main might be declared as follows: <pre>#if #environ(freestanding) int main(void); #else int main(int argc, char *argv[]); #endif</pre> A -A command-line option could then be used to select between the two prototypes. For example, to select the first of the two, the following command-line option could be used: -Aenviron(freestanding)
-A -predicate =answer	Cancel an assertion with the predicate <i>predicate</i> and answer <i>answer</i> .
-A predicate =answer	Make an assertion with the predicate <i>predicate</i> and answer <i>asnsver</i> . This form is preferred to the older form <code>-A predicate(answer)</code> , which is still supported, because it does not use shell special characters.
-C	Tell the preprocessor not to discard comments. Used with the -E option.
-dD	Tell the preprocessor to not remove macro definitions into the output, in their proper sequence.
-Dmacro	Define macro <i>macro</i> with the string 1 as its definition.
-Dmacro=defn	Define macro <i>macro</i> as <i>defn</i> . All instances of -D on the command line are processed before any -U options.
-dM	Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the -E option.
-dN	Like -dD except that the macro arguments and contents are omitted. Only <code>#define name</code> 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.

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-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- option 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.
-Idir	Add the directory <i>dir</i> to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one -I option, the directories are scanned in left-to-right order; the standard system directories come after.
-idirafter dir	Add the directory <i>dir</i> to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the 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 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 <i>prefix</i> as the prefix for subsequent -iwithprefix options.
-isystem dir	Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.
-iwithprefix dir	Add a directory to the second include path. The directory's name is made by concatenating prefix and <i>dir</i> , where prefix was specified previously with -iprefix. If a prefix has not yet been specified, the directory containing the installed passes of the compiler is used as the default.

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
<code>-iwithprefixbefore dir</code>	Add a directory to the main include path. The directory's name is made by concatenating prefix and <i>dir</i> , as in the case of <code>-iwithprefix</code> .
<code>-M</code>	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 <code>#include</code> header files it uses. This rule may be a single line or may be continued with <code>\-newline</code> if it is long. The list of rules is printed on standard output instead of the preprocessed C program. <code>-M</code> implies <code>-E</code> (see Section 3.5.2 “Options for Controlling the Kind of Output”).
<code>-MD</code>	Like <code>-M</code> but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a <code>.d</code> extension.
<code>-MF file</code>	When used with <code>-M</code> or <code>-MM</code> , specifies a file to which to write the dependencies. If no <code>-MF</code> switch is given the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options <code>-MD</code> or <code>-MMD</code> , <code>-MF</code> overrides the default dependency output file.
<code>-MG</code>	Treat missing header files as generated files and assume they live in the same directory as the source file. If <code>-MG</code> is specified, then either <code>-M</code> or <code>-MM</code> must also be specified. <code>-MG</code> is not supported with <code>-MD</code> or <code>-MMD</code> .
<code>-MM</code>	Like <code>-M</code> but the output mentions only the user header files included with <code>#include "file"</code> . System header files included with <code>#include <file></code> are omitted.
<code>-MMD</code>	Like <code>-MD</code> except mention only user header files, not system header files.
<code>-MP</code>	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 Makefile to match. This is typical output: test.o: test.c test.h test.h:
<code>-MQ</code>	Same as <code>-MT</code> , but it quotes any characters which are special to Make. <code>-MQ '\$(objpfx)foo.o'</code> gives <code>\$(objpfx)foo.o:</code> <code>foo.c</code> The default target is automatically quoted, as if it were given with <code>-MQ</code> .

TABLE 3-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
<code>-MT <i>target</i></code>	<p>Change the target of the rule emitted by dependency generation. By default CPP takes the name of the main input file, including any path, deletes any file suffix such as <code>.c</code>, and appends the platform's usual object suffix. The result is the target.</p> <p>An <code>-MT</code> 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 <code>-MT</code>, or use multiple <code>-MT</code> options. For example:</p> <p><code>-MT '\$(objpfx)foo.o'</code> might give <code>\$(objpfx)foo.o:foo.c</code></p>
<code>-nostdinc</code>	<p>Do not search the standard system directories for header files. Only the directories you have specified with <code>-I</code> options (and the current directory, if appropriate) are searched. (See Section 3.5.10 "Options for Directory Search") for information on <code>-I</code>.</p> <p>By using both <code>-nostdinc</code> and <code>-I-</code>, the include-file search path can be limited to only those directories explicitly specified.</p>
<code>-P</code>	Tell the preprocessor not to generate <code>#line</code> directives. Used with the <code>-E</code> option (see Section 3.5.2 "Options for Controlling the Kind of Output").
<code>-trigraphs</code>	Support ANSI C trigraphs. The <code>-ansi</code> option also has this effect.
<code>-U<i>macro</i></code>	Undefine macro <i>macro</i> . <code>-U</code> options are evaluated after all <code>-D</code> options, but before any <code>-include</code> and <code>-imacros</code> options.
<code>-undef</code>	Do not predefine any nonstandard macros (including architecture flags).

3.5.8 Options for Assembling

TABLE 3-12: ASSEMBLY OPTIONS

Option	Definition
<code>-Wa,<i>option</i></code>	Pass <i>option</i> as an option to the assembler. If <i>option</i> contains commas, it is split into multiple options at the commas.

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
<code>-Ldir</code>	Add directory <i>dir</i> to the list of directories to be searched for libraries specified by the command-line option <code>-l</code> .
<code>-llibrary</code>	<p>Search the library named <i>library</i> when linking.</p> <p>The linker searches a standard list of directories for the library, which is actually a file named <code>liblibrary.a</code>. The linker then uses this file as if it had been specified precisely by name.</p> <p>It makes a difference where in the command you write this option; the linker processes libraries and object files in the order they are specified. Thus, <code>foo.o -lz bar.o</code> searches library <code>z</code> after file <code>foo.o</code> but before <code>bar.o</code>. If <code>bar.o</code> refers to functions in <code>libz.a</code>, those functions may not be loaded.</p> <p>The directories searched include several standard system directories plus any that you specify with <code>-L</code>.</p> <p>Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an <code>-l</code> option (e.g., <code>-lmylib</code>) and specifying a file name (e.g., <code>libmylib.a</code>) is that <code>-l</code> searches several directories, as specified.</p> <p>By default the linker is directed to search: <code><install-path>\lib</code> for libraries specified with the <code>-l</code> option. For a compiler installed into the default location, this would be: <code>c:\pic30_tools\lib</code> This behavior can be overridden using the environment variables defined in Section 3.6 “Environment Variables”.</p>
<code>-nodefaultlibs</code>	<p>Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The compiler may generate calls to <code>memcpy</code>, <code>memset</code> and <code>memcpy</code>. These entries are usually resolved by entries in the standard compiler libraries.</p> <p>These entry points should be supplied through some other mechanism when this option is specified.</p>
<code>-nostdlib</code>	<p>Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify will be passed to the linker. The compiler may generate calls to <code>memcpy</code>, <code>memset</code>, and <code>memcpy</code>. These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.</p>
<code>-s</code>	Remove all symbol table and relocation information from the executable.
<code>-u symbol</code>	Pretend <i>symbol</i> is undefined to force linking of library modules to define the symbol. It is legitimate to use <code>-u</code> multiple times with different symbols to force loading of additional library modules.
<code>-Wl, option</code>	Pass <i>option</i> as an option to the linker. If <i>option</i> contains commas, it is split into multiple options at the commas.

TABLE 3-13: LINKING OPTIONS (CONTINUED)

Option	Definition
<code>-Xlinker option</code>	Pass <i>option</i> as an option to the linker. You can use this to supply system-specific linker options that MPLAB C30 does not know how to recognize.

3.5.10 Options for Directory Search

TABLE 3-14: DIRECTORY SEARCH OPTIONS

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

3.5.11 Options for Code Generation Conventions

Options of the form `-fflag` specify machine-independent flags. Most flags have both positive and negative forms; the negative form of `-ffoo` would be `-fno-foo`. In the table below, only one of the forms is listed (the one that is not the default.)

TABLE 3-15: CODE GENERATION CONVENTION OPTIONS

Option	Definition
<code>-fargument-alias</code> <code>-fargument-noalias</code> <code>-fargument-noalias-global</code>	<p>Specify the possible relationships among parameters and between parameters and global data.</p> <p><code>-fargument-alias</code> specifies that arguments (parameters) may alias each other and may alias global storage.</p> <p><code>-fargument-noalias</code> specifies that arguments do not alias each other, but may alias global storage.</p> <p><code>-fargument-noalias-global</code> specifies that arguments do not alias each other and do not alias global storage.</p> <p>Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.</p>

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

Option	Definition
<code>-fcall-saved-reg</code>	<p>Treat the register named <i>reg</i> as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register <i>reg</i> if they use it.</p> <p>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.</p> <p>A different sort of disaster will result from the use of this flag for a register in which function values may be returned.</p> <p>This flag should be consistently through all modules.</p>
<code>-fcall-used-reg</code>	<p>Treat the register named <i>reg</i> as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register <i>reg</i>.</p> <p>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.</p> <p>This flag should be consistently through all modules.</p>
<code>-ffixed-reg</code>	<p>Treat the register named <i>reg</i> as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).</p> <p><i>reg</i> must be the name of a register, e.g., <code>-ffixed-w3</code>.</p>
<code>-finstrument-functions</code>	<p>Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site.</p> <pre>void __cyg_profile_func_enter (void *this_fn, void *call_site); void __cyg_profile_func_exit (void *this_fn, void *call_site);</pre> <p>The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table. The profiling functions should be provided by the user.</p> <p>Function instrumentation requires the use of a frame pointer. Some optimization levels disable the use of the frame pointer. Using <code>-fno-omit-frame-pointer</code> will prevent this.</p> <p>This instrumentation is also done for functions expanded inline in other functions. The profiling calls will indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use <code>extern inline</code> in your C code, an addressable version of such functions must be provided.</p> <p>A function may be given the attribute <code>no_instrument_function</code>, in which case this instrumentation will not be done.</p>
<code>-fno-ident</code>	Ignore the <code>#ident</code> directive.
<code>-fpack-struct</code>	Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code sub-optimal, and the offsets of structure members won't agree with system libraries.

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

Option	Definition
	The dsPIC device requires that words be aligned on even byte boundaries, so care must be taken when using the packed attribute to avoid runtime addressing errors.
-fpcc-struct-return	Return short <code>struct</code> and <code>union</code> values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing capability between MPLAB C30-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 <code>double</code> type equivalent to <code>float</code> . This option makes <code>double</code> equivalent to <code>long double</code> . Mixing this option across modules can have unexpected results if modules share double data either directly through argument passage or indirectly through shared buffer space. Libraries provided with the product function with either switch setting.
-fshort-enums	Allocate to an <code>enum</code> type only as many bytes as it needs for the declared range of possible values. Specifically, the <code>enum</code> type will be equivalent to the smallest integer type which has enough room.
-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 MPLAB C30. 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 MPLAB C30 searches for header files, it tries the directories listed in the variable, after the directories specified with <code>-I</code> but before the standard header file directories. If the environment variable is undefined, the preprocessor chooses an appropriate value based on the standard installation. By default, the following directories are searched for include files: <install-path>\include and <install-path>\support\h
PIC30_COMPILER_PATH	The value of PIC30_COMPILER_PATH is a semicolon-separated list of directories, much like PATH. MPLAB C30 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 MPLAB C30 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 <code>-B</code> 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.
TMPDIR	If TMPDIR is set, it specifies the directory to use for temporary files. MPLAB C30 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 COMPILING A SINGLE FILE ON THE COMMAND LINE

This section demonstrates how to compile and link a single file. For the purpose of this discussion it is assumed the compiler is installed on your `c:` drive in a directory called `pic30-tools`. Therefore the following will apply:

TABLE 3-17: COMPILER-RELATED DIRECTORIES

Directory	Contents
<code>c:\pic30_tools\include</code>	Include directory for ANSI C header file. This directory is where the compiler stores the standard C library system header files. The <code>PIC30_C_INCLUDE_PATH</code> environment variable can point to that directory. (From the DOS command prompt, type <code>set</code> to check this.)
<code>c:\pic30_tools\support\h</code>	Include directory for dsPIC device-specific header files. This directory is where the compiler stores the dsPIC device-specific header files. The <code>PIC30_C_INCLUDE_PATH</code> environment variable can point to that directory. (From the DOS command prompt, type <code>set</code> to check this.)
<code>c:\pic30_tools\lib</code>	Library directory: this directory is where the libraries and precompiled object files reside.
<code>c:\pic30_tools\support\gld</code>	Linker script directory: this directory is where device-specific linker script files may be found.
<code>c:\pic30_tools\bin</code>	Executables directory: this directory is where the compiler programs are located. Your <code>PATH</code> 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 6. "Device Support Files"**.

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

```
C:\> pic30-gcc -o ex1.cof ex1.c
```

The command-line option `-o ex1.cof` names the output COFF executable file (if the `-o` option is not specified, then the output file is named `a.out`). 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.cof
```

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

3.8 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 -o ex1.cof 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.cof` is created.

Chapter 4. MPLAB C30 C Compiler Runtime Environment

4.1 INTRODUCTION

This section discusses the MPLAB C30 C Compiler runtime environment.

4.2 HIGHLIGHTS

Items discussed in this chapter are:

- Address Spaces
- Code and Data Sections
- Startup and Initialization
- Memory Spaces
- Memory Models
- X and Y Data Spaces
- Locating Code and Data
- Software Stack
- The C Stack Usage
- The C Heap Usage
- Function Call Conventions
- Register Conventions
- Bit Reversed and Modulo Addressing
- PSV Usage

4.3 ADDRESS SPACES

The dsPIC® microcontroller (MCU) devices are a combination of traditional PICmicro® MCU features (peripherals, Harvard architecture, RISC) and new DSP capabilities. The dsPIC devices have two distinct memory regions:

- Program Memory (Figure 4-1) contains executable code and optionally constant data.
- Data Memory (Figure 4-2) 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.

FIGURE 4-1: PROGRAM SPACE MEMORY MAP

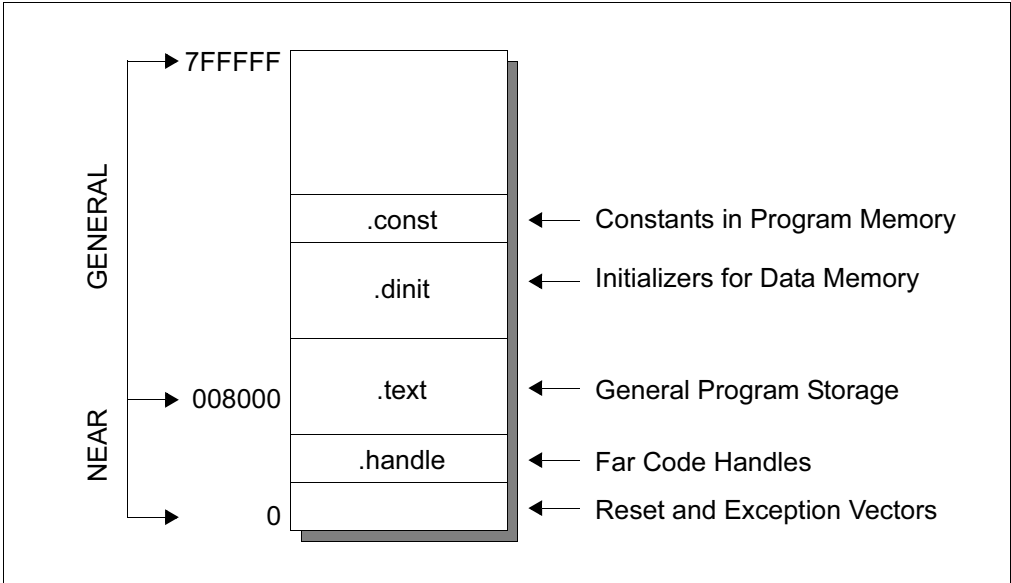
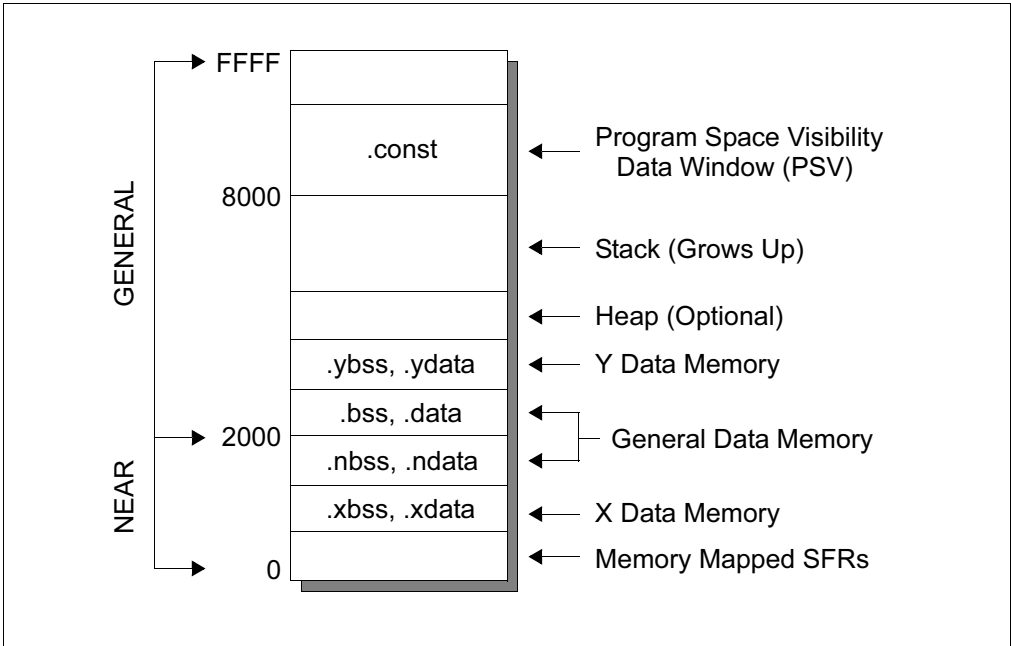


FIGURE 4-2: DATA SPACE MEMORY MAP



4.4 CODE AND DATA SECTIONS

A section is a locatable block of code or data that will occupy contiguous locations in the dsPIC device memory. In any given object file there are typically several sections. For example, a file may contain a section for program code and one for uninitialized data, among others.

The MPLAB C30 compiler will place code and data into default sections unless instructed otherwise through the use of section attributes (for information on the section attribute, see **Section 2.3 “Keyword Differences”**). While all compiler-generated executable code is allocated into a section named `.text`, data is allocated into different sections based on the type of data, as shown in Table 4-1.

TABLE 4-1: COMPILER-GENERATED DATA SECTIONS

	Initialized		Uninitialized	
	Variables	Constants in ROM	Constants in RAM	Variables
near	<code>.ndata</code>	<code>.const</code>	<code>.ndconst</code>	<code>.nbss</code>
far	<code>.data</code>	<code>.const</code>	<code>.dconst</code>	<code>.bss</code>

Each default section and a description of the type of information stored into that section is listed below.

`.text`

Executable code is allocated into the `.text` section.

`.data`

Initialized variables with the `far` attribute are allocated into the `.data` section. When the large data memory model is selected (i.e., when using the `-mlarge-data` command-line option), this is the default location for initialized variables.

`.ndata`

Initialized variables with the `near` attribute are allocated into the `.ndata` section. When the small data memory model is selected (i.e., when using the default `-msmall-data` command-line option), this is the default location for initialized variables.

`.const`

Constant values, such as string constants and `const`-qualified variables, are allocated into the `.const` section when using the default `-mconst-in-code` command-line option. This section is intended to be located in program memory and accessed using the PSV window.

`.dconst`

Constant values, such as string constants and `const`-qualified variables, are allocated into the `.dconst` section when using the default `-msmall-data` command-line option without using the `-mconst-in-code` command-line option. Unless the linker option `--no-data-init` is specified, the MPLAB C30 startup code will initialize this section by copying data from the `.dinit` section. The `.dinit` section is created by the linker and located in program memory.

`.ndconst`

Constant values, such as string constants and `const`-qualified variables, are allocated into the `.ndconst` section when using the `-msmall-data` command-line option without using the `-mconst-in-code` command-line option. The MPLAB C30 startup code will initialize this section by copying data from the `.dinit` section. The `.dinit` section is created by the linker and located in program memory.

.bss

Uninitialized variables with the `far` attribute are allocated into the `.bss` section. When the large data memory model is selected (i.e., when using the `-mlarge-data` command-line option), this is the default location for uninitialized variables.

.nbss

Uninitialized variables with the `near` attribute are allocated into the `.nbss` section. When the small data memory model is selected (i.e., when using the default `-msmall-data` command-line option), this is the default location for uninitialized variables.

.pbss - Persistent Data

Applications that require data storage in RAM which is not affected by a device reset can use section `.pbss` for this purpose. Section `.pbss` is allocated in near data memory and is not modified by the default startup module in `libpic30.a`.

Uninitialized variables may be placed in the `.pbss` section using the section attribute:

```
int i __attribute__((__section__(".pbss")));
```

To take advantage of persistent data storage, the `main()` function should begin with a test to determine what type of reset has occurred. Various bits in the RCON reset control register can be tested to determine the reset source. See Section 8 in the *dsPIC30F Family Reference Manual* (DS70046) for more information.

4.5 STARTUP AND INITIALIZATION

Two C runtime 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 4.10 “Software Stack”**.
2. If a `.const` section is defined, it is mapped into the Program Space Visibility (PSV) 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 MPLAB C30 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 4.4 “Code and Data Sections”**, as well as the user-defined sections.

Note: The persistent data section <code>.pbss</code> is never cleared or initialized.
--

4. The function `main` is called with no parameters.
5. 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 much smaller than the primary module, and can be selected to conserve program memory if data initialization is not required.

Source code (in dsPIC assembly language) for both modules is provided in the `c:\pic30_tools\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 switched to provide this support.

4.6 MEMORY SPACES

Static and external variables are normally allocated in the data memory space. If a variable is tagged with the code attribute, then it will be allocated in the program memory space.

For example, to allocate the integer variable `var1` in the `.text` section in program memory, use the following declaration:

```
int var1 __attribute__((__section__(".text"), __space__(__prog__)));
```

Note that variables allocated in the program space cannot be accessed directly by the C compiler. They must be accessed using either inline assembly or separate assembly language modules.

The compiler does support the use of `const`-qualified variables in program memory space by using the program space visibility (PSV) option of the dsPIC device processor. The compiler accesses such variables by using the PSV data window, that is, the upper 32 KB area of the data memory space. The compiler is commanded to place `const`-qualified variables in program memory using the default `-mconst-in-code` command line option. If this option is used, the C runtime startup code maps the `.const` section into the PSV data window before calling the main function.

4.7 MEMORY MODELS

The compiler supports several memory models. Command-line options are available for selecting the optimum memory model for your application, based on the specific dsPIC device part that you are using and the type of memory usage.

TABLE 4-2: MEMORY MODEL COMMAND LINE OPTIONS

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

The command-line options apply globally to the modules being compiled. Individual variables and functions can be declared as `near` or `far` 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”**.

4.7.1 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.

4.7.2 Near and Far Code

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

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

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

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.

4.8 X AND Y DATA SPACES

The C compiler does not directly support separating variables into X and Y data spaces. However, the section attribute can be used to explicitly locate variables in the X and Y spaces. For example, to locate an uninitialized variable in the `.ybss` section, use the following attribute:

```
__attribute__((__section__(".ybss"))) 
```

X and Y data sections are defined in linker scripts. X data sections (`.xbss`, `.xdata`) and Y data sections (`.ybss`, `.ydata`) are allocated into the appropriate memory range by the linker script.

The 'data' style sections are for user initialized values; the tool chain will allow data in this space to be initialized to some value. The 'bss' style sections cannot be initialized by the user; data in these sections will always be initialized to 0 at reset.

EXAMPLE 4-1: VALID USES OF THE SECTIONS

```
int status_flags[32] __attribute__((__section__(".ybss"))); /* = 0 */
char *message __attribute__((__section__(".ydata"))) = "Press SW1 to launch";
int buffer[32] __attribute__((__section__(".ydata"))); /* = 0 */
```

Variable `status_flags` will be initialized to 0 by the C Runtime Startup (CRT); the initial values for `message` and `buffer` will be copied into the appropriate space by the CRT. In this example, `buffer` is initialized inefficiently because it will require some program memory to hold the initial 0 values. `buffer` would be more efficiently placed in a 'bss' style section.

Invalid uses, all involving attempts to initialize variables placed in a 'bss' style variable, may not cause the compiler to complain but it will cause the 'bss' style sections to be converted to 'data' style sections. The net effect of this is that program memory usage may be greater than intended. The following examples illustrate what may happen.

EXAMPLE 4-2: MIXING A TRUE 'BSS' SECTION WITH A CO-ERCED SECTION

```
int status_flags[32] __attribute__((__section__(".ybss"))); /* = 0 */
char *message __attribute__((__section__(".ybss"))) = "Press SW1 to launch";
int buffer[32] __attribute__((__section__(".ydata"))); /* = 0 */
```

This example will cause the compiler to emit an error message:

```
file.c:line: status_flags causes a section type conflict
```

The compiler knows that `message` is initialized and that the name `".ybss"` is a 'data' style section (initialized variables are dealt with first). The `status_flags` definition will cause a section type conflict.

EXAMPLE 4-3: MIXING A TRUE 'BSS' SECTION WITH A CO-ERCED SECTION, IN SEPARATE FILES

```
file1.c:
int status_flags[32] __attribute__((__section__(".ybss"))); /* = 0 */
int buffer[32] __attribute__((__section__(".ybss"))); /* = 0 */

file2.c:
char *message __attribute__((__section__(".ybss"))) = "Press SW1 to launch";
```

`status_flags` would normally be placed in a 'bss' style section requiring three (3) program words to initialize the variable (no matter how large it is). However, the conflict in section type will be detected by the linker and fixed. `status_flags` will now require 25 program words by default and more if the `--no-pack-data` option is enabled.

In general it is best to keep 0 initialized data in 'bss' style sections and any initialized data in 'data' style sections.

4.9 LOCATING CODE AND DATA

As described in **Section 4.4 “Code and Data Sections”**, 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 to form the executable application program, the linker uses a linker script file that specifies the starting location of the various sections. The compiler distribution includes one linker script file for each device, and for most applications these device-specific linker script files will be suitable for locating sections on the part.

Cases may arise when a specific function or variable must be located at a specific address, or within some range of addresses. In this case, the function or variable must be placed in a user-defined section, and the starting address of the section must be specified. 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.

4.10 SOFTWARE STACK

The dsPIC 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 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 7. “Interrupts”** for details.

The C runtime startup module initializes the stack pointer (W15) and the stack pointer limit register (`SPLIM`) 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 ASM30*, *MPLAB LINK30* and *Utilities User's Guide* (DS51317) for details.

Alternatively, the stack of specific size may be allocated with a user-defined section in a custom linker script. In the following example, 0x100 bytes of data memory are reserved for the stack. Note that two symbols are declared, `__SP_init` and `__SPLIM_init`, for use by the C runtime startup module:

```
.stack :
{
    __SP_init = .;
    . += 0x100
    __SPLIM_init = .;
    . += 8
} >data
```

`__SP_init` defines the initial value for the stack pointer (W15) and `__SPLIM_init` defines the initial value for the stack pointer limit register (`SPLIM`). The value of `__SPLIM_INIT` should be at least 8 bytes less than the physical stack limit, to allow for stack error exception processing. This value should be decreased further to account for stack usage by the interrupt handler itself, if a stack error interrupt handler is installed. The default interrupt handler does not require additional stack usage.

4.11 THE C STACK USAGE

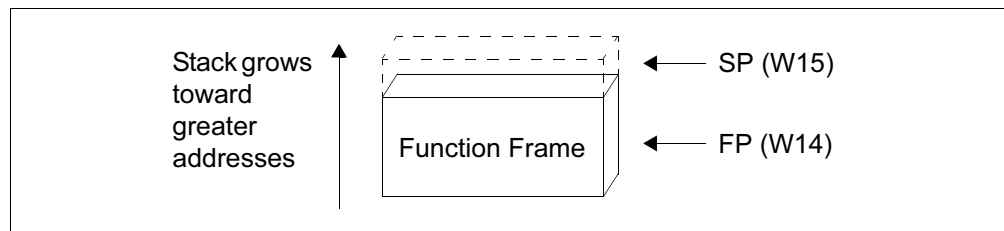
The C compiler uses the software stack to:

- 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 runtime 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.

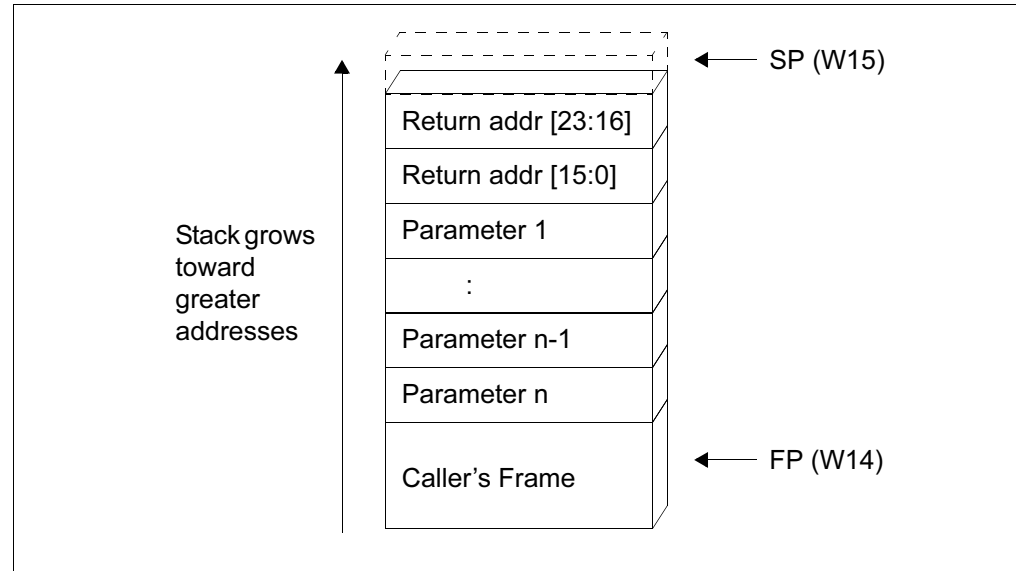
FIGURE 4-3: STACK AND FRAME POINTERS



The C runtime 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.

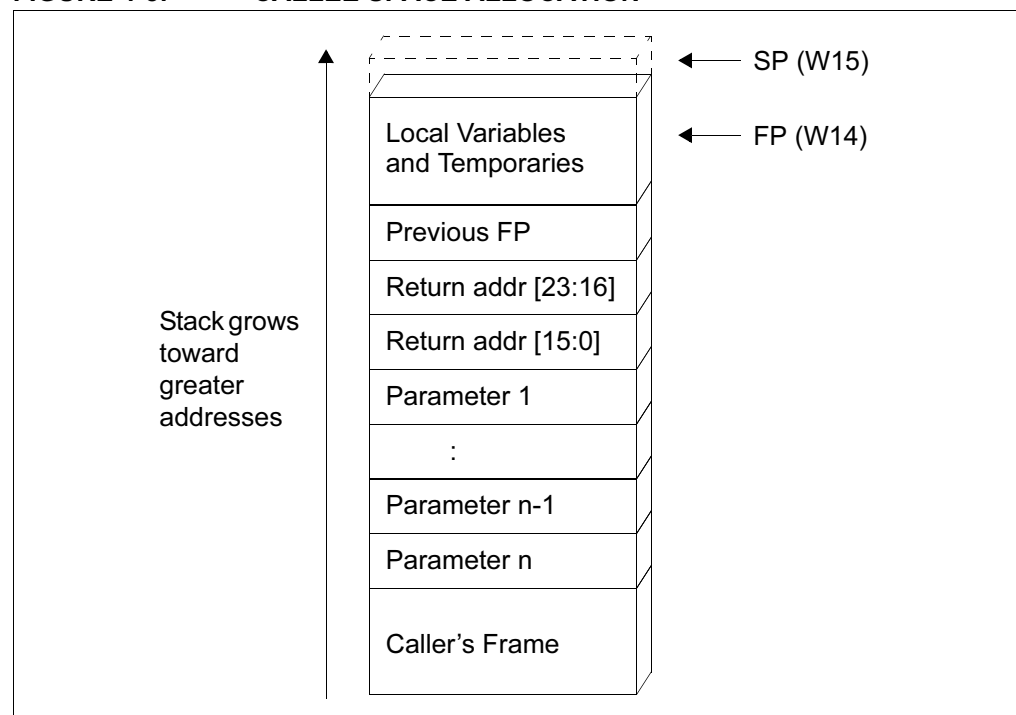
The following diagrams illustrate the steps involved in calling a function. Executing a `CALL` or `RCALL` instruction pushes the return address onto the software stack. See Figure 4-4.

FIGURE 4-4: CALL OR RCALL



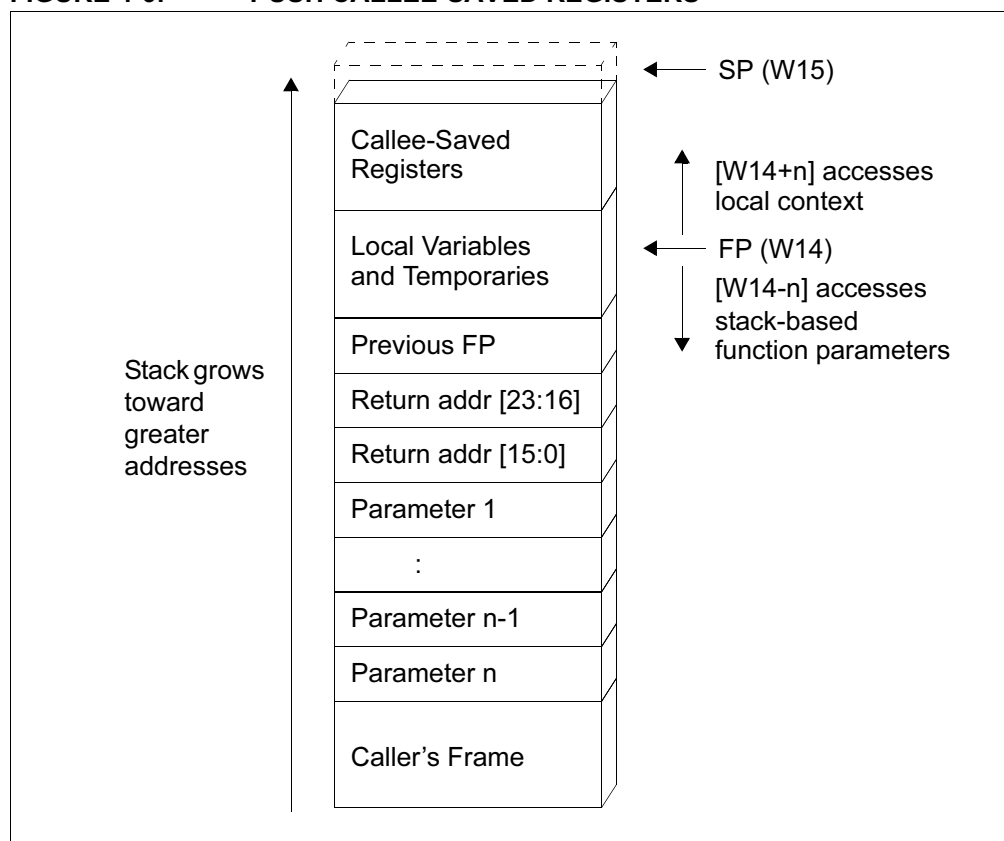
The called function (callee) can now allocate space for its local context (Figure 4-5).

FIGURE 4-5: CALLEE SPACE ALLOCATION



Finally, any callee-saved registers that are used in the function are pushed (Figure 4-6).

FIGURE 4-6: PUSH CALLEE-MAILED REGISTERS



4.12 THE C HEAP USAGE

The C runtime 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 (Figure 4-2).

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.

4.13 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.

TABLE 4-3: 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.

4.13.1 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 MPLAB C30 compiler to make the most efficient use of the available parameter registers.

EXAMPLE 4-4: 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 4-5: 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 4-6: 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 4-6) 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*.

4.13.2 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.

4.13.3 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.

4.14 REGISTER CONVENTIONS

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

TABLE 4-4: 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.
* double is equivalent to long double if -fno-short-double is used.	

4.15 BIT REVERSED AND MODULO ADDRESSING

The compiler does not 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.

4.16 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.

For more on PSV usage, see the *MPLAB ASM30*, *MPLAB LINK30* and *Utilities User's Guide*. (DS51317).

Chapter 5. Data Types

5.1 INTRODUCTION

This section discusses the MPLAB C30 data types.

5.2 HIGHLIGHTS

Items discussed in this chapter are:

- Data Representation
- Integer
- Floats
- 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	0x78	0x56	0x101
0x102	0x34	0x12	0x103

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 MPLAB C30.

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

MPLAB C30 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 MPLAB C30 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.

Chapter 6. Device Support Files

6.1 INTRODUCTION

This section discusses device support files used in support of MPLAB C30 compilation.

6.2 HIGHLIGHTS

This section discusses:

- Processor Header Files
- Register Definition Files
- Using SFR's

6.3 PROCESSOR HEADER FILES

The processor header files are distributed with the language tools. These header files define the available special function registers (SFR's) for each dsPIC 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 PIC30F2010 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 bitfields) 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__));
```

<p>Note: The symbols <code>CORCON</code> and <code>CORCONbits</code> refer to the same register and will resolve to the same address at link time.</p>

6.4 REGISTER DEFINITION FILES

The processor header files described in **Section 6.3 “Processor Header Files”** name all SFR's for each part, but they do not define the addresses of the SFR's. 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 PIC30F2010 part, then it may be compiled and linked using the following command line:

```
pic30-gcc -o app2010.cof -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 PIC30F2010 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”**.

6.5 USING SFRS

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

1. Include the processor header file for the appropriate device. This provides the source code with the SFR's that are available for that device. For instance, the following statement includes the header files for the PIC30F6014 part:

```
#include <p30f6014.h>
```

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

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

```
TMR1 = 0;
```

This next statement represents the 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 SFR's. (Remember the bit structure will have the same address as the SFR at link time.) Example 6.1 would use:

```
p30f6014.gld
```

See *MPLAB ASM30*, *MPLAB LINK30* and *Utilities User's Guide* (DS51317) for more information on using linker scripts.

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

EXAMPLE 6-1: SAMPLE REAL-TIME CLOCK

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

#include <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)
    {
        sticks = 0;              /* if time to rollover */
        RTclock.seconds++;        /* clear seconds ticks */
        RTclock.seconds++;        /* and increment seconds */
    }

    IFS0bits.T1IF = 0;           /* clear interrupt flag */
    return;
}
```


6.6 USING MACROS

Processor header files define, in addition to special function registers (SFR), useful macros for the dsPIC30F family of Digital Signal Controllers (DSCs).

6.6.1 Configuration Bits Setup Macros

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

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

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

Similarly, to set the FBORPOR bit:

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

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

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

6.6.2 In-Line 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");}
```

6.6.3 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__((section(".xbss,b"), aligned(N)))
#define _XDATA(N)   __attribute__((section(".xdata,d"), aligned(N)))
#define _YBSS(N)    __attribute__((section(".ybss,b"), aligned(N)))
#define _YDATA(N)   __attribute__((section(".ydata,d"), aligned(N)))
#define _EEDATA(N)  __attribute__((section(".eedata,r"), 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__((section(".pbss,b")))
#define _NEAR       __attribute__((near))
```

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

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

6.6.4 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);
```

<p>Note: ISRs will be installed into the interrupt vector tables automatically if the reserved names listed in Table 7-1 are used.</p>

Chapter 7. Interrupts

7.1 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 dsPIC30F 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 MPLAB C30 compiler provides full support for interrupt processing in C or inline assembly code. This chapter presents an overview of interrupt processing.

7.2 HIGHLIGHTS

This chapter covers the following topics:

- **Writing an Interrupt Service Routine** – You can designate one or more C functions as interrupt service routines (ISR's) 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 dsPIC30F 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** – MPLAB C30 supports nested interrupts.
- **Enabling/Disabling Interrupts** – Enabling and disabling interrupt sources occurs at two levels: globally and individually.

7.3 WRITING AN INTERRUPT SERVICE ROUTINE

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

7.3.1 Guidelines for Writing ISR's

The guidelines for writing ISR's are:

- declare ISR's with no parameters and a `void` return type (mandatory)
- do not let ISR's be called by main line code (mandatory)
- do not let ISR's call other functions (recommended)

An MPLAB C30 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).

ISR's 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, ISR's should not call other functions. This is recommended because of latency issues. See **Section 7.6 "Latency"** for more information.

7.3.2 Syntax for Writing ISR's

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 7.4 "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.

7.3.3 Coding ISR's

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);
```

7.3.4 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);
```

7.4 WRITING THE INTERRUPT VECTOR

The dsPIC device has two interrupt vector tables – a primary and an alternate table – each containing 62 exception vectors.

The 62 exception sources have associated with them a primary and alternate exception vector, each occupying a program word, as shown in Table 7-1. The alternate vector name is used when the `ALTIVT` bit is set in the `INTCON2` register.

Note: A dsPIC device reset is not handled through the interrupt vector table. Instead, upon device reset, the dsPIC 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 runtime startup module.

TABLE 7-1: INTERRUPT VECTORS

IRQ#	Vector Function	Primary Name	Alternate Name
n/a	Reserved	<code>_ReservedTrap0</code>	<code>_AltReservedTrap0</code>
n/a	Oscillator fail trap	<code>_OscillatorFail</code>	<code>_AltOscillatorFail</code>
n/a	Address error trap	<code>_AddressError</code>	<code>_AltAddressError</code>
n/a	Stack error trap	<code>_StackError</code>	<code>_AltStackError</code>
n/a	Math error trap	<code>_MathError</code>	<code>_AltMathError</code>
n/a	Reserved	<code>_ReservedTrap5</code>	<code>_AltReservedTrap5</code>
n/a	Reserved	<code>_ReservedTrap6</code>	<code>_AltReservedTrap6</code>
n/a	Reserved	<code>_ReservedTrap7</code>	<code>_AltReservedTrap7</code>
0	INT0-External interrupt 0	<code>_INT0Interrupt</code>	<code>_AltINT0Interrupt</code>
1	IC1-Input capture 1	<code>_IC1Interrupt</code>	<code>_AltIC1Interrupt</code>
2	OC1-Output compare 1	<code>_OC1Interrupt</code>	<code>_AltOC1Interrupt</code>
3	TMR1-Timer 1	<code>_T1Interrupt</code>	<code>_AltT1Interrupt</code>
4	IC2-Input capture 2	<code>_IC2Interrupt</code>	<code>_AltIC2Interrupt</code>
5	OC2-Output compare 2	<code>_OC2Interrupt</code>	<code>_AltOC2Interrupt</code>
6	TMR2-Timer 2	<code>_T2Interrupt</code>	<code>_AltT2Interrupt</code>
7	TMR3-Timer 3	<code>_T3Interrupt</code>	<code>_AltT3Interrupt</code>
8	SPI1-Serial peripheral interface 1	<code>_SPI1Interrupt</code>	<code>_AltSPI1Interrupt</code>
9	UART1RX-UART1 Receiver	<code>_U1RXInterrupt</code>	<code>_AltU1RXInterrupt</code>
10	UART1TX-UART1 Transmitter	<code>_U1TXInterrupt</code>	<code>_AltU1TXInterrupt</code>
11	ADC-ADC convert done	<code>_ADCInterrupt</code>	<code>_AltADCInterrupt</code>
12	NVM-NVM write complete	<code>_NVMInterrupt</code>	<code>_AltNVMInterrupt</code>
13	Slave I ² C Interrupt	<code>_SI2CInterrupt</code>	<code>_AltSI2CInterrupt</code>
14	Master I ² C Interrupt	<code>_MI2CInterrupt</code>	<code>_AltMI2CInterrupt</code>
15	CN-Input change interrupt	<code>_CNInterrupt</code>	<code>_AltCNInterrupt</code>
16	INT1-External interrupt 1	<code>_INT1Interrupt</code>	<code>_AltINT1Interrupt</code>
17	IC7-Input capture 7	<code>_IC7Interrupt</code>	<code>_AltIC7Interrupt</code>
18	IC8-Input capture 8	<code>_IC8Interrupt</code>	<code>_AltIC8Interrupt</code>
19	OC3-Output compare 3	<code>_OC3Interrupt</code>	<code>_AltOC3Interrupt</code>
20	OC4-Output compare 4	<code>_OC4Interrupt</code>	<code>_AltOC4Interrupt</code>
21	TMR4-Timer 4	<code>_T4Interrupt</code>	<code>_AltT4Interrupt</code>
22	TMR5-Timer 5	<code>_T5Interrupt</code>	<code>_AltT5Interrupt</code>
23	INT2-External interrupt 2	<code>_INT2Interrupt</code>	<code>_AltINT2Interrupt</code>

TABLE 7-1: INTERRUPT VECTORS (CONTINUED)

IRQ#	Vector Function	Primary Name	Alternate Name
24	UART2RX-UART2 receiver	_U2RXInterrupt	_AltU2RXInterrupt
25	UART2TX-UART2 transmitter	_U2TXInterrupt	_AltU2TXInterrupt
26	SPI2-Serial peripheral interface 2	_SPI2Interrupt	_AltSPI2Interrupt
27	CAN1-Combined IRQ	_C1Interrupt	_AltC1Interrupt
28	IC3-Input capture 3	_IC3Interrupt	_AltIC3Interrupt
29	IC4-Input capture 4	_IC4Interrupt	_AltIC4Interrupt
30	IC5-Input capture 5	_IC5Interrupt	_AltIC5Interrupt
31	IC6-Input capture 6	_IC6Interrupt	_AltIC6Interrupt
32	OC5-Output compare 5	_OC5Interrupt	_AltOC5Interrupt
33	OC6-Output compare 6	_OC6Interrupt	_AltOC6Interrupt
34	OC7-Output compare 7	_OC7Interrupt	_AltOC7Interrupt
35	OC8-Output compare 8	_OC8Interrupt	_AltOC8Interrupt
36	INT3-External interrupt 3	_INT3Interrupt	_AltINT3Interrupt
37	INT4-External interrupt 4	_INT4Interrupt	_AltINT4Interrupt
38	CAN2-Combined IRQ	_C2Interrupt	_AltC2Interrupt
39	PWM-PWM period match	_PWMInterrupt	_AltPWMInterrupt
40	QEI-Position counter compare	_QEInterrupt	_AltQEInterrupt
41	DCI-CODEC transfer done	_DCIInterrupt	_AltDCIInterrupt
42	PLVD-Low voltage detect	_LVDInterrupt	_AltLVDInterrupt
43	FLTA-MPWM fault A	_FLTAInterrupt	_AltFLTAInterrupt
44	FLTB-MPWM fault B	_FLTBInterrupt	_AltFLTBInterrupt
45	Reserved	_Interrupt45	_AltInterrupt45
46	Reserved	_Interrupt46	_AltInterrupt46
47	Reserved	_Interrupt47	_AltInterrupt47
48	Reserved	_Interrupt48	_AltInterrupt48
49	Reserved	_Interrupt49	_AltInterrupt49
50	Reserved	_Interrupt50	_AltInterrupt50
51	Reserved	_Interrupt51	_AltInterrupt51
52	Reserved	_Interrupt52	_AltInterrupt52
53	Reserved	_Interrupt53	_AltInterrupt53

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 `__Interruptn`, where `n` is the vector number. Therefore, the C identifiers `_Interrupt45` through `_Interrupt53` are reserved by the compiler. In the same way, if the `altirq` parameter of the interrupt attribute is used, the compiler creates the external symbol name `__AltInterruptn`, where `n` is the vector number. Therefore, the C identifiers `_AltInterrupt45` through `_AltInterrupt53` are reserved by the compiler.

7.5 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);
```


7.6 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** – MPLAB C30 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.

7.7 NESTING INTERRUPTS

The dsPIC30F devices support nested interrupts. Since processor resources are saved on the stack in an ISR, nested ISR's 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 dsPIC30F device comes out of reset with nested interrupts enabled. Each interrupt source is assigned a priority in the interrupt priority control registers (IPCn). If there is a pending 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.

7.8 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 (IECn). Setting an interrupt enable bit to one (1) enables the corresponding interrupt; clearing the interrupt enable bit to zero (0) disables the corresponding interrupt. When the dsPIC 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 in-line assembly. For example, the in-line assembly statement:

```
__asm__ volatile ("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,

```
__asm__ volatile("disi #0x3FFF"); /* disable interrupts */  
/* ... protected C code ... */  
__asm__ volatile("disi #0x0000"); /* enable interrupts */
```

An alternative approach is to write directly to the DISCNT register, which has the same effect in hardware as the DISI instruction, but it has the advantage for the C programmer that it avoids the use of in-line assembly. This is desirable because there are certain optimizations that the compiler will not perform if in-line assembly is used in a function. So, instead of the sequence shown above, use

```
DISCNT = 0x3FFF; /* disable interrupts */  
/* ... protected C code ... */  
DISCNT = 0x0000; /* enable interrupts */
```

Chapter 8. Mixing Assembly Language and C Modules

8.1 INTRODUCTION

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

8.2 HIGHLIGHTS

This chapter covers the following topics:

- **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.

8.3 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 4.14 “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 8-1: 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

.section .bss,"b"
.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 MPLAB C30 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.

Mixing Assembly Language and C Modules

EXAMPLE 8-2: 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 4.13.2 “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.

8.4 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. In-line 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`.

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.

The following constraint letters are supported by MPLAB C30.

TABLE 8-1: CONSTRAINT LETTERS SUPPORTED BY MPLAB C30

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 <i>earlyclobber</i> operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address.
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.
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.
T	A near or far data operand.
U	A near data operand.

For example, here is how to use the dsPIC device's *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 bitfield), the constraint must allow a register. In that case, MPLAB C30 will use the register as the output of the *asm*, and then store that register into the output. If output operands are write-only, MPLAB C30 will assume that the values in these operands before the instruction are dead and need not be generated.

Mixing Assembly Language and C Modules

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 for the dsPIC device:

```
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 MPLAB C30 to not keep memory values cached in registers across the assembler instruction.

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 guaranteed 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. Unless an output operand has the `&` constraint modifier, MPLAB C30 may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use `&` for each output operand that may not overlap an input operand. For example, consider the following function:

```
int
exprbad(int a, int b)
{
    int c;

    __asm__ ("add %1,%2,%0\n sl %0,%1,%0"
            : "=r" (c) : "r" (a), "r" (b));

    return(c);
}
```

The intention is to compute the value $(a + b) \ll a$. However, as written, the value computed may or may not be this value. The correct coding informs the compiler that the operand `c` is modified before the `asm` instruction is finished using the input operands, as follows:

```
int
exprgood(int a, int b)
{
    int c;

    __asm__ ("add %1,%2,%0\n sl %0,%1,%0"
            : "&r" (c) : "r" (a), "r" (b));

    return(c);
}
```

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 guarantee 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 guarantee 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).

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. An instruction without output operands will not be deleted or moved significantly, regardless, unless it is unreachable.

Appendix A. Implementation-Defined Behavior

A.1 INTRODUCTION

This section discusses MPLAB C30 implementation-defined behavior. 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 Bitfields
- 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?

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 "Runtime 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
<code>\a</code>	7
<code>\b</code>	8
<code>\f</code>	12
<code>\n</code>	10
<code>\r</code>	13
<code>\t</code>	9
<code>\v</code>	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 is the results of bitwise operations on signed integers? (ISO 6.3)

Shift operators retain the sign. Other operators act as for 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.

Implementation-Defined Behavior

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 BITFIELDS

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

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

No conversions are applied.

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

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

What is the equivalent type for a plain `int` bitfield? (ISO 6.5.2.1)

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

What is the order of allocation of bitfields within an `int`? (ISO 6.5.2.1)

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

Can a bitfield 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: #PRAMA BEHAVIOR

Pragma	Behavior
<code>#pragma code section-name</code>	Names the code section.
<code>#pragma code</code>	Resets the name of the code section to its default (viz., <code>.text</code>).
<code>#pragma idata section-name</code>	Names the initialized data section.
<code>#pragma idata</code>	Resets the name of the initialized data section to its default value (viz., <code>.data</code>).
<code>#pragma udata section-name</code>	Names the uninitialized data section.
<code>#pragma udata</code>	Resets the name of the uninitialized data section to its default value (viz., <code>.bss</code>).
<code>#pragma interrupt function-name</code>	Designates function-name as an interrupt function.

What are the definitions for `__DATE__` and `__TIME__` respectively, when the date and time of translation are not available? (ISO 6.8.8)

Not applicable. The compiler is not supported in environments where these functions are not available.

A.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 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. MPLAB C30 C Compiler Diagnostics

B.1 INTRODUCTION

This appendix lists the most common diagnostic messages generated by the MPLAB C30 compiler.

B.2 HIGHLIGHTS

The MPLAB C30 compiler can produce two kinds of diagnostic messages: errors and warnings. Each kind has a different purpose:

- *Errors* report problems that make it impossible to compile your program. MPLAB C30 reports errors with the source file name and line number where the problem is apparent.
- *Warnings* report other unusual conditions in your code that may indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text 'warning:' to distinguish them from error messages.

Warnings may indicate danger points 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 nonstandard features of MPLAB C30 C. 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.

B.3 ERRORS

Symbols

'...' in old-style identifier list

Ellipses ('...') is not legal in an old-style identifier list.

\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 not within a conditional

The `#elif` directive may only appear within a preprocessor conditional.

#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 not within a conditional

The #else directive may only appear within a preprocessor conditional.

#else or #elif after #else

In a preprocessor conditional, #elif or a second #else may not be used after a #else.

#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 "fname" or <fname>

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.

'#name' not within a conditional

The #if directive must be used before using #else, #elif or #endif.

'#' operator should be followed by a macro argument name

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.

address of global register variable 'identifier' requested

It is not legal to request the address of a global register variable.

"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 bitfield

The '__alignof' operator may not be applied to a bitfield.

__alignof__ applied to an incomplete type

The operator __alignof__ may not be applied to an incomplete type

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

-b is not supported by MPLAB C30.

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 '-V' is missing

-V is not supported by MPLAB C30.

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.

arguments given to macro '*identifier*'

The specified macro does not take arguments, yet it is called with arguments. This is not allowed.

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 initialized from non-constant array expression

Arrays must not be initialized from non-constant array expressions.

array size missing in '*identifier*'

An array size is missing.

array subscript is not an integer

Array subscripts must be integers.

arrays of functions are not meaningful

Do not attempt to use arrays of functions; they are not meaningful in C.

'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.

***token* at end of input**

The specified *token* appears at the end of input, where it is invalid.

attempt to take address of bitfield structure member '*identifier*'

It is not legal to attempt to take address of a bitfield structure member.

'*attribute*' attribute applies only to functions

The attribute '*attribute*' may only be applied to functions.

B

badly punctuated parameter list in '#define'

Do not use badly punctuated parameter lists in '#define' preprocessor directives.

***token* before string constant**

The specified *token* appears before a string constant, where it is invalid.

bitfield '*identifier*' has invalid type

Bitfields must be of enumerated or integral type.

bitfield '*identifier*' width not an integer constant

Bitfield widths must be integer constants.

both 'f' and 'l' in floating constant

Specify either 'f' or 'l' in a floating constant, not both.

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.

calling non-function symbol 'symbol' is not possible

Attempt to call a symbol which is not a function and not in memory.

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 bitfield 'identifier'

It is not legal to attempt to take address of a bitfield.

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 integer constant is too wide for complex int

The specified constant cannot be represented as a complex int.

complex invalid for '*identifier*'

The complex qualifier may only be applied to integral and floating types.

conflicting declarations of '*identifier*'

The *identifier* has multiple, conflicting declarations.

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 incomplete type

It is not possible to convert to an incomplete type.

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'.

'defined' must be followed by ident or (ident)

'defined' is expecting an identifier.

dereferencing pointer to incomplete type

A dereferenced pointer must be a pointer to an incomplete type.

directives may not be used inside a macro argument

Evaluation of a function macro has encountered an argument that looks like a preprocessor directive.

division by zero in #if

Division by zero is not computable.

double quoted strings not allowed in #if expressions

double quoted strings cannot be used in #if expressions.

duplicate case value

Case values must be unique.

duplicate label 'identifier'

Labels must be unique within their scope.

duplicate label declaration '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 MPLAB C30.

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 #if expression

A expression that evaluates to a constant arithmetic value was expected.

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.

extra elements in scalar initializer

Scalar initializers must not contain extra elements.

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.

floating constant exponent has no digits

Floating constant exponents must have digits.

floating constant out of range

Floating constants must not be out of range.

floating point numbers are not allowed in #if

Do not use floating-point numbers in #if expressions; they are not allowed.

'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 constant has no 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 *asm* statement is invalid.

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 ‘code’ as the space attribute parameter.

invalid attribute ‘*name*’ ignored

‘name’ is not a known attribute. See **Section 2.3.1 “Specifying Attributes of Variables”** and **Section 2.3.2 “Specifying Attributes of Functions”** for a list of valid attributes.

invalid character ‘*character*’ in #if

This message appears when an unprintable character, such as a control character, appears after #if.

invalid character constant in #if

Do not use an invalid character in #if.

invalid format ‘#line’ command

The format of the #line directive is invalid.

invalid initial value for member ‘*name*’

Bitfield ‘name’ can only be initialized by an integer.

invalid initializer

Do not use invalid initializers.

invalid initializer for bit string

The specified bitfield initializer is invalid.

Invalid location qualifier: ‘*symbol*’

Expecting ‘sfr’ or ‘gpr’, which are ignored on dsPIC, as location qualifiers.

invalid macro name

The macro name is not valid.

invalid macro name ‘*name*’

Macro names must start with a letter or underscore followed by more letters, numbers or underscores.

invalid number in #if expression

The specified number is not valid in an #if expression.

invalid operands to binary operator

The operands to the 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 pre prologue 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.

Invalid token in expression

A token in the expression is not a valid C token.

invalid truth-value expression

The expression is invalid.

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.

I/O error on output

The system encountered an input/output error. One possible cause is a lack of disk space.

"*name*" is not a valid filename

#line requires a valid filename.

"*token*" is not valid in #if expressions

An invalid *token* was used in the expression of the #if command. #if expects a constant expression that evaluates to a constant arithmetic value.

'*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 '*...*'.

J**junk after end of expression.**

Do not place junk after the end of expressions.

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.

'*LI*' and '*IL*' are not valid integer suffixes

Do not mix cases. Use uppercase LL or lowercase ll for the long, long suffix.

long long long is too long for GCC

MPLAB C30 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.

'lul' is not a valid integer suffix

The long long suffix must use 'LL' or 'll' and they may not be separated by another suffix.

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'.

macro or #include recursion too deep

Maximum level of nesting of macro expansion or #include is 200.

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

Expecting an operator between operands.

missing binary operator before ‘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.

missing white space after number ‘number’

White space should appear after the specified ‘number’.

more than *n* operands in ‘asm’

The asm statement is invalid.

more than one ‘f’ suffix on floating constant

There should be at most one ‘f’ in floating constants.

more than one ‘i’ or ‘j’ suffix on floating constant

It is not legal to have more than one ‘i’ or ‘j’ in a floating constant.

more than one ‘i’ or ‘j’ suffix on integer constant

It is not legal to have more than one ‘i’ or ‘j’ in an integer constant

more than one ‘l’ suffix on floating constant

There should be at most one ‘l’ in floating constants.

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 bitfield ‘identifier’

Bitfield 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 args to macro ‘name’

Macro ‘name’ was expecting 1 or more arguments.

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 MPLAB C30.

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.

numeric constant contains digits beyond the radix

Digits in numeric constants must not be beyond the radix.

numeric constant with no digits

Numeric constants must have digits.

O

only 1 arg to macro '*name*'

Not enough arguments were passed to macro '*name*'.

only n args to macro '*name*'

Not enough arguments were passed to macro '*name*'.

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 left operand

Preprocessor operator '*symbol*' requires an operand on the left side.

operator '*symbol*' has no right operand

Preprocessor operator '*symbol*' requires an operand on the right side.

output number *n* not directly addressable

The asm statement is invalid.

output operand constraint lacks '='

The asm statement is invalid.

output operand is constant in 'asm'

The asm statement is invalid.

overflow in enumeration values

Enumeration values must be in the range of 'int'.

P

parameter '*identifier*' declared void

Parameters may not be declared void.

parameter '*identifier*' has incomplete type

Parameters must have complete types.

parameter '*identifier*' has just a forward declaration

Parameters must have complete types; forward declarations are insufficient.

parameter '*identifier*' is initialized

It is not legal to initialize parameters.

parameter name missing

The macro was expecting a parameter name. Check for two commas without a name between.

parameter name missing from parameter list

Parameter names must be included in the parameter list.

parameter name omitted

Parameter names may not be omitted.

parameter types given both in parameter 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.

parse error; also virtual memory exhausted

Not enough memory left to write error message.

-pedantic and -traditional are mutually exclusive

The options -pedantic and -traditional can not be used together.

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.

possible real start of unterminated constant

This message identifies the possible real start of unterminated constant; shown in conjunction with the "unterminated string or character constant" message.

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.

prototype for '*identifier*' follows and number of arguments doesn't match

A function prototype follows a definition without a prototype, and the number of arguments is inconsistent between the two.

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 bitfield

'sizeof' must not be applied to a bitfield.

size of applied to an incomplete type

Size of can only be applied to complete types.

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.

static or type qualifiers in non-parameter array declarator

Keyword '*static*' may not be used in this context.

storage class specified for parameter '*identifier*'

A storage class may not be specified for a parameter.

storage class specified for structure field '*identifier*'

A storage class may not be specified for a structure field.

storage class specified for typename

A storage class may not be specified for a typename.

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.

string constants are not valid in #if

Do not use string constants numbers in #if expressions; they are not allowed.

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.

subscript missing in array reference

Do not omit subscripts array references.

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 following ':'

A conditional expression '?' expects ':' between the second and third operands, in this case, it was missing.

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.

three 'l' suffixes on integer constant

It is not legal to have three 'l's in an integer constant.

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 *n* args to macro '*name*'

Too many arguments were passed to macro '*name*'.

too many decimal points in floating constant

Expecting only one decimal point.

too many 'l' suffixes in integer constant

Do not use too many 'l' suffixes in integer constants.

top-level declaration of '*identifier*' specifies 'auto'

Auto variables can only be declared inside functions.

-trigraphs and -traditional are mutually exclusive

The options -trigraphs and -traditional can not be used together.

two or more data types in declaration of '*identifier*'

Each identifier may have only a single data type.

two types specified in one empty declaration

No more than one type should be specified.

two 'u' suffixes on integer constant

It is not legal to have two 'u's in an integer constant.

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.

U**unbalanced '#endif'**

An #endif directive appears with a balancing #if.

'*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.

underscore in number

An underscore is not allowed.

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 C standard '*standard*'

The argument *standard* to the `-std` command-line argument is invalid.

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.

unnamed fields of type other than struct or union are not allowed

Fields of type other than struct or union must have names.

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 #*'name'* conditional

`#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.

unterminated macro call

Macro calls may not be unterminated.

unterminated parameter list in '*#define*'

Parameter lists in '*#define*' preprocessor directives must not be unterminated.

unterminated string or character constant

A string or character constant is unterminated.

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.

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 abs

Do not use the wrong type of argument to this operator.

wrong type argument to bit-complement

Do not use the wrong type of argument to this operator.

wrong type argument to conjugation

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 or negative size array '*identifier*'

The size of the specified array must be strictly positive.

zero width for bitfield '*identifier*'

Bitfields may not have zero width.

B.4 WARNINGS

Symbols

'/' within comment

A comment mark was found within a comment.

'\$' character(s) in identifier

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.

#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.

#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.

#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 MPLAB C30. 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.

bitfield '*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 alignment 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 runtime result of the comparison. The result is always n .

comparison is always n due to width of bitfield

A comparison involving a bitfield always evaluates to n because of the width of the bitfield.

comparison is always false due to limited range of data type

A comparison will always evaluate to false at runtime, 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 runtime, 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 runtime.

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 runtime.

comparisons like $X \leq Y \leq Z$ do not have their mathematical meaning

A C expression does not necessarily mean the same thing as the corresponding mathematical expression. In particular the C expression $X \leq Y \leq Z$ is not equivalent to the mathematical expression $X \leq Y \leq Z$.

conflicting types for built-in function '*identifier*'

The specified function has the same name as a built-in function but is declared with conflicting types.

const declaration for '*identifier*' follows non-const

The specified identifier was declared const after it was previously declared as non-const.

control reaches end of non-void function

All exit paths from non-void function should return an appropriate value. The compiler detected a case where a non-void function terminates, without an explicit return value. Therefore, the return value might be unpredictable.

conversion from NaN to int

NaN is not-a-number so converting an undefined number to an int is unpredictable.

conversion from NaN to unsigned int

NaN is not-a-number so converting an undefined number to an unsigned int is unpredictable.

conversion lacks type at end of format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that a format field in the format string lacked a type specifier.

concatenation of string literals with `__FUNCTION__` is deprecated

`__FUNCTION__` will be handled the same way as `__func__` (which is defined by the ISO standard C99). `__func__` is a variable, not a string literal, so it does not concatenate with other string literals.

conflicting types for 'identifier'

The specified identifier has multiple, inconsistent declarations.

D

data definition has no type or storage class

A data definition was detected that lacked a type and storage class.

data qualifier 'qualifier' ignored

Data qualifiers, which include 'access', 'shared' and 'overlay', are not used in MPLAB C30 but are there for compatibility with MPLAB C17 and C18.

decimal constant is so large that it is unsigned

An decimal 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.

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.

declaration of 'identifier' shadows previous local

The specified *identifier* declaration shadows a previous local, making the previous local 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 specified range

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 MPLAB C30 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 may not be in radix 16

A floating constant may not be in radix 16.

floating point number exceeds range of ‘double’

A floating-point constant is too large or too small (in magnitude) to be represented as a ‘double’.

floating point number exceeds range of ‘float’

A floating-point constant is too large or too small (in magnitude) to be represented as a ‘float’.

floating point number 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

A number is too large to be expressed so it is assigned a special value that stands for infinity.

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

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.

initializer element is not constant

Initializer elements should be constant.

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 MPLAB C30 compiler, and is ignored.

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.

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.

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.

inline function '*name*' given attribute noline

The function '*name*' has been declared as inline but the noline attribute prevents the function from being considered for inlining.

inlining failed in call to '*identifier*' called from here

The compiler was unable to inline the call to the specified function.

integer constant contains digits beyond the radix

All digits must be within the radix being used. For instance, only the digits 0 thru 7 may be used for the octal radix.

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 larger than the maximum value of ‘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 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 bitfield 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 permit use of ‘varargs.h’

ISO C uses `stdarg.h` instead.

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 used 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 imaginary numeric constants

ISO C does not allow imaginary numeric constants.

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 does not support `'[*]'` array declarators

Variable length arrays are not supported in ISO C89.

ISO C89 does not support complex types

Complex types, such as, `__complex__ float x`, are not supported in ISO C89.

ISO C89 does not support flexible array members

A flexible array member is a new feature in C99. ISO C89 does not support it.

ISO C89 does not support `'long long'`

The `long long` type is not supported in ISO C89.

ISO C89 does not support `'static'` or type qualifiers in parameter array declarators

When using an array as a parameter to a function, ISO C89 does not allow the array declarator to use `'static'` or type qualifiers.

ISO C89 does not support the `'char'` `'function'` format

ISO C does not support the specification character `'char'` for the specified function format.

ISO C89 does not support the `'modifier'` `'function'` length modifier

The specified modifier is not supported as a length modifier for the given function.

ISO C89 forbids compound literals

Compound literals are not valid in ISO C89.

ISO C89 forbids long long integer constants

Integer constants are not allowed to be declared long long 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 C89 forbids variable-size array '*name*'

In ISO C89, 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 MPLAB C30 but are there for compatibility with MPLAB C17 and C18.

'long' switch expression not converted to 'int' in ISO C

ISO C does not convert 'long' switch expressions to 'int'.

M

'main' is usually a function

The identifier main is usually used for the name of the main entry point of an application. The compiler detected that it was being used in some other way, for example, as the name of a variable.

'*operation*' makes integer from pointer without a cast

A pointer has been implicitly converted to an integer.

'*operation*' makes pointer from integer without a cast

An integer has been implicitly converted to a pointer.

malformed '#pragma pack-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma pack(pop[,id])-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma pack(push[,id],<n>)-ignored'

The syntax of the pack pragma is incorrect.

malformed '#pragma weak-ignored'

The syntax of the weak pragma is incorrect.

'*identifier*' might be used uninitialized in this function

The compiler detected a control path through a function which might use the specified identifier before it has been initialized.

missing braces around initializer

A required set of braces around an initializer is missing.

missing initializer

An initializer is missing.

modification by 'asm' of read-only variable '*identifier*'

A const variable is the left-hand-side of an assignment in an 'asm' statement.

multi-character '*character*' constant

A character constant contains more than one character.

multi-line string literals are deprecated

A string extends over multiple lines.

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 name starts with a digit in #define

The -traditional switch will except a name that starts with a digit but it is recommended names start with a letter or underscore.

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.

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.

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.

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.

repeated '*flag*' flag in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that one of the flags { ,+,#,0,- } was repeated in the format string.

return-type defaults to 'int'

In the absence of an explicit function return-type declaration, the compiler assumes that the function returns an int.

return type of '*name*' is not 'int'

The compiler is expecting the return type of '*name*' to be 'int'.

'return' with a value, in function returning void

The function was declared as void but returned a value.

'return' with no value, in function returning non-void

A function declared to return a non-void value contains a return statement with no value. This is inconsistent.

right shift count \geq width of type

Shift counts should be less than the number of bits in the type being shifted. Otherwise, the shift is meaningless, and the result is undefined.

right shift count is negative

Shift counts should be positive. A negative right shift count does not mean shift left; it is meaningless.

S

second argument of '*identifier*' should be 'char **'

Expecting second argument of specified identifier to be of type 'char **'.

second parameter of '*va_start*' not last named argument

The second parameter of '*va_start*' must be the last named argument.

shadowing built-in function '*identifier*'

The specified function has the same name as a built-in function, and consequently shadows the built-in function.

shadowing library function '*identifier*'

The specified function has the same name as a library function, and consequently shadows the library function.

shift count \geq width of type

Shift counts should be less than the number of bits in the type being shifted. Otherwise, the shift is meaningless, and the result is undefined.

shift count is negative

Shift counts should be positive. A negative left shift count does not mean shift right, nor does a negative right shift count mean shift left; they are meaningless.

size of '*name*' is larger than *n* bytes

Using `-Wlarger-than-len` will produce the above warning when the size of '*name*' is larger than the *len* bytes defined.

size of '*identifier*' is *n* bytes

The size of the specified identifier (which is *n* bytes) is larger than the size specified with the `-Wlarger-than-len` command-line option.

size of return value of '*name*' is larger than *n* bytes

Using `-Wlarger-than-len` will produce the above warning when the size of the return value of '*name*' is larger than the *len* bytes defined.

size of return value of '*identifier*' is *n* bytes

The size of the return value of the specified function is *n* bytes, which is larger than the size specified with the `-Wlarger-than-len` command-line option.

sizeof applied to a function type

It is not recommended to apply the sizeof operator to a function type.

sizeof applied to a void type

The sizeof operator should not be applied to a void type.

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 Cn compilers are required to support

The maximum string length for ISO C89 is 509. The maximum string length for ISO C99 is 4095.

'struct identifier' declared inside parameter list

The specified struct is declared inside a function parameter list. It is usually better programming practice to declare structs outside parameter lists, since they can never become complete types when defined inside parameter lists.

struct has no members

The structure is empty, it has no members.

structure defined inside parms

A union is defined inside a function parameter list.

style of line directive is a GCC extension

Use the format '#line linenum' for traditional C.

subscript has type 'char'

An array subscript has type 'char'.

suggest explicit braces to avoid ambiguous 'else'

A nested if statement has an ambiguous else clause. It is recommended that braces be used to remove the ambiguity.

suggest hiding #directive from traditional C with an indented #

The specified directive is not traditional C and may be 'hidden' by indenting the #. A directive is ignored unless its # is in column 1.

suggest not using #elif in traditional C

#elif should not be used in traditional K&R C.

suggest parentheses around assignment used as truth value

When assignments are used as truth values, they should be surrounded by parentheses, to make the intention clear to readers of the source program.

suggest parentheses around + or - inside shift
suggest parentheses around && within ||
suggest parentheses around arithmetic in operand of |
suggest parentheses around comparison in operand of |
suggest parentheses around arithmetic in operand of ^
suggest parentheses around comparison in operand of ^
suggest parentheses around + or - in operand of &
suggest parentheses around comparison in operand of &

While operator precedence is well defined in C, sometimes a reader of an expression might be required to expend a few additional microseconds in comprehending the evaluation order of operands in an expression if the reader has to rely solely upon the precedence rules, without the aid of explicit parentheses. A case in point is the use of the '+' or '-' operator inside a shift. Many readers will be spared unnecessary effort if parentheses are used to clearly express the intent of the programmer, even though the intent is unambiguous to the programmer and to the compiler.

T

'identifier' takes only zero or two arguments

Expecting zero or two arguments only.

the meaning of '\a' varies with -traditional

When the `-traditional` command line argument 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' varies with -traditional

When the `-traditional` command line argument 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.

too many 'l' suffixes in integer constant

ISO C89 does not support long, long integer constants, only long integer constants.

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 'u' 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.

traditional is deprecated and may be removed

Traditional is pre-ISO C. It may be of use for compiling some very old programs that have not been updated to ISO C, but should not be used for new programs.

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 of 'identifier' defaults to 'int'

In the absence of an explicit type declaration, the compiler assumes that *identifier's* type is int.

type of external 'identifier' is not global

Externals must be of global type.

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 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 both ' ' and '+' flags in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that both the flags ' ' (space) and '+' appear in the format string.

use of both '0' and '-' flags in format

When checking the argument list of a call to *printf*, *scanf*, etc., the compiler found that both the flags '0' and '-' appear in the format string.

use of '*length*' length character 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 format string

When checking the argument list of a call to `printf`, `scanf`, etc., the compiler found that the format string was empty (`""`).

NOTES:

Appendix C. MPLAB C18 vs. MPLAB C30 C Compiler

C.1 INTRODUCTION

The purpose of this chapter is to highlight the differences between the MPLAB C18 and MPLAB C30 C compilers. For more details on the MPLAB C18 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 and Function Arguments
- Storage Qualifiers
- Predefined Macro Names
- Integer Promotions
- Numeric Constants
- String Constants
- Anonymous Structures
- Access Memory
- In-line Assembly
- Pragmas
- Memory Models
- Calling Conventions
- Startup Code
- Compiler-Managed Resources
- Optimizations
- Object Module Format
- Implementation-Defined Behavior
- Bitfields

C.3 DATA FORMATS

TABLE C-1: NUMBER OF BITS USED IN DATA FORMATS

Data Format	MPLAB C18 ⁽¹⁾	MPLAB C30 ⁽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:** MPLAB C18 uses its own data format, which is similar to IEEE-754 format, but with the top nine bits rotated (see Table C-2).
- 2:** MPLAB C30 uses IEEE-754 format.
- 3:** See Section 5.5 “Floating Point”.

TABLE C-2: MPLAB C18 FLOATING-POINT VS. MPLAB C30 IEEE-754 FORMAT

Standard	Byte 3	Byte 2	Byte 1	Byte 0
MPLAB C30	seeeeeee1	e ₀ ddd dddd16	dddd dddd8	dddd dddd0
MPLAB C18	eeeeeeee0	sddd dddd16	dddd dddd8	dddd dddd0

Legend: s = sign bit, d = mantissa, e = exponent

C.4 POINTERS

TABLE C-3: NUMBER OF BITS USED FOR POINTERS

Memory Type	MPLAB C18	MPLAB C30
Program Memory - Near	16	16
Program Memory - Far	24	16
Data Memory	16	16

C.5 STORAGE CLASSES

MPLAB C18 allows the non-ANSI storage class specifiers `overlay` for variables and `auto` or `static` for function arguments.

MPLAB C30 does not allow these specifiers.

C.6 STACK USAGE

TABLE C-4: TYPE OF STACK USED

Items on Stack	MPLAB C18	MPLAB C30
Return Addresses	hardware	software
Local Variables	software	software

C.7 STORAGE QUALIFIERS

MPLAB C18 uses the non-ANSI `far`, `near`, `rom` and `ram` type qualifiers.

MPLAB C30 uses the non-ANSI `far`, `near` and `space` attributes.

EXAMPLE C-1: DEFINING A NEAR VARIABLE

```
C18: near int gVariable;  
C30: __attribute__((near)) int gVariable;
```

EXAMPLE C-2: DEFINING A FAR VARIABLE

```
C18: far int gVariable;  
C30: __attribute__((far)) int gVariable;
```

EXAMPLE C-3: CREATING A VARIABLE IN PROGRAM MEMORY

```
C18: rom int gArray[6] = {0,1,2,3,4,5};  
C30: __attribute__((section(".romdata"), space(prog))) int  
gArray[6] =  
    {0,1,2,3,4,5};
```

C.8 PREDEFINED MACRO NAMES

MPLAB C18 defines `__18CXX`, `__18F242`, ... (all other processors with `__` prefix) and `__SMALL__` or `__LARGE__`, depending on the selected memory model.

MPLAB C30 defines `__dsPIC30`.

C.9 INTEGER PROMOTIONS

MPLAB C18 performs integer promotions at the size of the largest operand even if both operands are smaller than an `int`. MPLAB C18 provides the `-Oi+` option to conform to the standard.

MPLAB C30 performs integer promotions at `int` precision or greater as mandated by ISO.

C.10 NUMERIC CONSTANTS

MPLAB C18 supports the non-ANSI binary `0b` prefix.

MPLAB C30 does not.

C.11 STRING CONSTANTS

MPLAB C18 keeps string constants in program memory in its `.stringtable` section. MPLAB C18 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.

MPLAB C30 accesses string constants from data memory or from program memory through the PSV window, allowing constants to be accessed like any other data.

C.12 ANONYMOUS STRUCTURES

MPLAB C18 supports non-ANSI anonymous structures inside of unions.
MPLAB C30 does not.

C.13 ACCESS MEMORY

dsPIC30F devices do not have access memory.

C.14 INLINE ASSEMBLY

MPLAB C18 uses non-ANSI `_asm` and `_endasm` to identify a block of inline assembly.
MPLAB C30 uses non-ANSI `asm`, which looks more like a function call. The MPLAB C30 use of the `asm` statement is detailed in **Section 8.4 “Using Inline Assembly Language”**.

C.15 PRAGMAS

MPLAB C18 uses pragmas for sections (`code`, `romdata`, `udata`, `idata`), interrupts (high-priority and low-priority) and variable locations (`bank`, `section`).
MPLAB C30 uses non-ANSI attributes instead of pragmas.

TABLE C-5: MPLAB C18 PRAGMAS VS. MPLAB C30 ATTRIBUTES

Pragma (MPLAB C18)	Attribute (MPLAB C30)
<code>#pragma udata [name]</code>	<code>__attribute__((section ("name")))</code>
<code>#pragma idata [name]</code>	<code>__attribute__((section ("name")))</code>
<code>#pragma romdata [name]</code>	<code>__attribute__((space (prog)))</code>
<code>#pragma code [name]</code>	<code>__attribute__((section ("name"))), __attribute__((space (prog)))</code>
<code>#pragma interruptlow</code>	<code>__attribute__((interrupt))</code>
<code>#pragma interrupt</code>	<code>__attribute__((interrupt, shadow))</code>
<code>#pragma varlocate bank</code>	NA*
<code>#pragma varlocate name</code>	NA*
*dsPIC devices do not have banks.	

EXAMPLE C-4: SPECIFY AN UNINITIALIZED VARIABLE IN A USER SECTION IN DATA MEMORY

```
C18: #pragma udata mybss
      int gi;
C30: int __attribute__((__section__(".mybss"))) gi;
```

MPLAB C18 vs. MPLAB C30 C Compiler

EXAMPLE C-5: LOCATE THE VARIABLE MABONGA AT ADDRESS 0x100 IN DATA MEMORY

```
C18: #pragma idata myDataSection=0x100;
      int Mabonga = 1;
C30: source code:
      int __attribute__((__section__(".myDataSection"))) Mabonga = 1;
linker script:
      .myDataSection 0x100 :
      {
      *(.myDataSection);
      } >data
```

EXAMPLE C-6: SPECIFY A VARIABLE TO BE PLACED IN PROGRAM MEMORY

```
C18: #pragma romdata const_table
      const rom char my_const_array[10] = {0, 1, 2, 3, 4, 5, 6,
      7, 8, 9};
C30: const __attribute__((section(".constst_table"), space
      (prog)))
      char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};
```

Note: The MPLAB C30 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 4.16 “Program Space Visibility (PSV) Usage”** for more on the PSV window.

EXAMPLE C-7: LOCATE THE FUNCTION PRINTSTRING AT ADDRESS 0x8000 IN PROGRAM MEMORY

```
C18: #pragma code myTextSection=0x8000;
      int PrintString(const char *s){...};
C30: source code:
      int __attribute__((__section__(".myTextSection")))
      PrintString(const char *s){...};
linker script:
      .myTextSection 0x8000 :
      {
      *(.myTextSection);
      } >program
```

EXAMPLE C-8: COMPILER AUTOMATICALLY SAVE AND RESTORES THE VARIABLES `var1` AND `var2`

```
C18: #pragma interrupt isr0 save=var1, var2
      void isr0(void)
      {
        /* perform interrupt function here */
      }

C30: void __attribute__((__interrupt__(__save__(var1,var2))))
      isr0(void)
      {
        /* perform interrupt function here */
      }
```

C.16 MEMORY MODELS

MPLAB C18 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).

MPLAB C30 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.17 CALLING CONVENTIONS

There are many differences in MPLAB C18 and MPLAB C30 calling conventions. Please refer to **Section 4.13 “Function Call Conventions”** for a discussion of MPLAB C30 calling conventions.

C.18 STARTUP CODE

MPLAB C18 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).

MPLAB C30 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.19 COMPILER-MANAGED RESOURCES

MPLAB C18 has the following managed resources: PC, WREG, STATUS, PROD, section `.tmpdata`, section `MATH_DATA`, FSR0, FSR1, FSR2, TBLPTR, TABLAT.

MPLAB C30 has the following managed resources: W0-W15, RCOUNT, SR.

C.20 OPTIMIZATIONS

The following optimizations are part of each compiler.

MPLAB C18	MPLAB C30
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: In MPLAB C30 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.21 OBJECT MODULE FORMAT

MPLAB C18 and MPLAB C30 use different COFF File Formats that are not interchangeable.

C.22 IMPLEMENTATION-DEFINED BEHAVIOR

For the right-shift of a negative-signed integral value:

- MPLAB C18 does not retain the sign bit
- MPLAB C30 retains the sign bit

C.23 BITFIELDS

Bitfields in MPLAB C18 cannot cross byte storage boundaries and, therefore, cannot be greater than 8 bits in size.

MPLAB C30 supports bitfields with any bit size, up to the size of the underlying type. Any integral type can be made into a bitfield. 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 MPLAB C30. `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 MPLAB C30. `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.

Appendix D. ASCII Character Set

TABLE D-1: ASCII CHARACTER SET

		Most Significant Character							
Least Significant Character	HEX	0	1	2	3	4	5	6	7
	0	NUL	DLE	Space	0	@	P	'	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	\$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(8	H	X	h	x
	9	HT	EM)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[k	{
	C	FF	FS	,	<	L	\	l	
	D	CR	GS	-	=	M]	m	}
	E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL	

NOTES:

Appendix E. GNU Free Documentation License

GNU Free Documentation License

Version 1.2, November 2002

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

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

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

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify, or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

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.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- c) State on the Title page the name of the publisher of the Modified Version, as the publisher.
- d) Preserve all the copyright notices of the Document.
- e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- h) Include an unaltered copy of this License.
- i) Preserve the section Entitled "History", 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.
- o) Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Glossary

Access Memory (PIC18 Only)

Special registers on PIC18XXXXX devices that allow access regardless of the setting of the bank select register (BSR).

Address

Value that identifies a location in memory.

Alphabetic Character

Alphabetic characters are those characters that are letters of the arabic alphabet (a, b, ..., z, A, B, ..., Z).

Alphanumeric

Alphanumeric characters are comprised of alphabetic characters and decimal digits (0,1, ..., 9).

Anonymous Structure

An unnamed structure.

ANSI

American National Standards Institute is an organization responsible for formulating and approving standards in the United States.

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 7 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.

Attribute

Characteristics of variables or functions in a C program which are used to describe machine-specific properties.

C

A general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators.

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.

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.

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 devices.

Digital Signal Processing

The computer manipulation of digital signals, commonly 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.

DSC

See Digital Signal Controller.

DSP

See Digital Signal Processor.

Endianness

Describes order of bytes in a multi-byte object.

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.

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.

Executable Code

Software that is ready to be loaded for execution.

Expressions

Combinations of constants and/or symbols separated by arithmetic or logical operators.

File Registers

On-chip data memory, including general purpose registers (GPRs) and special function registers (SFRs).

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

A C compiler implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the ISO library clause is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stddef.h>` and `<stdint.h>`.

GPR

General Purpose Register. The portion of device data memory (RAM) available for general use.

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.

High Level Language

A language for writing programs that is further removed from the processor than assembly.

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.

Initialized Data

Data which is defined with an initial value. In C,

```
int myVar=5;
```

defines a variable which will reside in an initialized data section.

Instruction Set

The collection of machine language instructions that a particular processor understands.

Instructions

A sequence of bits that tells a central processing unit to perform a particular operation and can contain data to be used in the operation.

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.

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

A function that is invoked when an interrupt occurs.

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.

Little Endianess

A data ordering scheme for multibyte data whereby the least significant byte is stored at the lower addresses.

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

Macroinstruction. An instruction that represents a sequence of instructions in abbreviated form.

Memory Models

A representation of the memory available to the application.

Microcontroller

A highly integrated chip that contains a CPU, RAM, program memory, I/O ports and timers.

Mnemonics

Text instructions that can be translated directly into machine code. Also referred to as Opcodes.

MPLAB ASM30

Microchip's relocatable macro assembler for dsPIC30F digital signal controller devices.

MPLAB C1X

Refers to both the MPLAB C17 and MPLAB C18 C compilers from Microchip. MPLAB C17 is the C compiler for PIC17CXXX devices and MPLAB C18 is the C compiler for PIC18CXXX and PIC18FXXXX devices.

MPLAB C30

Microchip's C compiler for dsPIC30F digital signal controller devices.

MPLAB IDE

Microchip's Integrated Development Environment.

MPLAB LIB30

MPLAB LIB30 archiver/librarian is an object librarian for use with COFF object modules created using either MPLAB ASM30 or MPLAB C30 C compiler.

MPLAB LINK30

MPLAB LINK30 is an object linker for the Microchip MPLAB ASM30 assembler and the Microchip MPLAB C30 C 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.

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.

PICmicro MCUs

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

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.

Program Counter

The location that contains the address of the instruction that is currently executing.

Program Memory

The memory area in a device where instructions are stored.

Prologue

A portion of compiler-generated code that is responsible for allocating stack space, preserving registers and performing any other machine-specific requirement specified in the runtime model. This code executes before any user code for a given function.

RAM

Random Access Memory (Data Memory). Memory in which information can be accessed in any order.

Recursive Calls

A function that calls itself, either directly or indirectly.

Relocatable

An object file whose sections have not been assigned to a fixed location in memory.

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.

Runtime Model

Describes the use of target architecture resources.

Section

A named sequence of code or data.

SFR

See Special Function Registers.

Simulator

A software program that models the operation of devices.

Source Code

The form in which a computer program is written by the programmer. Source code is written in some formal programming language which can be translated into or 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.

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.

Storage Class

Determines the lifetime of an object.

Storage Qualifier

Indicates special properties of an object (e.g., `volatile`).

Trigraphs

Three-character sequences, all starting with `??`, that are defined by ISO C as replacements for single characters.

Uninitialized Data

Data which is defined without an initial value. In C,

```
int myVar;
```

defines a variable which will reside in an uninitialized data section.

Vector

The memory locations from which an application starts execution when a specific event occurs, such as a reset or interrupt.

Warning

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.

Index

Symbols

#define	42
#ident	48
#if	35
#include	43, 44, 71, 73
#line	45
#pragma	32, 97, 152
.bss	13, 56, 97
.const	55, 56, 68
.data	13, 55, 97
.dconst	55
.dinit	55, 56
.nbss	56
.ndata	55
.ndconst	55
.pbss	56
.text	18, 27, 55, 57, 60, 97
.tmpdata	154

A

-A	42
abort	16, 100
Access Memory	152
Address Spaces	53
alias Attribute	18
aligned Attribute	12
Alignment	12, 65, 96
Anonymous Structures	152
-ansi	19, 29, 45
ANSI C Standard	9
ANSI C, Differences with MPLAB C30	11
ANSI C, Strict	30
ANSI Standard Library Support	9
ANSI-89 extension	69
Archiver	8
Arrays and Pointers	95
ASCII Character Set	157
asm	12, 87, 152
Assembler	8
Assembly Options	45
-Wa	45
Assembly, Inline	87, 152
Assembly, Mixing with C	85
attribute	11, 16, 152
Attribute, Function	16
alias	18
const	17
deprecated	19
far	18
format	17
format_arg	17
interrupt	19, 79, 82

near	18
no_instrument_function	18, 48
noreturn	16, 35
section	18, 59, 60
shadow	18, 79
unused	18
weak	18
Attribute, Variable	11
aligned	12
deprecated	15
far	14, 55, 56, 58
mode	12
near	14, 55, 56, 58
packed	13
section	13
sfr	14
space	13
transparent_union	13
unused	13
weak	14

Automatic Variable	32, 33, 62
-aux-info	29

B

-B	47, 50
Bit Fields	29, 96, 156
Bit Reversed and Modulo Addressing	68

C

-C	42
-c	28, 46
C Dialect Control Options	29
-ansi	29
-aux-info	29
-ffreestanding	29
-fno-asm	29
-fno-builtin	29
-fno-signed-bitfields	29
-fno-unsigned-bitfields	29
-fsigned-bitfields	29
-fsigned-char	29
-funsigned-bitfields	29
-funsigned-char	29
-fwritable-strings	29, 155
-traditional	19
C Heap Usage	64
C Stack Usage	62
C, Mixing with Assembly	85
Calling Conventions	154
Case Ranges	24
Cast	32, 33, 34
char	12, 29, 30, 65, 67, 69
Characters	93

MPLAB® C30 User's Guide

Code and Data Sections	55	Integer	69
Code Generation Conventions Options	47	Pointers	70
-fargument-alias	47	-dD	42
-fargument-noalias	47	Debugging Information	36
-fargument-noalias-global	47	Debugging Options	36
-fcall-saved	48	-g	36
-fcall-used	48	-Q	36
-ffixed	48	-save-temps	36
-finstrument-functions	48	Declarators	96
-fno-ident	48	Defining Global Register Variables	20
-fno-short-double	49	deprecated Attribute	15, 19, 35
-fno-verbose-asm	49	Development Tools	7
-fpack-struct	48	Device Support Files	71
-fpcc-struct-return	49	Device-Specific Header Files	51
-fshort-enums	49	Diagnostics	103
-fverbose-asm	49	Differences Between	
-fvolatile	49	MPLAB C18 and MPLAB C30	149
-fvolatile-global	49	MPLAB C30 and ANSI C	11
-fvolatile-static	49	Directories	43, 44, 45, 51
Code Size, Reduce	27, 37	Directory Search Options	47
Coding ISR's	79	-B	47, 50
COFF	7, 8, 25, 51, 72, 155	-specs=	47
Command-Line Compiler	25	-dM	42
Command-Line Options	26	-dN	42
Command-Line Simulator	7, 8, 9	Document	
Comments	30, 42	Conventions	2
Common Subexpression Elimination	17, 38, 39, 40	Layout	1
Common Subexpressions	41	Numbering Conventions	3
Compiler	8	Updates	3
Command-Line	25	double	49, 65, 67, 70, 150
Driver	8, 9, 25, 47, 51	Double-Word Integers	22
Overview	7	dsPIC-Specific Options	27
Compiler-Managed Resources	154	-mconst-in-code	27
Compiling a Single File	51	-mconst-in-data	27
Compiling Multiple Files	52	-mlarge-code	27
Complex		-mlarge-data	27
Data Types	22	-mno-pa	27
Floating Types	22	-mpa	27
Integer Types	22	-mpa=	27
Numbers	22	-msmall-code	27
complex	22	-msmall-data	27
Conditional Expression	24	-msmall-scalar	27
Conditionals with Omitted Operands	24	-msmart-io	27
Configuration Bits Setup	75	-mtext=	27
const Attribute	17	E	
Constants, Numeric	151	-E	28, 42, 44, 45, 46
Constants, String	151	EEDATA	75
CORCON	56, 71, 72	EEPROM, data	75
Customer Notification Service	5	Enabling/Disabling Interrupts	83
Customer Support	6	endian	69
D		Enumerations	96
-D	42, 43, 45	Environment	92
Data Formats	150	Environment Variables	50
Data Memory Allocation	75	PIC30_C_INCLUDE_PATH	50
Data Memory Space	13, 27, 54, 57, 64	PIC30_COMPILER_PATH	50
Data Memory Space, Near	14	PIC30_EXEC_PREFIX	50
Data Representation	69	PIC30_LIBRARY_PATH	50
Data Type	12, 69	TMPDIR	50
Complex	22	errno	100
Floating Point	70		

Error Control Options		
-pedantic-errors	30	
-Werror	34	
-Werror-implicit-function-declaration	30	
Errors	103	
Escape Sequences	93	
Exception Vectors	54, 80	
Executables	51	
exit	100	
Extensions	44	
extern	20, 35, 41, 48	
External Symbols	85	
F		
-falign-functions	38	
-falign-labels	38	
-falign-loops	38	
far Attribute	14, 18, 55, 56, 58, 88, 151	
Far Data Space	58	
-fargument-alias	47	
-fargument-noalias	47	
-fargument-noalias-global	47	
-fcaller-saves	38	
-fcall-saved	48	
-fcall-used	48	
-fcse-follow-jumps	38	
-fcse-skip-blocks	38	
-fdata-sections	38	
-fdefer-pop. See -fno-defer		
Feature Set	9	
-fexpensive-optimizations	38	
-ffixed	21, 48	
-fforce-mem	37, 41	
-ffreestanding	29	
-ffunction-sections	38	
-fgcse	38	
-fgcse-lm	39	
-fgcse-sm	39	
File Extensions	26	
File Naming Convention	26	
Files	99	
-finline-functions	19, 34, 37, 41	
-finline-limit	41	
-finstrument-functions	18, 48	
-fkeep-inline-functions	20, 41	
-fkeep-static-consts	41	
Flags, Positive and Negative	41, 47	
float	12, 49, 65, 67, 70	
Floating	70	
Floating Point	70, 94	
Floating Types, Complex	22	
-fmove-all-movables	39	
-fno	41, 47	
-fno-asm	29	
-fno-builtin	29	
-fno-defer-pop	39	
-fno-function-cse	41	
-fno-ident	48	
-fno-inline	42	
-fno-keep-static-consts	41	
-fno-peekhole	39	
-fno-peekhole2	39	
-fno-short-double	49	
-fno-show-column	42	
-fno-signed-bitfields	29	
-fno-unsigned-bitfields	29	
-fno-verbose-asm	49	
-fomit-frame-pointer	37, 42	
-foptimize-register-move	39	
-foptimize-sibling-calls	42	
format Attribute	17	
format_arg Attribute	17	
-fpack-struct	48	
-fpcc-struct-return	49	
Frame Pointer (W14)	42, 48, 62	
-freduce-all-givs	39	
-fregmove	39	
-frename-registers	39	
-frerun-cse-after-loop	39, 40	
-frerun-loop-opt	39	
-fschedule-insns	39	
-fschedule-insns2	39	
-fshort-enums	49	
-fsigned-bitfields	29	
-fsigned-char	29	
FSRn	154	
-fstrength-reduce	39, 40	
-fstrict-aliasing	37, 40	
-fsyntax-only	30	
-fthread-jumps	37, 40	
Function		
Attributes	16	
Call Conventions	65	
Calls, Preserving Registers	67	
Parameters	65	
Pointers	57	
-funroll-all-loops	37, 40	
-funroll-loops	37, 40	
-funsigned-bitfields	29	
-funsigned-char	29	
-fverbose-asm	49	
-fvolatile	49	
-fvolatile-global	49	
-fvolatile-static	49	
-fwritable-strings	29, 155	
G		
-g	36	
general registers	88	
getenv	101	
Global Register Variables	20	
Guidelines for Writing ISR's	78	
H		
-H	43	
Header Files	26, 43, 44, 45, 50	
Processor	51, 71, 73	
Heap	54	
--heap	64	
Heap, C Usage	64	
--help	28	
Hex File	52	
High-Priority Interrupts	77	

MPLAB® C30 User's Guide

I	
-I	43, 45, 50
-I-	43, 45
Identifiers	93
-idirafter	43
IEEE 754	150
-imacros	43, 45
imag	22
Implementation-Defined Behavior	91, 155
-include	43, 45
Include Files	47
Inhibit Warnings	30
Initialized Variables	55
Inline	34, 37, 41, 57, 87, 152
inline	19, 42, 48
In-Line Assembly Usage	75
Inline Functions	19
int	12, 65, 67, 69
Integer	69, 88
Behavior	94
Double-Word	22
Promotions	151
Types, Complex	22
Internet Address	4
Interrupt	
Enabling/Disabling	83
Functions	85
Handling	85
High Priority	77
Latency	83
Low Priority	77
Nesting	83
Priority	83
Request	80
Service Routine Context Saving	82
Vectors	80
Vectors, List of	80
Vectors, Writing	80
interrupt Attribute	18, 19, 79, 82, 152
-iprefix	43
IRQ	80
ISR	
Coding	79
Guidelines for Writing	78
Syntax for Writing	78
Writing	78
ISR Declaration	76
-isystem	43, 47
-iwithprefix	43
-iwithprefixbefore	44
K	
Keyword Differences	11
L	
-L	46, 47
-l	46
Labels as Values	23
Large Code Model	27, 70
Large Data Model	27, 55, 56
Latency	83
Librarian	8
Library	46, 51
ANSI Standard	9
Functions	98
Linker	8, 46
Linker Script	51, 59, 60, 61, 72, 73
Linking Options	46
-L	46, 47
-l	46
-nodfaultlibs	46
-nostdlib	46
-s	46
-u	46
-Wl	46
-Xlinker	47
little endian	69
LL, Suffix	22
Local Register Variables	20, 21
Locating Code and Data	60
long	12, 65, 67, 69
long double	12, 49, 65, 67, 70
long long	12, 34, 67, 69, 150
long long int	22
Loop Optimization	17
Loop Optimizer	39
Loop Unrolling	40
Low-Priority Interrupts	77
M	
-M	44
Mabonga	60, 153
macro	20, 42, 43, 45
Macro Names, Predefined	151
Macros	75
Configuration Bits Setup	75
In-Line Assembly Usage	75
ISR Declaration	76
MacrosData Memory Allocation	75
MATH_DATA	154
-mconst-in-code	27, 55, 56, 57
-mconst-in-data	27, 57
-MD	44
Memory	100
Memory Models	9, 57, 154
-mconst-in-code	57
-mconst-in-data	57
-mlarge-code	57
-mlarge-data	57
-msmall-code	57
-msmall-data	57
-msmall-scalar	57
Memory Spaces	57
Memory, Access	152
-MF	44
-MG	44
Microchip Internet Web Site	4
Mixing Assembly Language and C Variables and Functions	85
-mlarge-code	27, 57
-mlarge-data	27, 55, 56, 57
-MM	44
-MMD	44

-mno-pa	27
mode Attribute	12
-MP	44
-mpa	27
-mpa=	27
MPLAB C18, Differences with MPLAB C30	149
MPLAB C30	7, 9
Command Line	25
Differences with ANSI C	11
Differences with MPLAB C18	149
-MQ	44
-msmall-code	27, 57, 58
-msmall-data	27, 55, 56, 57, 58
-msmall-scalar	27, 57, 58
-msmart-io	27
-MT	45
-mtext=	27
N	
Near and Far Code	58
Near and Far Data	58
near Attribute	14, 18, 55, 56, 58, 88, 151
Near Data Section	58
Near Data Space	89
Nesting Interrupts	83
no_instrument_function Attribute	18, 48
-nodefaultlibs	46
noreturn Attribute	16, 35
-nostdinc	43, 45
-nostdlib	46
Numeric Constants	151
O	
-O	36, 37
-o	28, 51
-O0	37
-O1	37
-O2	37, 41
-O3	37
Object File	7, 8, 38, 44, 46, 51, 55
Object Module Format	155
Omitted Operands	24
Optimization	9, 155
Optimization Control Options	37
-falign-functions	38
-falign-labels	38
-falign-loops	38
-fcaller-saves	38
-fcse-follow-jumps	38
-fcse-skip-blocks	38
-fdata-sections	38
-fexpensive-optimizations	38
-fforce-mem	41
-ffunction-sections	38
-fgcse	38
-fgcse-lm	39
-fgcse-sm	39
-finline-functions	41
-finline-limit	41
-fkeep-inline-functions	41
-fkeep-static-consts	41
-fmove-all-movables	39

-fno-defer-pop	39
-fno-function-cse	41
-fno-inline	42
-fno-peephole	39
-fno-peephole2	39
-fomit-frame-pointer	42
-foptimize-register-move	39
-foptimize-sibling-calls	42
-freduce-all-givs	39
-fregmove	39
-frename-registers	39
-frerun-cse-after-loop	39
-frerun-loop-opt	39
-fschedule-insns	39
-fschedule-insns2	39
-fstrength-reduce	39
-fstrict-aliasing	40
-fthread-jumps	40
-funroll-all-loops	40
-funroll-loops	40
-O	37
-O0	37
-O1	37
-O2	37
-O3	37
-Os	37
Optimization, Loop	17, 39
Optimization, Peephole	39
Options	
Assembling	45
C Dialect Control	29
Code Generation Conventions	47
Debugging	36
Directory Search	47
dsPIC Specific	27
Linking	46
Optimization Control	37
Output Control	28
Preprocessor Control	42
Warnings and Errors Control	30
-Os	37
Output Control Options	28
-c	28
-E	28
--help	28
-o	28
-S	28
-v	28
-x	28
P	
-P	45
packed Attribute	13, 49
Parameters, Function	65
PATH	51
PC	154
-pedantic	30, 34
-pedantic-errors	30
Peephole Optimization	39
persistent data	56, 75, 154
PIC30_C_INCLUDE_PATH	50, 51

MPLAB® C30 User's Guide

PIC30_COMPILER_PATH	50	Reduce Code Size	27, 37
PIC30_EXEC_PREFIX	47, 50	References	3
PIC30_LIBRARY_PATH	50	Register	
pic30-gcc	25	Behavior	95
pointer	65, 67	Conventions	67
Pointers	35, 70, 150	Definition Files	72
Frame	42, 48	register	20, 21
Function	57	Reset	80, 83
Stack	48	Reset Vectors	54
Pragmas	152	Return Type	31
Predefined Macro Names	151	Return Value	67
prefix	43, 47	Runtime Environment	53
Preprocessing Directives	97	S	
Preprocessor	47	-S	28, 46
Preprocessor Control Options	42	-s	46
-A	42	-save-temps	36
-C	42	Scalars	57
-D	42	section	38, 55, 154
-dD	42	section Attribute	13, 18, 55, 59, 60, 152
-dM	42	Sections, Code and Data	55
-dN	42	SFR	9, 51, 54, 71, 72, 73
-fno-show-column	42	sfr Attribute	14
-H	43	shadow Attribute	18, 79, 152
-I	43	short	65, 67, 69
-I-	43	short long	150
-idirafter	43	Signals	99
-imacros	43	signed char	69
-include	43	signed int	69
-iprefix	43	signed long	69
-isystem	43	signed long long	69
-iwithprefix	43	signed short	69
-iwithprefixbefore	44	Simulator, Command-Line	7, 8, 9
-M	44	Small Code Model	9, 27, 70
-MD	44	Small Data Model	9, 27, 55, 56
-MF	44	Software Stack	18, 61, 62
-MG	44	space Attribute	13, 151, 152
-MM	44	Special Function Registers	51, 71, 82
-MMD	44	Specifying Registers for Local Variables	21
-MQ	44	-specs=	47
-MT	45	SPLIM	61
-nostdinc	45	SR	154
-P	45	Stack	54, 82, 83
-trigraphs	45	C Usage	62
-U	45	Pointer (W15)	48, 56, 61, 62
-undef	45	Pointer Limit Register (SPLIM)	56, 61
Preserving Registers Across Function Calls	67	Software	61, 62
Procedural Abstraction	27, 155	Usage	150
Processor Header Files	51, 71, 73	Standard I/O Functions	9
PROD	154	Startup	
Program Memory Pointers	57	and Initialization	56
Program Memory Space	13, 27, 54, 57	Code	154
Program Space Visibility Window. See PSV Window		Module, Alternate	56
PSV Usage	68	Module, Primary	56
PSV Window	54, 55, 57, 68, 71	Modules	62
Q		Statement Differences	23
-Q	36	Statements	96
Qualifiers	96	static	49
R		STATUS	154
RCOUNT	154	Storage Classes	150
real	22	Storage Qualifiers	151

Streams	99
strerror	101
String Constants	151
Strings	29
structure	65
Structures	96
Structures, Anonymous	152
Suffix LL	22
Suffix ULL	22
switch	32
symbol	46
Syntax Check	30
Syntax for Writing ISR's	78
system	101
System Header Files	32, 44
T	
-T	72
TABLAT	154
TBLPTR	154
TMPDIR	50
tmpfile	100
-traditional	19, 29
Traditional C	35
Translation	92
transparent_union Attribute	13
Trigraphs	32, 45
-trigraphs	45
Troubleshooting	4
Type Conversion	34
typeof	22
U	
-U	42, 43, 45
-u	46
ULL, Suffix	22
-undef	45
Underscore	78, 85
Uninitialized Variables	56
Unions	96
Unroll Loop	40
unsigned char	69
unsigned int	69
unsigned long	69
unsigned long long	69
unsigned long long int	22
unsigned short	69
unused Attribute	13, 18, 32
Unused Function Parameter	32
Unused Variable	32
User-Defined Data Section	60
User-Defined Text Section	60
Using Inline Assembly Language	87
Using Macros	75
Using SFRs	73
V	
-v	28
Variable Attributes	11
Variables in Specified Registers	20
Vectors, Reset and Exception	54
void	67
volatile	49

W	
-W	30, 32, 33, 35, 103
-w	30
W Registers	65, 85
W14	62, 154
W15	62, 154
-Wa	45
-Waggregate-return	34
-Wall	30, 32, 33, 36
Warnings	126
Warnings and Errors Control Options	30
-fsyntax-only	30
-pedantic	30
-pedantic-errors	30
-W	33
-w	30
-Waggregate-return	34
-Wall	30
-Wbad-function-cast	34
-Wcast-align	34
-Wcast-qual	34
-Wchar-subscripts	30
-Wcomment	30
-Wconversion	34
-Wdiv-by-zero	30
-Werror	34
-Werror-implicit-function-declaration	30
-Wformat	30
-Wimplicit	30
-Wimplicit-function-declaration	30
-Wimplicit-int	30
-Winline	34
-Wlarger-than-	34
-Wlong-long	34
-Wmain	30
-Wmissing-braces	30
-Wmissing-declarations	34
-Wmissing-format-attribute	34
-Wmissing-noreturn	35
-Wmissing-prototypes	35
-Wmultichar	31
-Wnested-externs	35
-Wno-long-long	34
-Wno-multichar	31
-Wno-sign-compare	35
-Wpadded	35
-Wparentheses	31
-Wpointer-arith	35
-Wredundant-decls	35
-Wreturn-type	31
-Wsequence-point	31
-Wshadow	35
-Wsign-compare	35
-Wstrict-prototypes	35
-Wswitch	32
-Wsystem-headers	32
-Wtraditional	35
-Wtrigraphs	32
-Wundef	35
-Wuninitialized	32

-Wunknown-pragmas	32	-Wno-	30
-Wunreachable-code	35	-Wno-deprecated-declarations	35
-Wunused	32	-Wno-div-by-zero	30
-Wunused-function	32	-Wno-long-long	34
-Wunused-label	32	-Wno-multichar	31
-Wunused-parameter	33	-Wno-sign-compare	33, 35
-Wunused-value	33	-Wpadded	35
-Wunused-variable	33	-Wparentheses	31
-Wwrite-strings	36	-Wpointer-arith	35
Warnings, Inhibit	30	-Wredundant-decls	35
-Wbad-function-cast	34	WREG	154
-Wcast-align	34	-Wreturn-type	31
-Wcast-qual	34	Writing an Interrupt Service Routine	78
-Wchar-subscripts	30	Writing the Interrupt Vector	80
-Wcomment	30	-Wsequence-point	31
-Wconversion	34	-Wshadow	35
-Wdiv-by-zero	30	-Wsign-compare	35
weak Attribute	14, 18	-Wstrict-prototypes	35
-Werror	34	-Wswitch	32
-Werror-implicit-function-declaration	30	-Wsystem-headers	32
-Wformat	17, 30, 34	-Wtraditional	35
-Wimplicit	30	-Wtrigraphs	32
-Wimplicit-function-declaration	30	-Wundef	35
-Wimplicit-int	30	-Wuninitialized	32
-Winline	19, 34	-Wunknown-pragmas	32
-Wl	46	-Wunreachable-code	35
-Wlarger-than-	34	-Wunused	32, 33
-Wlong-long	34	-Wunused-function	32
-Wmain	30	-Wunused-label	32
-Wmissing-braces	30	-Wunused-parameter	33
-Wmissing-declarations	34	-Wunused-value	33
-Wmissing-format-attribute	34	-Wunused-variable	33
-Wmissing-noreturn	35	-Wwrite-strings	36
-Wmissing-prototypes	35	WWW Address	4
-Wmultichar	31	X	
-Wnested-externs	35	-x	28
		X and Y Data Spaces	59
		-Xlinker	47

NOTES:



WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: <http://www.microchip.com>

Atlanta

3780 Mansell Road, Suite 130
Alpharetta, GA 30022
Tel: 770-640-0034
Fax: 770-640-0307

Boston

2 Lan Drive, Suite 120
Westford, MA 01886
Tel: 978-692-3848
Fax: 978-692-3821

Chicago

333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071
Fax: 630-285-0075

Dallas

4570 Westgrove Drive, Suite 160
Addison, TX 75001
Tel: 972-818-7423
Fax: 972-818-2924

Detroit

Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo

2767 S. Albright Road
Kokomo, IN 46902
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles

18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888
Fax: 949-263-1338

Phoenix

2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7966
Fax: 480-792-4338

San Jose

1300 Terra Bella Avenue
Mountain View, CA 94043
Tel: 650-215-1444

Toronto

6285 Northam Drive, Suite 108
Mississauga, Ontario L4V 1X5, Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Australia

Suite 22, 41 Rawson Street
Epping 2121, NSW
Australia
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing

Unit 706B
Wan Tai Bei Hai Bldg.
No. 6 Chaoyangmen Bei Str.
Beijing, 100027, China
Tel: 86-10-85282100
Fax: 86-10-85282104

China - Chengdu

Rm. 2401-2402, 24th Floor,
Ming Xing Financial Tower
No. 88 TIDU Street
Chengdu 610016, China
Tel: 86-28-86766200
Fax: 86-28-86766599

China - Fuzhou

Unit 28F, World Trade Plaza
No. 71 Wusi Road
Fuzhou 350001, China
Tel: 86-591-7503506
Fax: 86-591-7503521

China - Hong Kong SAR

Unit 901-6, Tower 2, Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Shanghai

Room 701, Bldg. B
Far East International Plaza
No. 317 Xian Xia Road
Shanghai, 200051
Tel: 86-21-6275-5700
Fax: 86-21-6275-5060

China - Shenzhen

Rm. 1812, 18/F, Building A, United Plaza
No. 5022 Binhe Road, Futian District
Shenzhen 518033, China
Tel: 86-755-82901380
Fax: 86-755-8295-1393

China - Shunde

Room 401, Hongjian Building
No. 2 Fengxiangnan Road, Ronggui Town
Shunde City, Guangdong 528303, China
Tel: 86-765-8395507 Fax: 86-765-8395571

China - Qingdao

Rm. B505A, Fullhope Plaza,
No. 12 Hong Kong Central Rd.
Qingdao 266071, China
Tel: 86-532-5027355 Fax: 86-532-5027205

India

Divyasree Chambers
1 Floor, Wing A (A3/A4)
No. 11, O'Shaughnessey Road
Bangalore, 560 025, India
Tel: 91-80-2290061 Fax: 91-80-2290062

Japan

Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa, 222-0033, Japan
Tel: 81-45-471-6166 Fax: 81-45-471-6122

Korea

168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea 135-882
Tel: 82-2-554-7200 Fax: 82-2-558-5932 or
82-2-558-5934

Singapore

200 Middle Road
#07-02 Prime Centre
Singapore, 188980
Tel: 65-6334-8870 Fax: 65-6334-8850

Taiwan

Kaohsiung Branch
30F - 1 No. 8
Min Chuan 2nd Road
Kaohsiung 806, Taiwan
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan

Taiwan Branch
11F-3, No. 207
Tung Hua North Road
Taipei, 105, Taiwan
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

Austria

Durisolstrasse 2
A-4600 Wels
Austria
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

Denmark

Regus Business Centre
Lautrup høj 1-3
Ballerup DK-2750 Denmark
Tel: 45-4420-9895 Fax: 45-4420-9910

France

Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - 1er Etage
91300 Massy, France
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany

Steinheilstrasse 10
D-85737 Ismaning, Germany
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy

Via Quasimodo, 12
20025 Legnano (MI)
Milan, Italy
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands

P. A. De Biesbosch 14
NL-5152 SC Drunen, Netherlands
Tel: 31-416-690399
Fax: 31-416-690340

United Kingdom

505 Eskdale Road
Womersley Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44-118-921-5869
Fax: 44-118-921-5820

11/24/03